Vulcanexus is a ROS 2 (Robot Operating System) all-in-one tool set. It allows users to build robotics applications combining the unique Vulcanexus elements with the ROS 2 libraries, having Fast DDS as its fixed middleware implementation.

These open source elements include numerous features and tools, providing Vulcanexus users customizable solutions while improving overall system performance. With Vulcanexus, users have fast access to constantly improving functionalities, such as the latest Fast DDS version along with its new features.
GETTING STARTED

• **Installation Manual**
  – Instructions to set up ROS 2 with Vulcanexus for the first time.

• **Overview**
  – *Vulcanexus* concepts including *ROS 2* and *micro-ROS* Documentation.

• **Tutorials**
  – Collection of step-by-step instructions that help users to get started.

• **Support**
  – Vulcanexus officially supported platforms and releases.

• **Appendixes**
  – Glossary of terms used throughout this documentation.
VULCANEXUS PROJECT

Vulcanexus is composed of a collection of downloadable packages:

1. **Vulcanexus Core**: a set of software libraries that enables users to build the most comprehensive and straightforward robotics application. It consists of eProsima Fast DDS latest version as the ROS 2 middleware.

2. **Vulcanexus Tools**: a set of features and applications which allows users to test, improve and configure the performance of Vulcanexus in their systems.

3. **Vulcanexus Micro**: provides access for resource constrained devices (micro-controllers) to the DDS world, bridging the gap between them and ROS 2.

4. **Vulcanexus Cloud**: scales and integrates ROS 2 networks located in geographically spaced environments, and enables the deployment of DDS entities in the cloud in a quick and easy way.

5. **Vulcanexus Simulation**: enables users to design robotic simulations, providing an end-to-end development environment to model, program, and simulate robots.

6. **Vulcanexus Base**: includes within the same installation Vulcanexus tools, micro and cloud features.

7. **Vulcanexus Desktop**: includes ROS 2 desktop installation as well as every available Vulcanexus package. It is the more complete Vulcanexus installation.

The table below shows the components included in every one of the previously explained packages:
CONTACTS AND COMMERCIAL SUPPORT

Find more about us at Vulcanexus webpage.
Support available at:

- Email: support@eprosima.com
- Phone: +34 91 804 34 48
Vulcanexus Docs is an open source project, and as such all contributions, both in the form of feedback and content generation, are most welcomed. To make such contributions, please refer to the Contribution Guidelines hosted in our GitHub repository.

4.1 Linux binary installation

Debian packages for Vulcanexus Iron Imagination are currently available for Ubuntu Jammy. Since Vulcanexus is a ROS 2 all-in-one tool set, certain ROS 2 prerequisites need to be met before installing.

4.1.1 ROS 2 prerequisites

First of all, set up a UTF-8 locale as required by ROS 2. Locale settings can be checked and set up with the following commands:

```bash
locale  # check for UTF-8
sudo apt update && sudo apt install -y locales
# Any UTF-8 locale will work. Using en_US as an example
sudo locale-gen en_US en_US.UTF-8
sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
export LANG=en_US.UTF-8
```

ROS 2 also requires that the Ubuntu Universe repository is enabled. This can be checked and enabled with the following commands:

```bash
apt-cache policy | grep universe
# This should print something similar to:
# # 500 http://us.archive.ubuntu.com/ubuntu jammy/universe amd64 Packages
# release v=22.04,o=Ubuntu,a=jammy,n=jammy,l=Ubuntu,c=universe,b=amd64
# # Otherwise run

sudo apt install -y software-properties-common
sudo add-apt-repository universe -y
```

Now download ROS 2 GPG key into the keystore.
4.1.2 Setup Vulcanexus sources

Once all ROS 2 prerequisites have been met, it is time to set up Vulcanexus.

Add Vulcanexus GPG key so apt can retrieve the packages:

```
```

Next, add the eProsima Vulcanexus repository to the repository manager sources list:

```
echo "deb [arch=$(
sudo apt update &

Desktop install (Recommended): includes all the simulation tools, demos, and tutorials.

```
sudo apt install -y vulcanexus-iron-desktop
```

Base Install: basic installation without simulation tools, demos, and tutorials.

```
sudo apt install -y vulcanexus-iron-base
```

For other Vulcanexus packages, please refer to Vulcanexus Metapackages for more information.

4.1.4 Environment setup

In order to use the Vulcanexus installation, the environment must be set up sourcing the following file:

```
source /opt/vulcanexus/iron/setup.bash
```
4.1.5 Try some examples

In one terminal, source the setup file and then run a C++ talker:

```
. ~/vulcanexus_iron/install/local_setup.bash
ros2 run demo_nodes_cpp talker
```

In another terminal source the setup file and then run a Python listener:

```
. ~/vulcanexus_iron/install/local_setup.bash
ros2 run demo_nodes_py listener
```

At this point, the talker should prompt Publishing: Hello World: <id> messages while the listener echoes I heard: [Hello World: <id>]. This verifies both the C++ and Python APIs are working properly. Hooray!

4.1.6 Uninstall eProsima Vulcanexus packages

To uninstall Vulcanexus, it is enough to run the following command:

```
sudo apt autoremove vulcanexus-iron-desktop
```

4.2 Linux installation from sources

This section explains how to build Vulcanexus in Ubuntu Jammy. Since Vulcanexus is a ROS 2 all-in-one tool set, certain ROS 2 prerequisites need to be met before building.

4.2.1 ROS 2 prerequisites

First of all, set up a UTF-8 locale as required by ROS 2. Locale settings can be checked and set up with the following commands:

```
locale  # check for UTF-8

sudo apt update && sudo apt install -y locales
# Any UTF-8 locale will work. Using en_US as an example
sudo locale-gen en_US en_US.UTF-8
sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
export LANG=en_US.UTF-8
```

ROS 2 also requires that the Ubuntu Universe repository is enabled. This can be checked and enabled with the following commands:

```
apt-cache policy | grep universe

# This should print something similar to:
#
# 500 http://us.archive.ubuntu.com/ubuntu jammy/universe amd64 Packages
# release v=22.04,o=Ubuntu,a=jammy,n=jammy,l=Ubuntu,c=universe,b=amd64
```

(continues on next page)
# Otherwise run

```bash
sudo apt install -y software-properties-common
sudo add-apt-repository universe -y
```

Now download ROS 2 GPG key into the keystore.

```bash
sudo apt update && sudo apt install -y curl gnupg lsb-release
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o /usr/share/keyrings/ros-archive-keyring.gpg
```

And then add ROS 2 repository to the repository manager sources list.

```bash
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-archive-keyring.gpg] http://packages.ros.org/ros2/ubuntu $(source /etc/os-release && echo "$UBUNTU_CODENAME") main" | sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null
```

With the ROS 2 repository properly set up the next step is to install the required dependencies and tools for cloning and testing the ROS 2 packages within the workspace.

```bash
sudo apt update && sudo apt install -y build-essential cmake git
python3-colcon-common-extensions python3-flake8 python3-flake8-blind-except python3-flake8-builtin python3-flake8-class-newline python3-flake8-comprehensions python3-flake8-deprecated python3-flake8-docstrings python3-flake8-import-order python3-flake8-quotes python3-pip python3-pytest python3-pytest-cov python3-pytest-repeat python3-pytest-rerunfailures python3-rosdep python3-setuptools python3-vcstool wget
```

Chapter 4. Contributing to the documentation
4.2.2 Get ROS 2 code

Create a workspace for Vulcanexus and clone the ROS 2 repositories

```
mkdir -p ~/vulcanexus_iron/src
cd ~/vulcanexus_iron
wget https://raw.githubusercontent.com/ros2/ros2/iron/ros2.repos
vcs import src < ros2.repos
```

Now download the required dependencies for these packages.

```
sudo apt upgrade -y
sudo rosdep init
rosdep update
rosdep install --from-paths src --ignore-src -y --skip-keys "fastcdr rti-connext-dds-6.0.1 urdfdom_headers"
```

4.2.3 Get Vulcanexus code

Add the Vulcanexus repositories and metadata files to the Vulcanexus workspace:

```
cd ~
cd vulcanexus_iron
# Remove ROS 2 packages overridden by Vulcanexus
rm -rf src/ros2/rosidl_typesupport_fastrtps/ src/eProsima/foonathan_memory_vendor/ src/...ros2/rmw_fastrtps/
# Get Vulcanexus sources
wget https://raw.githubusercontent.com/eProsima/vulcanexus/iron/vulcanexus.repos
wget https://raw.githubusercontent.com/eProsima/vulcanexus/iron/colcon.meta
vcs import --force src < vulcanexus.repos
```

4.2.4 Install Vulcanexus dependencies

Some additional dependencies which are required for the Vulcanexus distribution must be installed. Install the Vulcanexus required development tools with the following command:

```
sudo apt update && sudo apt install -y
   libasio-dev
   libengne-pkcs11-openssl
   liblog4cxx-dev
   libp11-dev
   libqt5charts5-dev
   libssl-dev
   libtinyxml2-dev
   libyaml-cpp-dev
   openjdk-8-jdk
   python3-sphinx
   python3-sphinx-rtd-theme
   qtbase5-dev
   qdeclarative5-dev
   qquickcontrols2-5-dev
   swig
```
4.2.5 Build the code in the workspace

If any other Vulcanexus or ROS 2 distribution has been installed from binaries, please ensure that the build is done in a fresh environment (previous installation is not sourced). This can be checked running the following command:

```bash
printenv | grep 'VULCANEXUS\|ROS'
```

The output should be empty. Please, be aware that in case the environment sourcing has been added to .bashrc, it must be removed in order to get a fresh environment.

**Build Fast DDS-Gen (Optional)**

Fast DDS-Gen is a Java application that generates source code using the data types defined in an IDL file. This tool must be built separately following the instructions below. Please, refer to Fast DDS-Gen documentation for more information about this tool.

```bash
cd src/eProsima/Fast-DDS-Gen
./gradlew assemble
```

The generated Java application can be found in `src/eProsima/Fast-DDS-Gen/share/fastddsgen`. However, the scripts folder provides some user friendly scripts that are recommended to be used. This scripts can be made accessible to the session adding the `scripts` folder path to the `PATH` environment variable.

```bash
export PATH=~/vulcanexus_iron/src/eProsima/Fast-DDS-Gen/scripts:$PATH
```

**Build workspace**

In order to build the workspace, the command line tool colcon is used. This tool is based on CMake and it is aimed at building sets of software packages, handling ordering and setting up the environment to use them.

```bash
cd ~/vulcanexus_iron
colcon build
```

**Important:** In case that only a set of packages are going to be built, please ensure to include always `vulcanexus_base` package in the set. E.g.:

```bash
colcon build --packages-up-to demo_nodes_cpp vulcanexus_base
```

This auxiliary package is required to set several environment variables required by the distribution such as `VULCANEXUS_DISTRO` and `VULCANEXUS_HOME`. 

4.2.6 Environment setup

In order to use the Vulcanexus installation, the environment must be set up sourcing the following file:

```bash
source ~/vulcanexus_iron/install/setup.bash
```

4.2.7 Try some examples

In one terminal, source the setup file and then run a C++ talker:

```bash
. ~/vulcanexus_iron/install/local_setup.bash
ros2 run demo_nodes_cpp talker
```

In another terminal source the setup file and then run a Python listener:

```bash
. ~/vulcanexus_iron/install/local_setup.bash
ros2 run demo_nodes_py listener
```

At this point, the talker should prompt Publishing: Hello World: <id> messages while the listener echoes I heard: [Hello World: <id>]. This verifies both the C++ and Python APIs are working properly. Hooray!

4.3 Docker installation

Vulcanexus offers the possibility of running from a containerized environment by providing a Docker image which contains Vulcanexus’s Desktop installation. This Docker image can be found in Vulcanexus’s Downloads. To run it, first install Docker:

```bash
sudo apt install docker.io
```

And then load the image with:

```bash
docker load -i ubuntu-vulcanexus-iron-desktop.tar
```

Vulcanexus Docker image can be run with:

**GUI support**

```bash
xhost local:root
docker run \\n    -it \\
    --privileged \\
    -e DISPLAY=$DISPLAY \\
    -v /tmp/.X11-unix:/tmp/.X11-unix \\
    ubuntu-vulcanexus:iron-desktop
```

**CLI support**

```bash
docker run -it ubuntu-vulcanexus:iron-desktop
```

To run more than one session within the same container, Vulcanexus installation must be sourced. Given a running container, you can open another session by:
docker exec -it <running-container-id> bash

Then, within the container, source the Vulcanexus installation with:

```
source /opt/vulcanexus/iron/setup.bash
```

To verify that the sourcing was correct, run:

```
echo $VULCANEXUS_HOME
```

The output should be:

```
/opt/vulcanexus/iron
```

## 4.4 Vulcanexus Overview

Vulcanexus, the ROS 2 all-in-one tool set, expands the ROS 2 environment improving the developer experience with ROS 2 providing open source features and tools not available with the current ROS 2 release. Consequently, Vulcanexus shares the underlying concepts, principles and architecture of ROS 2. For more information about ROS 2 please refer to the Robot Operating System 2: Design, architecture, and uses in the wild\(^1\). This section will follow and summarize the most important points contained in this paper, pointing out the differences and improvements that Vulcanexus provides.

### 4.4.1 Scope

ROS 2 provides a software ecosystem to develop robotics applications (or also known as a Software Development Kit or SDK). It is not an Operating System (OS) in the traditional sense, but a framework that provides a huge range of libraries and tools intended to ease development for a wide variety of devices using several different technologies by providing a unified set of APIs and conventions. The main categories within this ecosystem are:

**Middleware**

The middleware layer is charged with the communication between components. After considering several communication protocols the Object Management Group (OMG) DDS (Data Distribution Service) protocol was selected for being an open standard with a security extension, having a distributed dynamic discovery and being highly customizable for every kind of application. Several DDS vendors, both open source and licensed, are available within the ROS 2 environment. However, Vulcanexus supports eProsima’s Fast DDS open source implementation. The main advantage being that Vulcanexus releases are not tied to a specific Fast DDS version which is enforced by the ROS 2 ROS Enhancement Proposal (REP) 2004 i.vi. requirement:

> **Must have a policy that keeps API and ABI stability within a released ROS distribution.**

Consequently, only patch releases that fix bugs without modifying API and ABI are considered for updates in the ROS 2 environment. Vulcanexus, on the other hand, updates the middleware layer any time eProsima Fast DDS releases a new version in order to benefit from the latest features and fixes. Vulcanexus takes charge of releasing a new binary stable distribution that is compatible with the potential ABI breaks so the Vulcanexus user has only to update its Vulcanexus distro to benefit from the latest Fast DDS release. For instance, Vulcanexus provides PKCS #11 security support whereas ROS 2 does not support it (as of January 2023).

---

Algorithms

Another category includes robotic application’s algorithms. **Vulcanexus**, being a **ROS 2** all-in-one tool set, provides the algorithms included in **ROS 2** ecosystem.

Developer Tools

**ROS 2** provides several development tools for debugging, logging, visualization, introspection, simulation, etc. **Vulcanexus** adds to **ROS 2** toolset the following tools:

- **ROS 2 Monitor**: graphical desktop application to monitor **ROS 2** communications.
- **Fast DDS Statistics Backend**: **ROS 2 Monitor**’s backend. Can be leveraged with other monitoring frontend applications (e.g. **Prometheus**).
- **ROS 2 Shapes Demo**: first demo application to understand the most used **ROS 2** Quality of Service (QoS) and test DDS and **ROS 2** communication.
- **ROS 2 Router**: end-user software application enabling the connection of distributed **ROS 2** environments.
- **Webots**: open-source three-dimensional mobile robot simulator.

**Note**: This documentation provides several tutorials showcasing the capabilities and advantages of **Vulcanexus**.

4.4.2 ROS 2 Architecture

**ROS 2** architecture is based on several abstraction layers. Developers usually interact exclusively with the client libraries, which expose the core communication APIs. Each one developed in a specific programming language to leave users freedom to choose the one that best applies to its application. Below, a common interface, **rcl**, connects with the **ROS** MiddleWare (RMW) layer where the essential communication APIs are defined. Each DDS vendor provides its specific RMW implementation using their own DDS library. **Vulcanexus**, supporting **eProsima Fast DDS**, constantly updates the **rmw_fastrtps** library (Fast DDS previously was know as Fast-RTPS) in order to provide the latest features included in Fast DDS library to the **Vulcanexus** community, instead of having to wait to the next **ROS 2** release.

For more information about **ROS 2**, the user is encouraged to read **ROS 2** documentation which is included within **Vulcanexus** documentation in the following section, especially the Concepts chapter.

4.5 Vulcanexus Middleware

**Vulcanexus** supports **eProsima Fast DDS** as the official middleware. **Fast DDS** provides many features of interest to the development of robotics applications, providing also custom solutions to well-known problems as, for instance, discovery issues for large deployments. This section provides an introduction to **Fast DDS** and its main advantages. More information can be found in **Fast DDS documentation**.
4.5.1 Fast DDS

eProsima Fast DDS is a C++ implementation of the DDS (Data Distribution Service) Specification, a protocol defined by the Object Management Group (OMG). The eProsima Fast DDS library provides both an Application Programming Interface (API) and a communication protocol that deploy a Data-Centric Publisher-Subscriber (DCPS) model, with the purpose of establishing efficient and reliable information distribution among Real-Time Systems. eProsima Fast DDS is predictable, scalable, flexible, and efficient in resource handling. For meeting these requirements, it makes use of typed interfaces and hinges on a many-to-many distributed network paradigm that neatly allows separation of the publisher and subscriber sides of the communication.

Main Features

• **Real-Time behaviour**: eProsima Fast DDS can be configured to offer real-time features, guaranteeing responses within specified time constrains.

• **Built-in Discovery Server**: eProsima Fast DDS is based on the dynamical discovery of existing publishers and subscribers, and performs this task continuously without the need to contacting or setting any servers. However, a Client-Server discovery as well as other discovery paradigms can also be configured. See [ROS 2 Discovery Server](#) for more information. Several tutorials are available showcasing different discovery mechanisms and strategies: discovery server, initial peers, and static EDP discovery.

• **Sync and Async publication modes**: eProsima Fast DDS supports both synchronous and asynchronous data publication. See [Changing publication mode](#) and [Mixing synchronous and asynchronous publications in the same node](#) tutorials for more information.

• **Best effort and reliable communication**: eProsima Fast DDS supports an optional reliable communication paradigm over Best Effort communications protocols such as UDP. Furthermore, another way of setting a reliable communication is to use the also supported TCP transport.

• **Transport layers**: eProsima Fast DDS implements an architecture of pluggable transports: UDPv4, UDPv6, TCPv4, TCPv6 and SHM (shared memory). See [Customizing Network Transports](#) tutorial for more information.

• **Security**: eProsima Fast DDS can be configured to provide secure communications. For this purpose, it implements pluggable security at three levels: authentication of remote participants, access control of entities and encryption of data. Fast DDS also supports hardware secure modules implementing PKCS#11 as it is showcased in [Storing private keys in Hardware Secure Modules (HSM)](#) tutorial.

• **Statistics Module**: eProsima Fast DDS can be configured to gather and provide information about the data being exchanged by the user application. Several tutorials are available showcasing different options to obtain the statistics within Vulcanexus ecosystem: [ROS 2 Monitor](#), [Fast DDS Statistics Backend](#), SQL, and Prometheus.

• **Plug-and-play Connectivity**: New applications and services are automatically discovered, and can join and leave the network at any time without the need for reconfiguration.

• **Scalability and Flexibility**: DDS builds on the concept of a global data space. The middleware is in charge of propagating the information between publishers and subscribers. This guarantees that the distributed network is adaptable to reconfigurations and scalable to a large number of entities.

• **Application Portability**: The DDS specification includes a platform specific mapping to IDL, allowing an application using DDS to switch among DDS implementations with only a re-compile.

• **Extensibility**: eProsima Fast DDS allows the protocol to be extended and enhanced with new services without breaking backwards compatibility and interoperability.

• **Configurability and Modularity**: eProsima Fast DDS provides an intuitive way to be configured, either through code or XML profiles. Modularity allows simple devices to implement a subset of the protocol and still participate in the network. See [Configuring Fast-DDS QoS via XML profiles](#) tutorial for more information.
• **High performance:** *eProsima Fast DDS* uses a static low-level serialization library, *Fast CDR*, a C++ library that serializes according to the standard CDR serialization mechanism defined in the *RTPS Specification* (see the Data Encapsulation chapter as a reference).

• **Easy to use:** The interactive demo *ROS 2 Shapes Demo* is available for the user to dive into the DDS world. See *Hand-on ROS 2 Shapes Demo* tutorial for more information.

• **Low resources consumption:** *eProsima Fast DDS*:
  - Allows to preallocate resources, to minimize dynamic resource allocation.
  - Avoids the use of unbounded resources.
  - Minimizes the need to copy data.

• **Multi-platform:** The OS dependencies are treated as pluggable modules. Users may easily implement platform modules using the *eProsima Fast DDS* library on their target platforms. By default, the project can run over Linux, Windows and MacOS.

• **Free and Open Source:** Both *eProsima Fast DDS*, its internal dependencies (such as *eProsima Fast CDR*), and its external ones (such as the *foonathan* library) are free and open source.

### 4.5.2 ROS 2 Discovery Server

*ROS 2 Discovery Server* is a Fast DDS enabled feature that procures an alternative discovery mechanism to the default ROS 2 discovery mechanism, *Simple Discovery Protocol (SDP)*, which is served by the DDS implementations according to the DDS specification. Whereas SDP provides automatic out-of-the-box discovery by leveraging multicast, ROS 2 Discovery Server provides a centralized hub for managing discovery which drastically reduces network bandwidth utilization when compared to SDP, since the nodes, publishers, and subscribers, only discovered those remote ROS 2 entities with which they need to communicate (as opposed to the SDP model where everyone knows about each other). Furthermore, it does not relay on multicast, which makes this mechanism more robust over WiFi, and simplifies ROS 2 deployments in managed networks, where the use of multicast is often restricted. Its main features are:

- **Ease of use:** Vulcanexus (through Fast DDS) provides a CLI to instantiate Discovery Servers with one command. To connect a node (Client) to the Discovery Server, a simple environment variable is used (much like in ROS 1).

- **Scalability:** The discovery related network traffic can be reduced by more than an 85 % margin when compared to SDP.

- **Robustness:** ROS 2 Discovery Server supports redundant servers, effectively removing the single point of failure that its ROS 1 counterpart (ROS Master) entailed.

- **Run-time mutability:** It is possible to change the Server to which a node (Client) connects on run-time.

- **Ease of deployment:** All that is necessary to get rid of all ROS 2 discovery related problems during deployment (WiFi, multicast, bandwidth exhaustion, etc.) is a process to run the Discovery Server, and an environment variable to configure the Clients.

---

**Note:** Please refer to the *Discovery Server documentation* for more information on all possible Discovery Server configuration options and use-cases.
4.6 Developer Tools

ROS 2 provides several development tools for debugging, logging, visualization, introspection, simulation, etc. **Vulcanexus** adds to ROS 2 toolset the following tools:

4.6.1 ROS 2 QoS Profiles Manager

**ROS 2 QoS Profiles Manager** is a tool suite that facilitates the generation of XML configuration files for Vulcanexus middleware, Fast DDS. The suite provides with both a Graphical User Interface (GUI) and a Command Line Interface (CLI). The user can generate new configuration files from scratch, modify already existing configurations, print specific configurations from a profiles file, validate previously generated configurations, etc.

Next Steps

Visit [eProsima Fast DDS QoS Profiles Manager Documentation](#) for more information about this tools suite.

4.6.2 ROS 2 Monitor

**ROS 2 Monitor** is a graphical desktop application shipped with **Vulcanexus Tools** aimed at monitoring DDS environments deployed using the **eProsimat** library, as it is the case in ROS 2 ecosystems. Thus, the user can monitor in real time the status of publication/subscription communications between ROS 2 nodes. They can also choose from a wide variety of communication parameters to be measured (latency, throughput, packet loss, etc.), as well as record and compute in real time statistical measurements on these parameters (mean, variance, standard deviation, etc.).

Furthermore, the user can check the status of the deployed ROS 2 network at any time, i.e. see for each DDS Domain which DomainParticipants are instantiated, as well as their publishers and subscribers and the topics under which they publish or to which they subscribe respectively. It is also possible to see the physical architecture of the network on which the ROS 2 applications are running.

Overview

**ROS 2 Monitor** is designed to meet the following criteria:

1. **Monitoring**: real-time tracking of network status and communication.
2. **Intuitive**: graphical user interface developed following a user experience design approach.
3. **Introspection**: easily navigate through the deployed and active ROS 2 nodes being able to inspect their configuration and physical deployment.
4. **Troubleshooting**: detect at a glance the possible issues or anomalous events that may occur in the communication.

**Warning**: If having explicitly enabled statistics and statistical data are still not correctly received, only few data arrive or even none, configure the ROS 2 nodes publishing statistics data with a less restrictive memory constraints. Please check the following documentation for more details on how to do this.
Next Steps

Visit eProsima Fast DDS Monitor Documentation for more information on how to use this application. A hands-on example is also available in this link, which provides step-by-step instructions on how to use Vulcanexus Tools for monitoring a ROS 2 talker/listener demo.

4.6.3 Fast DDS Statistics Backend

Note: This page is under maintenance and will be updated soon.

4.6.4 ROS 2 Shapes Demo

ROS 2 Shapes Demo is an application in which Publishers and Subscribers are shapes of different colors and sizes moving on a board. Each shape refers to its own topic: Square, Triangle or Circle. A single instance of the ROS 2 Shapes Demo can publish on or subscribe to several topics at a time.

It demonstrates the capabilities of eProsima Fast DDS or as a proof of interoperability with other DDS-compliant implementations.

Running ROS 2 Shapes Demo

ROS 2 Shapes Demo application is available in the Vulcaenxus Tools, Vulcanexus Base and Vulcanexus Desktop packages. Regardless of the installation method used, to run ROS 2 Shapes Demo, load the Vulcanexus environment first, and then run shapesdemo.

```
source /opt/vulcanexus/iron/setup.bash
shapesdemo
```

Warning: Please verify that you are running the ROS 2 Shapes Demo version. To check this simply verify that the Vulcanexus banner is displayed in the background of the main box. If the eProsima logo appears instead, activate the ROS 2 version in Options > Participant Configuration > Use ROS 2 Topics.

Getting started

After the executable is launched, a window similar to the one presented in the following image should be displayed.

Publishing a Topic

The Publish button allow the users to define the Shape (topic) and Quality of Service for their publication. The following image shows an example of the Publication menu.

There are multiple parameters that the user can define in this menu:

- **Shape**: This parameter defines the topic where the publication is going to occur. Three different shapes can be published: Square, Circle and Triangle (see Fast DDS Topic Documentation).
- **Color**: The user can define the color of the shape.
- **Size**: This parameter allows to control the size of the shape. The size can vary between 1 and 99.
Chapter 4. Contributing to the documentation
Publish (on 81cfd4305fa1)

**Shape**
- Square

**Color**
- RED

**Size**
- 30

**Partition**
- A
- B
- C
- D

**QoS Settings**

**History:** 1

**Durability:** VOLATILE

**Reliability**
- ✅ Reliable

**Liveliness**

**Kind:** AUTOMATIC

**Lease Duration (ms):** INF

**Announcement Period (ms):** INF

**Ownership**

**Kind:** SHARED_OWNERSHIP

**Ownership Strength:** 0

**Deadline**

**Duration (ms):** INF

**Lifespan**

**Duration (ms):** INF
• **Partition**: The user can select different partitions to differentiate groups of publishers and subscribers. The user can select *no partition* or any combination between four partitions (A, B, C and D). Additionally the user can select the * partition, that will be matched against all other partitions (not only the four given here) (see Fast DDS Partitions Documentation).

  **Note:** Using the wildcard (*) partition is not the same as not using any partition. A publisher that uses the wildcard partition will not be matched with a subscriber that do not defines any partitions.

• **Reliable**: The user can select to disable the Reliable check-box to use a *Best-Effort* publisher (see Fast DDS ReliabilityQosPolicy Documentation).

• **History and Durability**: The publishers’s History is set to *KEEP_LAST*. The user can select the number of samples that the publisher is going to save and whether this History is going to be *VOLATILE* or *TRANSIENT_LOCAL*. The latter will send that last stored values to subscribers joining after the publisher has been created (see Fast DDS DurabilityQosPolicy Documentation).

• **Liveliness**: The user can select the Liveliness QoS of the publisher from three different values: *AUTOMATIC*, *MANUAL_BY_PARTICIPANT* and *MANUAL_BY_TOPIC*. The Lease Duration value and Announcement Period can also be configured. The latter only applies if Liveliness is set to *AUTOMATIC* or *MANUAL_BY_PARTICIPANT* (see Fast DDS LivelinessQosPolicy Documentation).

• **Ownership**: The Ownership QoS determines whether the key (color) of a Topic (Shape) is owned by a single publisher. If the selected ownership is *EXCLUSIVE* the publisher will use the Ownership strength value as the strength of its publication. Only the publisher with the highest strength can publish in the same Topic with the same Key (see Fast DDS OwnershipQosPolicy Documentation).

• **Deadline**: The Deadline QoS determines the maximum expected amount of time between samples. When the deadline is missed the application will be notified and a message will be printed on the console (see Fast DDS DeadlineQosPolicy Documentation).

• **Lifespan**: The Lifespan QoS determines the duration while the sample is still valid. When a sample’s lifespan expires, it will be removed from publisher and subscriber histories. (see Fast DDS LifespanQosPolicy Documentation).

  **Note:** Using Deadline and Lifespan QoS will not have any visual effect.

---

**Subscribing to a Topic**

When the Subscriber button is pressed, a new window appear to allow the user to define the Shape (topic) and Quality of Service for its subscription. The following image shows an example of the Subscribe menu.

This menu is highly similar to the Publication menu but the user cannot change the color and size of the Shape, and it has additional elements:

• **Liveliness**: This QoS policy is applied in the same way as in the publisher except for the Announcement Period, which does not apply for the Subscriber (see Fast DDS LivelinessQosPolicy Documentation).

• **Time Based Filter**: This value can be used by the user to specify the minimum amount of time (in milliseconds) that the subscriber wants between updates (see Fast DDS TimeBasedFilterQosPolicy Documentation).

• **Content Based Filter**: This filter draws a rectangle in the instances window. Only the shapes that are included in this rectangle are accepted while the rest of them are ignored. The user can dynamically resize and move this content filter.
### QoS Settings

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>History</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Durability</td>
<td>VOLATILE</td>
<td></td>
</tr>
<tr>
<td>Reliable</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Liveliness

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kind</td>
<td>AUTOMATIC</td>
<td></td>
</tr>
<tr>
<td>Lease Duration</td>
<td>INF</td>
<td></td>
</tr>
</tbody>
</table>

### Ownership

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kind</td>
<td>SHARED_OWNERSHIP</td>
<td></td>
</tr>
</tbody>
</table>

### Deadline

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duration</td>
<td>INF</td>
<td></td>
</tr>
</tbody>
</table>

### Lifespan

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duration</td>
<td>INF</td>
<td></td>
</tr>
</tbody>
</table>

### Filters

- Time Based (ms): 0
- Content Based: No
Participant configuration

The *ROS 2 Shapes Demo* application allows the user to define Participant policies. To see the Options window, please go to Options->Participant Configuration in the main bar. The following image shows the Options Menu.

- **Transport Protocol**: You can select between UDP protocol, TCP LAN Server, TCP WAN Server or TCP Client, and Shared Memory transport. In case no transport has been activated, *Fast DDS* default transports will be used (UDP + SHM) (see *Fast DDS Transports Documentation*).
- **Same host delivery**: *Fast DDS* has some features that allow Participants running in the same host or process to share resources in order to improve the communication:
  - **Intraprocess**: Allow using Intraprocess delivery when both Endpoints are running in the same process (see *Fast DDS Intraprocess Documentation*).
  - **Data Sharing**: Allow using Data Sharing delivery when both Endpoints are running in the same host (see *Fast DDS Data Sharing Documentation*).
- **Domain**: The user can select different Domain IDs. Shapes Demo instances using different Domain IDs will not communicate. To modify the Domain ID the user needs to stop the participant (thus removing all existing publishers and subscribers) and start a new one with the new Domain ID (see *Fast DDS Domain Documentation*).
- **Statistics**: The user can activate *Fast DDS Statistics module* so different instrumentation data could be collected and analyzed by the *Fast DDS Statistics Backend*, or be represented by *Fast DDS Monitor*. This module requires to have compiled *Fast DDS* with Statistics Module ON (see *Fast DDS Statistics Module Documentation*).
- **ROS**: By activating “Use ROS 2 Topics”, *ROS 2 Shapes Demo* automatically renames the topic name and the topic data type according to the mangling used in ROS 2.

In case that the Participant is already running, it should be stopped in order to change its configuration. This will drop every endpoint already created.
Endpoints and Output tabs

A table including all created endpoints is also provided. An example of this legend is shown in the following figure.

<table>
<thead>
<tr>
<th>Endpoints</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Topic</strong></td>
<td><strong>Color</strong></td>
</tr>
<tr>
<td>1 Square</td>
<td>RED</td>
</tr>
<tr>
<td>2 Triangle</td>
<td>CYAN</td>
</tr>
<tr>
<td>3 Circle</td>
<td>GRAY</td>
</tr>
<tr>
<td>4 Circle</td>
<td>---</td>
</tr>
</tbody>
</table>

This table can be used to remove endpoints. Two methods are provided:

- Right click in an endpoint: An option to remove the endpoint is shown.
- Pressing the delete button when the endpoint is selected.

The output tab shows the output log messages. An example of the output tab is shown in the figure below.

Next Steps

Visit eProsima Shapes Demo Documentation for more information on how to use this application.

4.6.5 ROS 2 Router

ROSS 2 Router (a.k.a. DDS Router) is an end-user software application shipped with Vulcanexus Cloud that enables the connection of distributed ROS 2 networks. That is, ROS 2 nodes deployed in one geographic location and using a dedicated local network will be able to communicate with other nodes deployed in different geographic areas on their own dedicated local networks as if they were all on the same network through the use of ROS 2 Router. This is achieved by deploying a ROS 2 Router on an edge device of each local network so that the ROS 2 Router routes DDS traffic from one network to the other through WAN communication.

Furthermore, ROS 2 Router is a software designed for various forms of distributed networks, such as mesh networks in which nodes are deployed in different private local networks that are auto-discovered without any centralized network.
node, or cloud-based networks where there is a data processing cloud and multiple geographically distributed edge devices.

Overview

Following are some of the key features of ROS 2 Router:

1. **WAN communication over TCP**: it supports WAN over TCP communication to establish communications over the Internet.

2. **Distributed nature**: the user may deploy intermediate ROS 2 Router nodes to discover new entities that enter and leave the network dynamically.

3. **Efficient data routing**: ROS 2 Router avoids data introspection achieving a zero-copy system in data forwarding.

4. **Easy deployment**: it is based on an easily configurable modular system for users with no knowledge of computer networks.

5. **Topic allowlisting**: it is possible to configure a ROS 2 Router to forward just the user data belonging to a topic specified by the user.

6. **Dynamic topic discovery**: the user does not need to fully specify over which topics to communicate (i.e. provide concrete topic names and types). The discovery of topics matching the allowlisting rules automatically triggers the creation of all entities required for communication.

7. **Quality of Service preservation**: ROS 2 Router uses the QoS set in the user’s ROS 2 network and keeps the reliability and durability of the communication for each topic. These QoS are also manually configurable.
Next Steps

Visit eProsima DDS Router Documentation for more information on how to configure and deploy a ROS 2 Router instance. Also feel free to review Vulcanexus Cloud Tutorials to discover the possibilities that this tool has to offer in different scenarios.

4.6.6 ROS 2 Record & Replay

ROS 2 Record & Replay (a.k.a. DDS Record & Replay) is an end-user software application that efficiently saves ROS 2 messages published into a DDS environment in a MCAP format database. Thus, the exact playback of the recorded network events is possible as the data is linked to the timestamp at which the original data was published.

ROS 2 Record & Replay is easily configurable and installed with a default setup, so that DDS topics, data types and entities are automatically discovered without the need to specify the types of data recorded. This is because the recording tool leverages the DynamicTypes functionality of eProsima Fast DDS, the C++ implementation of the DDS (Data Distribution Service) Specification defined by the Object Management Group (OMG).

Overview

ROS 2 Record & Replay includes the following tools:

• **ROS 2 Recorder tool.** The main functionality of this tool is to save the data in a MCAP database. The database contains the records of the publication timestamp of the data, the serialized data, and the definition of the data serialization type and format. The output MCAP file can be read with any user tool compatible with MCAP file reading since it contains all the necessary information for reading and reproducing the data.

• **ROS 2 Replay tool.** This application allows to reproduce DDS traffic recorded with a ROS 2 Recorder. A user can specify which messages to replay by either setting a time range (begin/end times) out of which messages will be discarded, or directly by blocking/whitelisting a set of topics of interest. It is also possible to choose a different playback rate, as well as to use topic QoS different to the ones recorded.
Next Steps

Visit eProsima DDS Record & Replay Documentation for more information on how to configure and deploy a ROS 2 Record & Replay instance. Also feel free to review Vulcanexus Tools Tutorials to discover the possibilities that this tool has to offer in different scenarios.

4.6.7 Fast DDS Spy

Fast DDS Spy is a CLI interactive tool that allows the introspection of a ROS 2 network in human readable format. It is possible to query the network about the Participants connected, their endpoints (Publishers and Subscriptions) and the topics they communicate in. It is also possible to see the user data sent through network topics in a schematic format in run time.

Overview

Fast DDS Spy is a tool that captures DDS packages in the network and maintain a local database that is accessible from a interactive CLI. Fast DDS Spy responds to user commands introduced by text and prints to stdout the information requested. This tool has several commands to interact with it, allowing the user to get information about the status of the network. It supports listing of existing topics, Participants, Publishers, and Subscriptions, and printing data in real time in a human readable format.

Fast DDS Spy is easily configurable and installed with a default setup, so that DDS topics, data types and entities are automatically discovered without the need to specify the types of data. This is because this tool leverages the DynamicTypes functionality of eProsima Fast DDS, the C++ implementation of the DDS (Data Distribution Service) Specification defined by the Object Management Group (OMG).
Next Steps

Visit eProsima Fast DDS Spy Documentation for more information on how to configure and use Fast DDS Spy. Also feel free to review Vulcanexus Tools Tutorials to discover the possibilities that this tool has to offer in different scenarios.

4.6.8 Webots

Webots is an open-source three-dimensional mobile robot simulator. It was originally developed as a research tool to investigate various control algorithms in mobile robotics. Since December 2018, Webots is released as an open source software under the Apache 2.0 license. Cyberbotics Ltd. maintains Webots as its main product continuously since 1998. Please, refer to the original Webots User Guide Documentation for more details on how to setup and get started with Webots.

Webots offers a rapid prototyping environment, that allows the user to create 3D virtual worlds with physics properties such as mass, joints, friction coefficients, etc. The user can add simple passive objects or active objects called mobile robots. These robots can have different locomotion schemes (wheeled robots, legged robots, or flying robots). Moreover, they may be equipped with a number of sensor and actuator devices, such as distance sensors, drive wheels, cameras, motors, touch sensors, emitters, receivers, etc. Finally, the user can program each robot individually to exhibit the desired behavior. Webots contains a large number of robot models and controller program examples to help users get started. Webots also contains a number of interfaces to real mobile robots, so that once your simulated robot behaves as expected, you can transfer its control program to a real robot like e-puck, DARwIn-OP, Nao, etc. Adding new interfaces is possible through the related system.

What can I do with Webots?

Webots is well suited for research and educational projects related to mobile robotics. Many mobile robotics projects have relied on Webots for years in the following areas:

- Mobile robot prototyping (academic research, the automotive industry, aeronautics, the vacuum cleaner industry, the toy industry, hobbyists, etc.).
- Robot locomotion research (legged, humanoids, quadrupeds robots, etc.).
- Multi-agent research (swarm intelligence, collaborative mobile robots groups, etc.).
- Adaptive behavior research (genetic algorithm, neural networks, AI, etc.).
- Teaching robotics (robotics lectures, C/C++/Java/Python programming lectures, etc.).
- Robot contests (e.g. Robotstadium or Rat’s Life).
What Do I Need to Know to Use Webots?

You will need a minimal amount of technical knowledge to develop your own simulations:

- A basic knowledge of the C, C++, Java, Python or MATLAB programming language is necessary to program your own robot controllers. However, even if you don’t know these languages, you can still program the e-puck and Hemisson robots using a simple graphical programming language called BotStudio.

- If you don’t want to use existing robot models provided within Webots and would like to create your own robot models, or add special objects in the simulated environments, you will need a basic knowledge of 3D computer graphics and VRML97 description language. That will allow you to create 3D models in Webots or import them from 3D modeling software.

Webots Simulation

A Webots simulation is composed of following items:

1. A Webots world file (.wbt) that defines one or several robots and their environment. The .wbt file does sometimes depend on external PROTO files (.proto) and textures.

2. One or several controller programs for the above robots (in C/C++/Java/Python/MATLAB).

3. An optional physics plugin that can be used to modify Webots regular physics behavior (in C/C++).
Next Steps

Visit Webots Documentation for more information.

Note: Copyright © 2022 Cyberbotics Ltd.

- **ROS 2 QoS Profiles Manager**: tool suite to generate XML configuration files for Vulcanexus middleware Fast DDS.
- **ROS 2 Monitor**: graphical desktop application to monitor ROS 2 communications.
- **Fast DDS Statistics Backend**: ROS 2 Monitor’s backend. Can be leveraged with other monitoring frontend applications (e.g. Prometheus).
- **ROS 2 Shapes Demo**: first demo application to understand the most used ROS 2 Quality of Service (QoS) and test DDS and ROS 2 communication.
- **ROS 2 Router**: end-user software application enabling the connection of distributed ROS 2 environments.
- **ROS 2 Record & Replay**: end-user software applications that efficiently save messages published in a ROS 2 environment and playback these messages in the order in which they were recorded.
- **Fast DDS Spy**: CLI interactive tool that allows the introspection of a ROS 2 environment in a human readable format.
- **Webots**: open-source three-dimensional mobile robot simulator.

Note: This documentation provides several tutorials showcasing the capabilities and advantages of Vulcanexus.

4.7 Vulcanexus Metapackages

Vulcanexus provides several binary distributions that install specific packages depending on the developer’s needs. Vulcanexus Documentation table summarizes the different Vulcanexus package distributions along with the most important included packages.

4.7.1 Vulcanexus Core

Vulcanexus Core is the most fundamental and minimal Vulcanexus installation. It is composed by the ROS 2 Base libraries but with the latest eProsima Fast DDS and Fast CDR releases and the ROS MiddleWare (RMW) needed to benefit from the latest features included in those releases. No additional developer tools are included with this basic installation.

Note: Vulcanexus Core Tutorials showcase the specific features that Vulcanexus implements but are not yet included within the ROS 2 ecosystem.

Note: The binary package is called following the next convention vulcanexus-<version_name>-core being the version_name the corresponding adjective associated with the specific release. For example, for Vulcanexus v3, version_name is iron. Consequently, in order to install this package run the following command after setting up the repository as explained in the installation guide.
4.7.2 Vulcanexus Tools

Vulcanexus Tools is focused on helping developers with the introspection and debugging of their system. Thus, the ROS 2 Monitor and the ROS 2 Statistics Backend are included in order to monitor and diagnose the ROS 2 communication network. Also, ROS 2 Shapes Demo is provided as a simple demo to help testing that the communication is successfully established in the system.

Note: Vulcanexus Tools Tutorials introduce these developer tools with some simple use case in order to show its potential.

Note: The binary package is called following the next convention vulcanexus-<version_name>-tools being the version_name the corresponding adjective associated with the specific release. For example, for Vulcanexus v3, version_name is iron. Consequently, in order to install this package run the following command after setting up the repository as explained in the installation guide

```
sudo apt install vulcanexus-iron-tools
```

4.7.3 Vulcanexus Micro

Vulcanexus Micro installs micro-ROS toolkit oriented to the development and deployment of embedded ROS 2 applications. Thus, eProsima Micro XRCE-DDS, the protocol that enables a resource constrained, low-powered device to communicate with a DDS network through the micro-ROS Agent, and the Agent itself are installed. Micro-ROS provides also a set of tools for integration within many build systems, toolchains and embedded development frameworks.

Note: More information about Micro-ROS can be found in Vulcanexus Micro Tutorials.

Note: The binary package is called following the next convention vulcanexus-<version_name>-micro being the version_name the corresponding adjective associated with the specific release. For example, for Vulcanexus v3, version_name is iron. Consequently, in order to install this package run the following command after setting up the repository as explained in the installation guide

```
sudo apt install vulcanexus-iron-micro
```
4.7.4 Vulcanexus Cloud

Vulcanexus Cloud is oriented to geographically spaced scenarios, deploying robotic applications through a distributed network. To this end, ROS 2 Router is installed in this Vulcanexus package distribution, allowing ROS 2 communications through different network layers (bridging the local networks through WAN communication).

Note: Vulcanexus Cloud Tutorials showcase the features and capabilities of the Vulcanexus Cloud package distribution.

Note: The binary package is called following the next convention vulcanexus-<version_name>-cloud being the version_name the corresponding adjective associated with the specific release. For example, for Vulcanexus v3, version_name is iron. Consequently, in order to install this package run the following command after setting up the repository as explained in the installation guide

```
sudo apt install vulcanexus-iron-cloud
```

4.7.5 Vulcanexus Simulation

Vulcanexus Simulation installs the open-source robot simulator Webots. Webots allows to simulate real robotic environments providing examples of robot models and controllers

Note: The binary package is called following the next convention vulcanexus-<version_name>-simulation being the version_name the corresponding adjective associated with the specific release. For example, for Vulcanexus v3, version_name is iron. Consequently, in order to install this package run the following command after setting up the repository as explained in the installation guide

```
sudo apt install vulcanexus-iron-simulation
```

4.7.6 Vulcanexus Base

Vulcanexus Base installs the Tools, Micro and Cloud components jointly.

Note: The binary package is called following the next convention vulcanexus-<version_name>-base being the version_name the corresponding adjective associated with the specific release. For example, for Vulcanexus v3, version_name is iron. Consequently, in order to install this package run the following command after setting up the repository as explained in the installation guide

```
sudo apt install vulcanexus-iron-base
```
4.7.7 Vulcanexus Desktop

Finally, **Vulcanexus Desktop** package distribution installs all of the above packages besides **ROS 2 Desktop package distribution**. **ROS 2 Desktop** provides additional visualization tools, examples, demos, and tutorials. This is the most complete **Vulcanexus** installation and it is intended for developers that want a better understanding of the **ROS 2** ecosystem.

**Note:** The binary package is called following the next convention `vulcanexus-<version_name>-desktop` being the `version_name` the corresponding adjective associated with the specific release. For example, for Vulcanexus v3, the `version_name` is iron. Consequently, in order to install this package run the following command after setting up the repository as explained in the installation guide

```
sudo apt install vulcanexus-iron-desktop
```

4.8 ROS 2 Documentation

4.8.1 Installation

Options for installing ROS 2 Iron Irwini:

**Ubuntu (Debian packages)**

<table>
<thead>
<tr>
<th>Table of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Resources</td>
</tr>
<tr>
<td>• System setup</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>• Install ROS 2</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>• Setup environment</td>
</tr>
<tr>
<td>• Try some examples</td>
</tr>
<tr>
<td>• Next steps after installing</td>
</tr>
<tr>
<td>• Use the ROS 1 bridge (optional)</td>
</tr>
<tr>
<td>• Troubleshoot</td>
</tr>
<tr>
<td>• Uninstall</td>
</tr>
</tbody>
</table>

Debian packages for ROS 2 Iron Irwini are currently available for Ubuntu Jammy. The Rolling Ridley distribution will change target platforms from time to time as new platforms are selected for development. The target platforms are defined in REP 2000 Most people will want to use a stable ROS distribution.
Resources

- Status Page:
  - ROS 2 Iron (Ubuntu Jammy): amd64, arm64
- Jenkins Instance
- Repositories

System setup

Set locale

Make sure you have a locale which supports UTF-8. If you are in a minimal environment (such as a docker container), the locale may be something minimal like POSIX. We test with the following settings. However, it should be fine if you’re using a different UTF-8 supported locale.

```
locale  # check for UTF-8
sudo apt update && sudo apt install locales
sudo locale-gen en_US en_US.UTF-8
sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
export LANG=en_US.UTF-8
locale  # verify settings
```

Enable required repositories

You will need to add the ROS 2 apt repository to your system. First ensure that the Ubuntu Universe repository is enabled.

```
sudo apt install software-properties-common
sudo add-apt-repository universe
```

Now add the ROS 2 GPG key with apt.

```
sudo apt update && sudo apt install curl -y
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o /usr/share/keyrings/ros-archive-keyring.gpg
```

Then add the repository to your sources list.

```
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-archive-keyring.gpg] http://packages.ros.org/ros2/ubuntu $(. /etc/os-release && echo $UBUNTU_CODENAME) main" | sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null
```
Install development tools (optional)

If you are going to build ROS packages or otherwise do development, you can also install the development tools:

```
sudo apt update && sudo apt install ros-dev-tools
```

Install ROS 2

Update your apt repository caches after setting up the repositories.

```
sudo apt update
```

ROS 2 packages are built on frequently updated Ubuntu systems. It is always recommended that you ensure your system is up to date before installing new packages.

```
sudo apt upgrade
```

**Warning:** Due to early updates in Ubuntu 22.04 it is important that systemd and udev-related packages are updated before installing ROS 2. The installation of ROS 2’s dependencies on a freshly installed system without upgrading can trigger the removal of critical system packages.

Please refer to ros2/ros2#1272 and Launchpad #1974196 for more information.

Desktop Install (Recommended): ROS, RViz, demos, tutorials.

```
sudo apt install ros-iron-desktop
```

ROS-Base Install (Bare Bones): Communication libraries, message packages, command line tools. No GUI tools.

```
sudo apt install ros-iron-ros-base
```

Install additional RMW implementations (optional)

The default middleware that ROS 2 uses is Fast DDS, but the middleware (RMW) can be replaced at runtime. See the guide on how to work with multiple RMWs.

Setup environment

Set up your environment by sourcing the following file.

```
# Replace ".bash" with your shell if you're not using bash
# Possible values are: setup.bash, setup.sh, setup.zsh
source /opt/ros/iron/setup.bash
```
Try some examples

If you installed `ros-iron-desktop` above you can try some examples.
In one terminal, source the setup file and then run a C++ talker:

```
source /opt/ros/iron/setup.bash
ros2 run demo_nodes_cpp talker
```

In another terminal source the setup file and then run a Python listener:

```
source /opt/ros/iron/setup.bash
ros2 run demo_nodes_py listener
```

You should see the talker saying that it’s Publishing messages and the listener saying I heard those messages. This verifies both the C++ and Python APIs are working properly. Hooray!

Next steps after installing

Continue with the tutorials and demos to configure your environment, create your own workspace and packages, and learn ROS 2 core concepts.

Use the ROS 1 bridge (optional)

The ROS 1 bridge can connect topics from ROS 1 to ROS 2 and vice-versa. See the dedicated document on how to build and use the ROS 1 bridge.

Troubleshoot

Troubleshooting techniques can be found here.

Uninstall

If you need to uninstall ROS 2 or switch to a source-based install once you have already installed from binaries, run the following command:

```
sudo apt remove ~nros-iron-* && sudo apt autoremove
```

You may also want to remove the repository:

```
sudo rm /etc/apt/sources.list.d/ros2.list
sudo apt update
sudo apt autoremove
# Consider upgrading for packages previously shadowed.
sudo apt upgrade
```
Windows (binary)

This page explains how to install ROS 2 on Windows from a pre-built binary package.

Note: The pre-built binary does not include all ROS 2 packages. All packages in the ROS base variant are included, and only a subset of packages in the ROS desktop variant are included. The exact list of packages are described by the repositories listed in this ros2.repos file.
System requirements

Only Windows 10 is supported.

System setup

Install Chocolatey

Chocolatey is a package manager for Windows, install it by following their installation instructions:
https://chocolatey.org/install
You’ll use Chocolatey to install some other developer tools.

Install Python

Open a Command Prompt and type the following to install Python via Chocolatey:

```
choco install -y python --version 3.8.3
```

Install Visual C++ Redistributables

Open a Command Prompt and type the following to install them via Chocolatey:

```
choco install -y vcredist2013 vcredist140
```

Install OpenSSL

Download the Win64 OpenSSL v1.1.1n OpenSSL installer from this page. Scroll to the bottom of the page and download Win64 OpenSSL v1.1.1n. Don’t download the Win32 or Light versions, or the v3.X.Y installers.

Run the installer with default parameters, as the following commands assume you used the default installation directory.

This command sets an environment variable that persists over sessions:

```
setx /m OPENSSL_CONF "C:\Program Files\OpenSSL-Win64\bin\openssl.cfg"
```

You will need to append the OpenSSL-Win64 bin folder to your PATH. You can do this by clicking the Windows icon, typing “Environment Variables”, then clicking on “Edit the system environment variables”. In the resulting dialog, click “Environment Variables”, then click “Path” on the bottom pane, finally click “Edit” and add the path below.

- C:\Program Files\OpenSSL-Win64\bin\
Install Visual Studio


If you already have a paid version of Visual Studio 2019 (Professional, Enterprise), skip this step.

Microsoft provides a free of charge version of Visual Studio 2019, named Community, which can be used to build applications that use ROS 2. You can download the installer directly through this link.

Make sure that the Visual C++ features are installed.

An easy way to make sure they’re installed is to select the Desktop development with C++ workflow during the install.

Make sure that no C++ CMake tools are installed by unselecting them in the list of components to be installed.

Install OpenCV

Some of the examples require OpenCV to be installed.

You can download a precompiled version of OpenCV 3.4.6 from https://github.com/ros2/ros2/releases/download/opencv-archives/opencv-3.4.6-vc16.VS2019.zip .

Assuming you unpacked it to C:\opencv, type the following on a Command Prompt (requires Admin privileges):

```
setx /m OpenCV_DIR C:\opencv
```

Since you are using a precompiled ROS version, we have to tell it where to find the OpenCV libraries. You have to extend the PATH variable to C:\opencv\x64\vc16\bin.
Install dependencies

There are a few dependencies not available in the Chocolatey package database. In order to ease the manual installation process, we provide the necessary Chocolatey packages.

As some chocolatey packages rely on it, we start by installing CMake

```
choco install -y cmake
```

You will need to append the CMake bin folder `C:\Program Files\CMake\bin` to your PATH.

Please download these packages from this GitHub repository.

- asio.1.12.1.nupkg
- bullet.3.17.nupkg
- cunit.2.1.3.nupkg
- eigen-3.3.4.nupkg
- tinyxml-usestl.2.6.2.nupkg
- tinyxml2.6.0.0.nupkg

Once these packages are downloaded, open an administrative shell and execute the following command:

```
choco install -y -s <PATH\TO\DOWNLOADS> asio cunit eigen tinyxml-usestl tinyxml2 bullet
```

Please replace `<PATH\TO\DOWNLOADS>` with the folder you downloaded the packages to.

First upgrade pip and setuptools:

```
python -m pip install -U pip setuptools==59.6.0
```

Now install some additional python dependencies:

```
python -m pip install -U catkin_pkg cryptography empy importlib-metadata jsonschema lark==1.1.1 lxml matplotlib netifaces numpy opencv-python PyQt5 pillow psutil pycairo pydot pyparsing==2.4.7 pyyaml rosdistro
```

Install miscellaneous prerequisites

Next install xmllint:

- Download the 64 bit binary archives of **libxml2** (and its dependencies **iconv** and **zlib**) from `https://www.zlatkovic.com/projects/libxml/`
- Unpack all archives into e.g. C:\xmllint
- Add C:\xmllint\bin to the PATH.
Install Qt5

Download the 5.12.X offline installer from Qt’s website. Run the installer. Make sure to select the MSVC 2017 64-bit component under the Qt -> Qt 5.12.12 tree.

Finally, in an administrator cmd.exe window set these environment variables. The commands below assume you installed it to the default location of C:\Qt.

```
setx /m Qt5_DIR C:\Qt\Qt5.12.12\5.12.12\msvc2017_64
setx /m QT_QPA_PLATFORM_PLUGIN_PATH C:\Qt\Qt5.12.12\5.12.12\msvc2017_64\plugins\platforms
```

Note: This path might change based on the installed MSVC version, the directory Qt was installed to, and the version of Qt installed.

RQt dependencies

To run rqt_graph you need to download and install Graphviz. The installer will ask if to add graphviz to PATH, choose to either add it to the current user or all users.

Install ROS 2

- Download the latest package for Windows, e.g., ros2-package-windows-AMD64.zip.

Note: There may be more than one binary download option which might cause the file name to differ.

Note: To install debug libraries for ROS 2, see Extra Stuff for Debug. Then continue on with downloading ros2-package-windows-debug-AMD64.zip.

- Unpack the zip file somewhere (we’ll assume C:\dev\ros2_iron).

Install additional RMW implementations (optional)

The default middleware that ROS 2 uses is Fast DDS, but the middleware (RMW) can be replaced at runtime. See the guide on how to work with multiple RMWs.

Setup environment

Start a command shell and source the ROS 2 setup file to set up the workspace:

```
call C:\dev\ros2_iron\local_setup.bat
```

It is normal that the previous command, if nothing else went wrong, outputs “The system cannot find the path specified.” exactly once.
Try some examples

In a command shell, set up the ROS 2 environment as described above and then run a C++ talker:

```
ros2 run demo_nodes_cpp talker
```

Start another command shell and run a Python listener:

```
ros2 run demo_nodes_py listener
```

You should see the talker saying that it’s Publishing messages and the listener saying I heard those messages. This verifies both the C++ and Python APIs are working properly. Hooray!

Next steps after installing

Continue with the tutorials and demos to configure your environment, create your own workspace and packages, and learn ROS 2 core concepts.

Troubleshoot

Troubleshooting techniques can be found here.

Uninstall

1. If you installed your workspace with colcon as instructed above, “uninstalling” could be just a matter of opening a new terminal and not sourcing the workspace’s setup file. This way, your environment will behave as though there is no Iron install on your system.

2. If you’re also trying to free up space, you can delete the entire workspace directory with:

```
rmdir /s /q \ros2_iron
```

Extra Stuff for Debug

To download the ROS 2 debug libraries you’ll need to download ros2-iron-*-windows-debug-AMD64.zip. Please note that debug libraries require some more additional configuration/setup to work as given below.

Python installation may require modification to enable debugging symbols and debug binaries:

- Search in windows Search Bar and open Apps and Features.
- Search for the installed Python version.
- Click Modify.
• Click Next to go to **Advanced Options**.

• Make sure **Download debugging symbols** and **Download debug binaries** are checked.
Click Install.

(Alternative) ROS 2 Build Installation from aka.ms/ros

https://aka.ms/ros project hosts ROS 2 builds against the release snapshots. This section explains how to install ROS 2 from this channel.

Install ROS 2 builds

In an administrative command prompt, run the following commands.

```
mkdir c:\opt\chocolatey
set PYTHONNOUSERSITE=1
set ChocolateyInstall=c:\opt\chocolatey
choco source add -n=ros-win -s="https://aka.ms/ros/public" --priority=1
choco upgrade ros-foxy-desktop -y --execution-timeout=0
```

Setup environment

Start an administrative command prompt and source the ROS 2 setup file to set up the workspace:

```
call C:\opt\ros\foxy\x64\local_setup.bat
```

Stay up-to-date

To keep up-to-date with the latest builds, run:

```
set ChocolateyInstall=c:\opt\chocolatey
choco upgrade all -y --execution-timeout=0
```

Uninstall

If you want to completely remove the environment downloaded above, run this command:

```
rmdir /s /q C:\opt\`
```

RHEL (RPM packages)

Table of Contents

- Resources
- System setup
  - Set locale
  - Enable required repositories
RPM packages for ROS 2 Iron Irwini are currently available for RHEL 9. The Rolling Ridley distribution will change target platforms from time to time as new platforms are selected for development. The target platforms are defined in REP 2000 Most people will want to use a stable ROS distribution.

Resources

- Status Page:
  - ROS 2 Iron (RHEL 9): amd64
- Jenkins Instance
- Repositories

System setup

Set locale

Make sure you have a locale which supports UTF-8. If you are in a minimal environment (such as a docker container), the locale may be something minimal like C. We test with the following settings. However, it should be fine if you’re using a different UTF-8 supported locale.

```
locale  # check for UTF-8
sudo dnf install langpacks-en glibc-langpack-en
export LANG=en_US.UTF-8
locale  # verify settings
```

Enable required repositories

You will need to enable the EPEL repositories and the PowerTools repository:

```
sudo dnf install 'dnf-command(config-manager)' epel-release -y
sudo dnf config-manager --set-enabled crb
```
Note: This step may be slightly different depending on the distribution you are using. Check the EPEL documentation: https://docs.fedoraproject.org/en-US/epel/#_quickstart

Next, download the ROS 2 .repo file:

```bash
sudo dnf install curl
sudo curl --output /etc/yum.repos.d/ros2.repo http://packages.ros.org/ros2/rhel/ros2.repo
```

Then, update your metadata cache. DNF may prompt you to verify the GPG key, which should match the location https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc.

```bash
sudo dnf makecache
```

### Install development tools (optional)

If you are going to build ROS packages or otherwise do development, you can also install the development tools:

```bash
sudo dnf install -y \
cmake \
gcc-c++ \
git \
make \
patch \
python3-colcon-common-extensions \
python3-flake8-builtin \
python3-flake8-comprehensions \
python3-flake8-docstrings \
python3-flake8-import-order \
python3-flake8-quotes \
python3-mypy \
python3-pip \
python3-pydocstyle \
python3-pytest \
python3-pytest-repeat \
python3-pytest-rerunfailures \
python3-rosdep \
python3-setuptools \
python3-vcstool \
wget
```

# install some pip packages needed for testing and # not available as RPMs

```bash
python3 -m pip install -U --user \
flake8-blind-except==0.1.1 \
flake8-class-newline \
flake8-deprecated
```
Install ROS 2

ROS 2 packages are built on frequently updated RHEL systems. It is always recommended that you ensure your system is up to date before installing new packages.

```
sudo dnf update
```

Desktop Install (Recommended): ROS, RViz, demos, tutorials.

```
sudo dnf install ros-iron-desktop
```

ROS-Base Install (Bare Bones): Communication libraries, message packages, command line tools. No GUI tools.

```
sudo dnf install ros-iron-ros-base
```

Install additional RMW implementations (optional)

The default middleware that ROS 2 uses is Fast DDS, but the middleware (RMW) can be replaced at runtime. See the guide on how to work with multiple RMWs.

Setup environment

Set up your environment by sourcing the following file.

```
# Replace ".bash" with your shell if you're not using bash
# Possible values are: setup.bash, setup.sh, setup.zsh
source /opt/ros/iron/setup.bash
```

Try some examples

If you installed ros-iron-desktop above you can try some examples.

In one terminal, source the setup file and then run a C++ talker:

```
source /opt/ros/iron/setup.bash
ros2 run demo_nodes_cpp talker
```

In another terminal source the setup file and then run a Python listener:

```
source /opt/ros/iron/setup.bash
ros2 run demo_nodes_py listener
```

You should see the talker saying that it’s Publishing messages and the listener saying I heard those messages. This verifies both the C++ and Python APIs are working properly. Hooray!
Next steps after installing

Continue with the tutorials and demos to configure your environment, create your own workspace and packages, and learn ROS 2 core concepts.

Troubleshoot

Troubleshooting techniques can be found here.

Uninstall

If you need to uninstall ROS 2 or switch to a source-based install once you have already installed from binaries, run the following command:

```
sudo dnf remove ros-iron-*
```

Alternatives

A list of alternative ways to install ROS 2 – whether it’s by building from source or installing a binary.

Ubuntu (source)

Table of Contents

- System requirements
- System setup
  - Set locale
  - Enable required repositories
  - Install development tools
- Build ROS 2
  - Get ROS 2 code
  - Install dependencies using rosdep
  - Install additional RMW implementations (optional)
  - Build the code in the workspace
- Setup environment
- Try some examples
- Next steps
- Use the ROS 1 bridge (optional)
- Alternate compilers
  - Clang
System requirements

The current Debian-based target platforms for Iron Irwini are:

- Tier 1: Ubuntu Linux - Jammy (22.04) 64-bit
- Tier 3: Debian Linux - Bullseye (11) 64-bit

As defined in REP 2000.

System setup

Set locale

Make sure you have a locale which supports UTF-8. If you are in a minimal environment (such as a docker container), the locale may be something minimal like POSIX. We test with the following settings. However, it should be fine if you’re using a different UTF-8 supported locale.

```
locale   # check for UTF-8
sudo apt update && sudo apt install locales
sudo locale-gen en_US en_US.UTF-8
sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
export LANG=en_US.UTF-8
locale   # verify settings
```

Enable required repositories

You will need to add the ROS 2 apt repository to your system.

First ensure that the Ubuntu Universe repository is enabled.

```
sudo apt install software-properties-common
sudo add-apt-repository universe
```

Now add the ROS 2 GPG key with apt.

```
sudo apt update && sudo apt install curl -y
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o /usr/share/keyrings/ros-archive-keyring.gpg
```

Then add the repository to your sources list.

```
echo "deb [arch="$(dpkg --print-architecture)" signed-by=/usr/share/keyrings/ros-archive-keyring.gpg] http://packages.ros.org/ros2/ubuntu $(. /etc/os-release && echo $UBUNTU_CODENAME) main" | sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null
```
Install development tools

```
sudo apt update && sudo apt install -y
  python3-flake8-docstrings
  python3-pip
  python3-pytest-cov
  python3-flake8-blind-except
  python3-flake8-builtins
  python3-flake8-class-newline
  python3-flake8-comprehensions
  python3-flake8-deprecated
  python3-flake8-import-order
  python3-flake8-quotes
  python3-pytest-repeat
  python3-pytest-rerunfailures
  ros-dev-tools
```

Build ROS 2

Get ROS 2 code

Create a workspace and clone all repos:

```
mkdir -p ~/ros2_iron/src
cd ~/ros2_iron
vcs import --input https://raw.githubusercontent.com/ros2/ros2/iron/ros2.repos src
```

Install dependencies using rosdep

ROS 2 packages are built on frequently updated Ubuntu systems. It is always recommended that you ensure your system is up to date before installing new packages.

```
sudo apt upgrade
```

```
sudo rosdep init
orosdep update
rosdep install --from-paths src --ignore-src -y --skip-keys "fastcdr rti-connext-dds-6.0...1 urdfdom_headers"
```

**Note:** If you’re using a distribution that is based on Ubuntu (like Linux Mint) but does not identify itself as such, you’ll get an error message like Unsupported OS [mint]. In this case append `--os=ubuntu:jammy` to the above command.
**Install additional RMW implementations (optional)**

The default middleware that ROS 2 uses is **Fast DDS**, but the middleware (RMW) can be replaced at build or runtime. See the guide on how to work with multiple RMWs.

**Build the code in the workspace**

If you have already installed ROS 2 another way (either via Debians or the binary distribution), make sure that you run the below commands in a fresh environment that does not have those other installations sourced. Also ensure that you do not have source /opt/ros/${ROS_DISTRO}/setup.bash in your .bashrc. You can make sure that ROS 2 is not sourced with the command `printenv | grep -i ROS`. The output should be empty.

More info on working with a ROS workspace can be found in this tutorial.

```
    cd ~/ros2_iron/
colcon build --symlink-install
```

**Note:** If you are having trouble compiling all examples and this is preventing you from completing a successful build, you can use the `--packages-skip` colcon flag to ignore the package that is causing problems. For instance, if you don’t want to install the large OpenCV library, you could skip building the packages that depend on it using the command:

```
colcon build --symlink-install --packages-skip image_tools intra_process_demo
```

**Setup environment**

Set up your environment by sourcing the following file.

```
# Replace ".bash" with your shell if you're not using bash
# Possible values are: setup.bash, setup.sh, setup.zsh
. ~/ros2_iron/install/local_setup.bash
```

**Try some examples**

In one terminal, source the setup file and then run a C++ talker:

```
    . ~/ros2_iron/install/local_setup.bash
    ros2 run demo_nodes_cpp talker
```

In another terminal source the setup file and then run a Python listener:

```
    . ~/ros2_iron/install/local_setup.bash
    ros2 run demo_nodes_py listener
```

You should see the talker saying that it’s Publishing messages and the listener saying I heard those messages. This verifies both the C++ and Python APIs are working properly. Hooray!
Next steps

Continue with the tutorials and demos to configure your environment, create your own workspace and packages, and learn ROS 2 core concepts.

Use the ROS 1 bridge (optional)

The ROS 1 bridge can connect topics from ROS 1 to ROS 2 and vice-versa. See the dedicated document on how to build and use the ROS 1 bridge.

Alternate compilers

Using a different compiler besides gcc to compile ROS 2 is easy. If you set the environment variables CC and CXX to executables for a working C and C++ compiler, respectively, and retrigger CMake configuration (by using --force-cmake-config or by deleting the packages you want to be affected), CMake will reconfigure and use the different compiler.

Clang

To configure CMake to detect and use Clang:

```
sudo apt install clang
export CC=clang
export CXX=clang++
colcon build --cmake-force-configure
```

Stay up to date

See Maintain source checkout to periodically refresh your source installation.

Troubleshoot

Troubleshooting techniques can be found here.

Uninstall

1. If you installed your workspace with colcon as instructed above, “uninstalling” could be just a matter of opening a new terminal and not sourcing the workspace’s setup file. This way, your environment will behave as though there is no Iron install on your system.

2. If you’re also trying to free up space, you can delete the entire workspace directory with:

```
rm -rf ~/ros2_iron
```
Ubuntu (binary)

This page explains how to install ROS 2 on Ubuntu Linux from a pre-built binary package.

Note: The pre-built binary does not include all ROS 2 packages. All packages in the ROS base variant are included, and only a subset of packages in the ROS desktop variant are included. The exact list of packages are described by the repositories listed in this ros2.repos file.

There are also Debian packages available.

System requirements

We currently support Ubuntu Linux Jammy (22.04) 64-bit x86 and 64-bit ARM.

System setup

Set locale

Make sure you have a locale which supports UTF-8. If you are in a minimal environment (such as a docker container), the locale may be something minimal like POSIX. We test with the following settings. However, it should be fine if you’re using a different UTF-8 supported locale.
locale  # check for UTF-8

sudo apt update && sudo apt install locales
sudo locale-gen en_US en_US.UTF-8
sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
export LANG=en_US.UTF-8
locale  # verify settings

Enable required repositories

You will need to add the ROS 2 apt repository to your system.
First ensure that the Ubuntu Universe repository is enabled.

sudo apt install software-properties-common
sudo add-apt-repository universe

Now add the ROS 2 GPG key with apt.

sudo apt update && sudo apt install curl -y
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o /usr/share/keyrings/ros-archive-keyring.gpg

Then add the repository to your sources list.

```
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-archive-keyring.gpg] http://packages.ros.org/ros2/ubuntu $(. /etc/os-release && echo $UBUNTU_CODENAME) main" | sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null
```

Install prerequisites

There are a few packages that must be installed in order to get and unpack the binary release.

```
sudo apt install tar bzip2 wget -y
```

Install development tools (optional)

If you are going to build ROS packages or otherwise do development, you can also install the development tools:

```
sudo apt update && sudo apt install ros-dev-tools
```
Install ROS 2

- Go to the releases page
- Download the latest package for Ubuntu; let’s assume that it ends up at `~/Downloads/ros2-package-linux-x86_64.tar.bz2`.
  - Note: there may be more than one binary download option which might cause the file name to differ.
- Unpack it:

```
mkdir -p ~/ros2_iron
cd ~/ros2_iron
tar xf ~/Downloads/ros2-package-linux-x86_64.tar.bz2
```

Install dependencies using rosdep

ROS 2 packages are built on frequently updated Ubuntu systems. It is always recommended that you ensure your system is up to date before installing new packages.

```
sudo apt upgrade
```

```
sudo apt update
dsdo apt install -y python3-rosdep
sudo rosdep init
rosdep update
rosdep install --from-paths ~/ros2_iron/ros2-linux/share --ignore-src -y --skip-keys
  -"cyclonedds fastcdr fastrtps iceoryx_binding_c rmw_connextdds rti-connext-dds-6.0.1␣
  → urdfdom_headers"
```

Note: If you’re using a distribution that is based on Ubuntu (like Linux Mint) but does not identify itself as such, you’ll get an error message like Unsupported OS [mint]. In this case append `--os=ubuntu:jammy` to the above command.

Install additional RMW implementations (optional)

The default middleware that ROS 2 uses is Fast DDS, but the middleware (RMW) can be replaced at runtime. See the guide on how to work with multiple RMWs.

Setup environment

Set up your environment by sourcing the following file.

```
# Replace ".bash" with your shell if you're not using bash
# Possible values are: setup.bash, setup.sh, setup.zsh
./~/ros2_iron/ros2-linux/setup.bash
```
Try some examples

In one terminal, source the setup file and then run a C++ talker:

```
. ~/ros2_iron/ros2-linux/setup.bash
ros2 run demo_nodes_cpp talker
```

In another terminal source the setup file and then run a Python listener:

```
. ~/ros2_iron/ros2-linux/setup.bash
ros2 run demo_nodes_py listener
```

You should see the talker saying that it's Publishing messages and the listener saying I heard those messages. This verifies both the C++ and Python APIs are working properly. Hooray!

Next steps

Continue with the tutorials and demos to configure your environment, create your own workspace and packages, and learn ROS 2 core concepts.

Use the ROS 1 bridge (optional)

The ROS 1 bridge can connect topics from ROS 1 to ROS 2 and vice-versa. See the dedicated document on how to build and use the ROS 1 bridge.

Troubleshoot

Troubleshooting techniques can be found here.

Uninstall

1. If you installed your workspace with colcon as instructed above, “uninstalling” could be just a matter of opening a new terminal and not sourcing the workspace’s setup file. This way, your environment will behave as though there is no Iron install on your system.

2. If you’re also trying to free up space, you can delete the entire workspace directory with:

```
rm -rf ~/ros2_iron
```

Windows (source)

Table of Contents

- System requirements
  - Language support
- System setup
This guide is about how to setup a development environment for ROS 2 on Windows.

**System requirements**

Only Windows 10 is supported.
Language support

Make sure you have a locale which supports UTF-8. For example, for a Chinese-language Windows 10 installation, you may need to install an English language pack.

System setup

Install Chocolatey

Chocolatey is a package manager for Windows, install it by following their installation instructions:
https://chocolatey.org/install
You’ll use Chocolatey to install some other developer tools.

Install Python

Open a Command Prompt and type the following to install Python via Chocolatey:

```
choco install -y python --version 3.8.3
```

Install Visual C++ Redistributables

Open a Command Prompt and type the following to install them via Chocolatey:

```
choco install -y vcredist2013 vcredist140
```

Install OpenSSL

Download the Win64 OpenSSL v1.1.1n OpenSSL installer from this page. Scroll to the bottom of the page and download Win64 OpenSSL v1.1.1t. Don’t download the Win32 or Light versions, or the v3.X.Y installers.
Run the installer with default parameters, as the following commands assume you used the default installation directory.
This command sets an environment variable that persists over sessions:

```
setx /m OPENSSL_CONF "C:\Program Files\OpenSSL-Win64\bin\openssl.cfg"
```
You will need to append the OpenSSL-Win64 bin folder to your PATH. You can do this by clicking the Windows icon, typing “Environment Variables”, then clicking on “Edit the system environment variables”. In the resulting dialog, click “Environment Variables”, then click “Path” on the bottom pane, finally click “Edit” and add the path below.

- C:\Program Files\OpenSSL-Win64\bin\
Install Visual Studio


If you already have a paid version of Visual Studio 2019 (Professional, Enterprise), skip this step.

Microsoft provides a free of charge version of Visual Studio 2019, named Community, which can be used to build applications that use ROS 2. You can download the installer directly through this link.

Make sure that the Visual C++ features are installed.

An easy way to make sure they’re installed is to select the Desktop development with C++ workflow during the install.

Make sure that no C++ CMake tools are installed by unselecting them in the list of components to be installed.

Install OpenCV

Some of the examples require OpenCV to be installed.

You can download a precompiled version of OpenCV 3.4.6 from https://github.com/ros2/ros2/releases/download/opencv-archives/opencv-3.4.6-vc16.VS2019.zip.

Assuming you unpacked it to C:\opencv, type the following on a Command Prompt (requires Admin privileges):

```
setx /m OpenCV_DIR C:\opencv
```

Since you are using a precompiled ROS version, we have to tell it where to find the OpenCV libraries. You have to extend the PATH variable to C:\opencv\x64\vc16\bin.
**Install dependencies**

There are a few dependencies not available in the Chocolatey package database. In order to ease the manual installation process, we provide the necessary Chocolatey packages.

As some chocolatey packages rely on it, we start by installing CMake

```
choco install -y cmake
```

You will need to append the CMake bin folder `C:\Program Files\CMake\bin` to your PATH.

Please download these packages from this GitHub repository.

- `asio.1.12.1.nupkg`
- `bullet.3.17.nupkg`
- `cunit.2.1.3.nupkg`
- `eigen-3.3.4.nupkg`
- `tinyxml-usestl.2.6.2.nupkg`
- `tinyxml2.6.0.0.nupkg`

Once these packages are downloaded, open an administrative shell and execute the following command:

```
choco install -y -s <PATH\TO\DOWNLOADS> asio cunit eigen tinyxml-usestl tinyxml2 bullet
```

Please replace `<PATH\TO\DOWNLOADS>` with the folder you downloaded the packages to.

First upgrade pip and setuptools:

```
python -m pip install -U pip setuptools==59.6.0
```

Now install some additional python dependencies:

```
python -m pip install -U catkin_pkg cryptography empy importlib-metadata jsonschema lark==1.1.1 lxml matplotlib netifaces numpy opencv-python PyQt5 pillow psutil pycairo pydot pyparsing==2.4.7 pyyaml rosdistro
```

**Install miscellaneous prerequisites**

Next install xmllint:

- Download the 64 bit binary archives of `libxml2` (and its dependencies `iconv` and `zlib`) from `https://www.zlatkovic.com/projects/libxml/`
- Unpack all archives into e.g. `C:\xmllint`
- Add `C:\xmllint\bin` to the PATH.
Install Qt5

Download the 5.12.X offline installer from Qt’s website. Run the installer. Make sure to select the MSVC 2017 64-bit component under the Qt -> Qt 5.12.12 tree.

Finally, in an administrator cmd.exe window set these environment variables. The commands below assume you installed it to the default location of C:\Qt.

```bash
setx /m Qt5_DIR C:\Qt\5.12.12\msvc2017_64
setx /m QT_QPA_PLATFORM_PLUGIN_PATH C:\Qt\5.12.12\msvc2017_64\plugins\platforms
```

**Note:** This path might change based on the installed MSVC version, the directory Qt was installed to, and the version of Qt installed.

RQt dependencies

To run rqt_graph you need to download and install Graphviz. The installer will ask if to add graphviz to PATH, choose to either add it to the current user or all users.

Install additional prerequisites from Chocolatey

```bash
choco install -y cppcheck curl git winflexbison3
```

You will need to append the Git cmd folder C:\Program Files\Git\cmd to the PATH (you can do this by clicking the Windows icon, typing “Environment Variables”, then clicking on “Edit the system environment variables”. In the resulting dialog, click “Environment Variables”, the click “Path” on the bottom pane, then click “Edit” and add the path).

Install Python prerequisites

Install additional Python dependencies:

```bash
pip install -U colcon-common-extensions coverage flake8 flake8-blind-except flake8-builtin\nflake8-class-newline flake8-comprehensions flake8-deprecated flake8-de\ndocstrings flake8-import-order flake8-quotes mock mypy==0.931 pep8 pydocstyle pytest py\ntest-mock vcs
tool
```

Build ROS 2

Get ROS 2 code

Now that we have the development tools we can get the ROS 2 source code.

First setup a development folder, for example C:\iron:
Note: It is very important that the chosen path is short, due to the short default Windows path limits (260 characters). To allow longer paths, see https://learn.microsoft.com/en-us/windows/win32/fileio/maximum-file-path-limitation?tabs=registry.

```
md \iron\src
cd \iron
```

Get the ros2.repos file which defines the repositories to clone from:

```
vcs import --input https://raw.githubusercontent.com/ros2/ros2/iron/ros2.repos src
```

**Install additional RMW implementations (optional)**

The default middleware that ROS 2 uses is Fast DDS, but the middleware (RMW) can be replaced at build or runtime. See the guide on how to work with multiple RMWs.

**Build the code in the workspace**

To build ROS 2 you will need a Visual Studio Command Prompt (“x64 Native Tools Command Prompt for VS 2019”) running as Administrator.

To build the \iron folder tree:

```
colcon build --merge-install
```

**Note:** We’re using --merge-install here to avoid a PATH variable that is too long at the end of the build. If you’re adapting these instructions to build a smaller workspace then you might be able to use the default behavior which is isolated install, i.e. where each package is installed to a different folder.

**Note:** If you are doing a debug build use python_d path\to\colcon_executable colcon. See Extra stuff for debug mode for more info on running Python code in debug builds on Windows.

**Setup environment**

Start a command shell and source the ROS 2 setup file to set up the workspace:

```
call C:\iron\install\local_setup.bat
```

This will automatically set up the environment for any DDS vendors that support was built for.

It is normal that the previous command, if nothing else went wrong, outputs “The system cannot find the path specified.” exactly once.
Try some examples

Note that the first time you run any executable you will have to allow access to the network through a Windows Firewall popup.

You can run the tests using this command:

```
colcon test --merge-install
```

**Note:** `--merge-install` should only be used if it was also used in the build step.

Afterwards you can get a summary of the tests using this command:

```
colcon test-result
```

To run the examples, first open a clean new `cmd.exe` and set up the workspace by sourcing the `local_setup.bat` file. Then, run a C++ talker:

```
call install\local_setup.bat
ros2 run demo_nodes_cpp talker
```

In a separate shell you can do the same, but instead run a Python listener:

```
call install\local_setup.bat
ros2 run demo_nodes_py listener
```

You should see the talker saying that it's Publishing messages and the listener saying I heard those messages. This verifies both the C++ and Python APIs are working properly. Hooray!

**Note:** It is not recommended to build in the same `cmd` prompt that you've sourced the `local_setup.bat`.

Next steps

Continue with the tutorials and demos to configure your environment, create your own workspace and packages, and learn ROS 2 core concepts.

Extra stuff for Debug mode

If you want to be able to run all the tests in Debug mode, you'll need to install a few more things:

- To be able to extract the Python source tarball, you can use PeaZip:

```
choco install -y peazip
```

- You'll also need SVN, since some of the Python source-build dependencies are checked out via SVN:

```
choco install -y svn hg
```

- You'll need to quit and restart the command prompt after installing the above.
- Get and extract the Python 3.8.3 source from the tgz:
To keep these instructions concise, please extract it to C:\dev\Python-3.8.3

- Now, build the Python source in debug mode from a Visual Studio command prompt:

```bash
cd C:\dev\Python-3.8.3\PCbuild
getExternals.bat
build.bat -p x64 -d
```

- Finally, copy the build products into the Python38 installation directories, next to the Release-mode Python executable and DLL's:

```bash
cd C:\dev\Python-3.8.3\PCbuild\amd64
copy python_d.exe C:\Python38 /Y
copy python38_d.dll C:\Python38 /Y
copy python3_d.dll C:\Python38 /Y
copy python3_d.lib C:\Python38\libs /Y
copy python3_d.lib C:\Python38\libs /Y
copy sqlite3_d.dll C:\Python38\DLLs /Y
for %I in (_d.pyd) do copy %I C:\Python38\DLLs /Y
```

- Now, from a fresh command prompt, make sure that python_d works:

```bash
python_d -c "import _ctypes ; import coverage"
```

- Once you have verified the operation of python_d, it is necessary to reinstall a few dependencies with the debug-enabled libraries:

```bash
python_d -m pip install --force-reinstall https://github.com/ros2/ros2/releases/download/˓numpy-archives/numpy-1.18.4-cp38-cp38d-win_amd64.whl
python_d -m pip install --force-reinstall https://github.com/ros2/ros2/releases/download/˓lxml-archives/lxml-4.5.1-cp38-cp38d-win_amd64.whl
```

- To verify the installation of these dependencies:

```bash
python_d -c "from lxml import etree ; import numpy"
```

- When you wish to return to building release binaries, it is necessary to uninstall the debug variants and use the release variants:

```bash
python -m pip uninstall numpy lxml
python -m pip install numpy lxml
```

- To create executables python scripts(.exe), python_d should be used to invoke colcon

```bash
python_d path\to\colcon_executable build
```

- Hooray, you're done!
Stay up to date

See *Maintain source checkout* to periodically refresh your source installation.

Troubleshoot

Troubleshooting techniques can be found *here*.

Uninstall

1. If you installed your workspace with colcon as instructed above, “uninstalling” could be just a matter of opening a new terminal and not sourcing the workspace’s setup file. This way, your environment will behave as though there is no Iron install on your system.

2. If you’re also trying to free up space, you can delete the entire workspace directory with:

   ```
   rmdir /s /q \ros2_iron
   ```

RHEL (source)
System requirements

The current target Red Hat platforms for Iron Irwini are:

- Tier 2: RHEL 9 64-bit

As defined in REP 2000.

System setup

Set locale

Make sure you have a locale which supports UTF-8. If you are in a minimal environment (such as a docker container), the locale may be something minimal like C. We test with the following settings. However, it should be fine if you’re using a different UTF-8 supported locale.

```
locale  # check for UTF-8

sudo dnf install langpacks-en glibc-langpack-en
export LANG=en_US.UTF-8
locale  # verify settings
```

Enable required repositories

RHEL

The rosdep database contains packages from the EPEL and PowerTools repositories, which are not enabled by default. They can be enabled by running:

```
sudo dnf install 'dnf-command(config-manager)' epel-release -y
sudo dnf config-manager --set-enabled crb
```

Note: This step may be slightly different depending on the distribution you are using. Check the EPEL documentation:
https://docs.fedoraproject.org/en-US/epel/#_quickstart

Fedora

No additional setup required.

Install development tools

```
sudo dnf install -y \
  cmake \
  gcc-c++ \
  git \
  make \
  patch \
  python3-colcon-common-extensions \
  python3-flake8-builtin
```

(continues on next page)
Build ROS 2

Get ROS 2 code

Create a workspace and clone all repos:

```bash
mkdir -p ~/ros2_iron/src
cd ~/ros2_iron
vcs import --input https://raw.githubusercontent.com/ros2/ros2/iron/ros2.repos src
```

Install dependencies using rosdep

ROS 2 packages are built on frequently updated RHEL systems. It is always recommended that you ensure your system is up to date before installing new packages.

```bash
sudo dnf update

sudo rosdep init
rosdep update
rosdep install --from-paths src --ignore-src -y --skip-keys "assimp fastcdr ignition-...
->cmake2 ignition-math6 python3-matplotlib python3-pygraphviz rti-connext-dds-6.0.1
->urdfdom_headers"
```
Install additional RMW implementations (optional)

The default middleware that ROS 2 uses is Fast DDS, but the middleware (RMW) can be replaced at build or runtime. See the guide on how to work with multiple RMWs.

Build the code in the workspace

If you have already installed ROS 2 another way (either via RPMs or the binary distribution), make sure that you run the below commands in a fresh environment that does not have those other installations sourced. Also ensure that you do not have source /opt/ros/${ROS_DISTRO}/setup.bash in your .bashrc. You can make sure that ROS 2 is not sourced with the command printenv | grep -i ROS. The output should be empty.

More info on working with a ROS workspace can be found in this tutorial.

```
   cd ~/ros2_iron/
colcon build --symlink-install
```

**Note:** If you are having trouble compiling all examples and this is preventing you from completing a successful build, you can use the --packages-skip colcon flag to ignore the package that is causing problems. For instance, if you don’t want to install the large OpenCV library, you could skip building the packages that depend on it using the command:

```
   colcon build --symlink-install --packages-skip image_tools intra_process_demo
```

Setup environment

Set up your environment by sourcing the following file.

```
   # Replace ".bash" with your shell if you're not using bash
   # Possible values are: setup.bash, setup.sh, setup.zsh
   . ~/ros2_iron/install/local_setup.bash
```

Try some examples

In one terminal, source the setup file and then run a C++ talker:

```
   . ~/ros2_iron/install/local_setup.bash
   ros2 run demo_nodes_cpp talker
```

In another terminal source the setup file and then run a Python listener:

```
   . ~/ros2_iron/install/local_setup.bash
   ros2 run demo_nodes_py listener
```

You should see the talker saying that it’s Publishing messages and the listener saying I heard those messages. This verifies both the C++ and Python APIs are working properly. Hooray!
Next steps

Continue with the tutorials and demos to configure your environment, create your own workspace and packages, and learn ROS 2 core concepts.

Alternate compilers

Using a different compiler besides gcc to compile ROS 2 is easy. If you set the environment variables CC and CXX to executables for a working C and C++ compiler, respectively, and retrigger CMake configuration (by using --force-cmake-config or by deleting the packages you want to be affected), CMake will reconfigure and use the different compiler.

Clang

To configure CMake to detect and use Clang:

```
sudo dnf install clang
export CC=clang
export CXX=clang++
colcon build --cmake-force-configure
```

Stay up to date

See Maintain source checkout to periodically refresh your source installation.

Troubleshoot

Troubleshooting techniques can be found here.

Uninstall

1. If you installed your workspace with colcon as instructed above, “uninstalling” could be just a matter of opening a new terminal and not sourcing the workspace’s setup file. This way, your environment will behave as though there is no Iron install on your system.

2. If you’re also trying to free up space, you can delete the entire workspace directory with:

```
rm -rf ~/ros2_iron
```
RHEL (binary)

This page explains how to install ROS 2 on RHEL from a pre-built binary package.

Note: The pre-built binary does not include all ROS 2 packages. All packages in the ROS base variant are included, and only a subset of packages in the ROS desktop variant are included. The exact list of packages are described by the repositories listed in this ros2.repos file.

There are also RPM packages available.

System requirements

We currently support RHEL 9 64-bit.

System setup

Set locale

Make sure you have a locale which supports UTF-8. If you are in a minimal environment (such as a docker container), the locale may be something minimal like C. We test with the following settings. However, it should be fine if you’re using a different UTF-8 supported locale.

```
locale  # check for UTF-8
sudo dnf install langpacks-en glibc-langpack-en
```

(continues on next page)
export LANG=en_US.UTF-8
locale  # verify settings

Enable required repositories

The `rosdist` database contains packages from the EPEL and PowerTools repositories, which are not enabled by default. They can be enabled by running:

```
sudo dnf install 'dnf-command(config-manager)' epel-release -y
sudo dnf config-manager --set-enabled crb
```

**Note:** This step may be slightly different depending on the distribution you are using. Check the EPEL documentation:
https://docs.fedoraproject.org/en-US/epel/#_quickstart

Install prerequisites

There are a few packages that must be installed in order to get and unpack the binary release.

```
sudo dnf install tar bzip2 wget -y
```

Install development tools (optional)

If you are going to build ROS packages or otherwise do development, you can also install the development tools:

```
sudo dnf install -y \
cmake \
gcc-c++ \
git \
make \
patch \
python3-colcon-common-extensions \
python3-flake8-bultins \
python3-flake8-comprehensions \
python3-flake8-docstrings \
python3-flake8-import-order \
python3-flake8-quotes \
python3-mypy \
python3-pip \
python3-pydocstyle \
python3-pytest \
python3-pytest-repeat \
python3-pytest-reunfailures \
python3-rosdep \
python3-setuptools \
python3-vcstool \
wget
```

(continues on next page)
# install some pip packages needed for testing and
# not available as RPMs
python3 -m pip install -U --user
   flake8-blind-except==0.1.1
   flake8-class-newline
   flake8-deprecated

Install ROS 2

- Go to the releases page
- Download the latest package for RHEL; let's assume that it ends up at ~/Downloads/ros2-package-linux-x86_64.tar.bz2.
  - Note: there may be more than one binary download option which might cause the file name to differ.
- Unpack it:

```bash
mkdir -p ~/ros2_iron
cd ~/ros2_iron
tar xf ~/Downloads/ros2-package-linux-x86_64.tar.bz2
```

Install dependencies using rosdep

ROS 2 packages are built on frequently updated RHEL systems. It is always recommended that you ensure your system is up to date before installing new packages.

```bash
sudo dnf update
rosdep install --from-paths ~/ros2_iron/ros2-linux/share --ignore-src -y --skip-keys
   "assimp cyclonedds fastcdr fastrtps ignition-cmake2 ignition-math6 python3-matplotlib
   python3-pygraphviz rti-connext-dds-6.0.1 urdfdom_headers"
```

Install additional RMW implementations (optional)

The default middleware that ROS 2 uses is Fast DDS, but the middleware (RMW) can be replaced at runtime. See the guide on how to work with multiple RMWs.

Setup environment

Set up your environment by sourcing the following file.

```bash
# Replace ".bash" with your shell if you're not using bash
# Possible values are: setup.bash, setup.sh, setup.zsh
. ~/ros2_iron/ros2-linux/setup.bash
```
Try some examples

In one terminal, source the setup file and then run a C++ talker:

```bash
. ~/ros2_iron/ros2-linux/setup.bash
ros2 run demo_nodes_cpp talker
```

In another terminal source the setup file and then run a Python listener:

```bash
. ~/ros2_iron/ros2-linux/setup.bash
ros2 run demo_nodes_py listener
```

You should see the talker saying that it's Publishing messages and the listener saying I heard those messages. This verifies both the C++ and Python APIs are working properly. Hooray!

Next steps

Continue with the tutorials and demos to configure your environment, create your own workspace and packages, and learn ROS 2 core concepts.

Troubleshoot

Troubleshooting techniques can be found here.

Uninstall

1. If you installed your workspace with colcon as instructed above, “uninstalling” could be just a matter of opening a new terminal and not sourcing the workspace’s setup file. This way, your environment will behave as though there is no Iron install on your system.

2. If you’re also trying to free up space, you can delete the entire workspace directory with:

```bash
rm -rf ~/ros2_iron
```

macOS (source)

Table of Contents

- System requirements
- System setup
  - Install prerequisites
  - Disable System Integrity Protection (SIP)
- Build ROS 2
  - Get ROS 2 code
  - Install additional RMW implementations (optional)
Build the code in the workspace

- Setup environment
- Try some examples
- Next steps
- Use the ROS 1 bridge (optional)
- Stay up to date
- Troubleshoot
- Uninstall

System requirements

We currently support macOS Mojave (10.14). The Rolling Ridley distribution will change target platforms from time to time as new platforms become available. Most people will want to use a stable ROS distribution.

System setup

Install prerequisites

You need the following things installed to build ROS 2:

1. Xcode
   - If you don’t already have it installed, install [Xcode](https://apps.apple.com/app/xcode/id497799835).
   - Note: Versions of Xcode later than 11.3.1 can no longer be installed on macOS Mojave, so you will need to install an older version manually, see: [https://stackoverflow.com/a/61046761](https://stackoverflow.com/a/61046761)
   - Also, if you don’t already have it installed, install the Command Line Tools:

   ```bash
   xcode-select --install
   # This command will not succeed if you have not installed Xcode.app
   sudo xcode-select --switch /Applications/Xcode.app/Contents/Developer
   # If you installed Xcode.app manually, you need to either open it or run:
   sudo xcodebuild -license
   # To accept the Xcode.app license
   ```

2. brew (needed to install more stuff; you probably already have this):
   - Follow installation instructions at [http://brew.sh/](http://brew.sh/)
   - Optional: Check that brew is happy with your system configuration by running:

   ```bash
   brew doctor
   ```
   Fix any problems that it identifies.

3. Use brew to install more stuff:

   ```bash
   brew install asio assimp bison bullet cmake console_bridge cppcheck \
   cunit eigen freetype graphviz opencv openssl orocos-kdl pcre poco \
   pyqt5 python qt@5 sip spdlog tinyxml tinyxml2
   ```

4.8. ROS 2 Documentation
4. Setup some environment variables:

```bash
# Add the openssl dir for DDS-Security
# if you are using BASH, then replace '.zshrc' with '.bashrc'
echo "export OPENSSL_ROOT_DIR=$(/bin/bash --prefix openssl)" >> ~/.zshrc

# Add the Qt directory to the PATH and CMAKE_PREFIX_PATH
export CMAKE_PREFIX_PATH=CMAKE_PREFIX_PATH:$(/bin/bash --prefix qt@5)
export PATH=PATH:$(/bin/bash --prefix qt@5)/bin
```

5. Use `python3 -m pip` (just `pip` may install Python3 or Python2) to install more stuff:

```bash
python3 -m pip install -U \
    argcomplete catkin_pkg colcon-common-extensions coverage \
    cryptography empy flake8 flake8-blind-except==0.1.1 flake8-builtins \
    flake8-class-newline flake8-comprehensions flake8-deprecated \
    flake8-docstrings flake8-import-order flake8-quotes \
    importlib-metadata jsonschema lark==1.1.1 lxml matplotlib mock mypy==0.931␣
˓→netifaces \n    nose pep8 psutil pydocstyle pydot pygraphviz pyparsing==2.4.7 \n    pytest-mock rosdep rosdistro setuptools==59.6.0 vcstool
```

Please ensure that the $PATH environment variable contains the install location of the binaries ($(/bin/bash --prefix)/bin)

6. Optional: if you want to build the ROS 1<->2 bridge, then you must also install ROS 1:

- Start with the normal install instructions: http://wiki.ros.org/kinetic/Installation/OSX/Homebrew/Source
- When you get to the step where you call `rosinstall_generator` to get the source code, here’s an alternate invocation that brings in just the minimum required to produce a useful bridge:

```bash
rosinstall_generator catkin common_msgs roscpp rosmsg --rosdistro kinetic -- \
    --deps --wet-only --tar > kinetic-ros2-bridge-deps.rosinstall
wstool init -j8 src kinetic-ros2-bridge-deps.rosinstall
```

Otherwise, just follow the normal instructions, then source the resulting `install_isolated/setup.bash` before proceeding here to build ROS 2.
Disable System Integrity Protection (SIP)

macOS/OS X versions >=10.11 have System Integrity Protection enabled by default. So that SIP doesn’t prevent processes from inheriting dynamic linker environment variables, such as DYLD_LIBRARY_PATH, you’ll need to disable it following these instructions.

Build ROS 2

Get ROS 2 code

Create a workspace and clone all repos:

```bash
mkdir -p ~/ros2_iron/src
cd ~/ros2_iron
vcs import --input https://raw.githubusercontent.com/ros2/ros2/iron/ros2.repos src
```

Install additional RMW implementations (optional)

The default middleware that ROS 2 uses is Fast DDS, but the middleware (RMW) can be replaced at build or runtime. See the guide on how to work with multiple RMWs.

Build the code in the workspace

Run the colcon tool to build everything (more on using colcon in this tutorial):

```bash
cd ~/ros2_iron/
colcon build --symlink-install --packages-skip-by-dep python_qt_binding
```

Note: due to an unresolved issue with SIP, Qt@5, and PyQt5, we need to disable python_qt_binding to have the build succeed. This will be removed when the issue is resolved, see: https://github.com/ros-visualization/python_qt_binding/issues/103

Setup environment

Source the ROS 2 setup file:

```
. ~/ros2_iron/install/setup.zsh
```

This will automatically set up the environment for any DDS vendors that support was built for.
Try some examples

In one terminal, set up the ROS 2 environment as described above and then run a C++ talker:

```
ros2 run demo_nodes_cpp talker
```

In another terminal source the setup file and then run a Python listener:

```
ros2 run demo_nodes_py listener
```

You should see the talker saying that it’s Publishing messages and the listener saying I heard those messages. This verifies both the C++ and Python APIs are working properly. Hooray!

Next steps

Continue with the tutorials and demos `<../../Tutorials>` to configure your environment, create your own workspace and packages, and learn ROS 2 core concepts.

Use the ROS 1 bridge (optional)

The ROS 1 bridge can connect topics from ROS 1 to ROS 2 and vice-versa. See the dedicated documentation on how to build and use the ROS 1 bridge.

Stay up to date

See Maintain source checkout to periodically refresh your source installation.

Troubleshoot

Troubleshooting techniques can be found here.

Uninstall

1. If you installed your workspace with colcon as instructed above, “uninstalling” could be just a matter of opening a new terminal and not sourcing the workspace’s setup file. This way, your environment will behave as though there is no Iron install on your system.

2. If you’re also trying to free up space, you can delete the entire workspace directory with:

   ```
   rm -rf ~/ros2_iron
   ```
Latest development (source)

If you plan to contribute directly to the latest ROS 2 development, you can install ROS 2 by building it from source or installing testing binaries. This will give you the latest bug fixes and features.

Testing binaries

See Testing.

Build from source

Note: The latest development does not go through the same rigorous testing as releases and is not recommended if you are looking for a stable version of ROS 2. Instead, choose a ROS distribution that has already been released.

Follow the links below for the latest setup instructions for your platform:

- Ubuntu Linux
- Windows
- RHEL
- macOS

For keeping your source code up-to-date, see Maintain a source checkout.

Maintain source checkout

- Update your repository list
  - Latest ROS 2 Iron branches
- Update your repositories
- Download the new source code
- Rebuild your workspace
- Inspect your source checkout

If you have installed ROS 2 from source, there may have been changes made to the source code since the time that you checked it out. To keep your source checkout up to date, you will have to periodically update your ros2.repos file, download the latest sources, and rebuild your workspace.
Update your repository list

Each ROS 2 release includes a `ros2.repos` file that contains the list of repositories and their version for that release.

Latest ROS 2 Iron branches

If you wish to checkout the latest code for ROS 2 Iron, you can get the relevant repository list by running:

Linux

```bash
cd ~/ros2_iron
mv -i ros2.repos ros2.repos.old
wget https://raw.githubusercontent.com/ros2/ros2/iron/ros2.repos
```

macOS

```bash
cd ~/ros2_iron
mv -i ros2.repos ros2.repos.old
wget https://raw.githubusercontent.com/ros2/ros2/iron/ros2.repos
```

Windows

```bash
# CMD
cd \dev\ros2_iron

# PowerShell
cd \dev\ros2_iron
curl https://raw.githubusercontent.com/ros2/ros2/iron/ros2.repos -o ros2.repos
```

Update your repositories

You will notice that in the `ros2.repos` file, each repository has a version associated with it that points to a particular commit hash, tag, or branch name. It is possible that these versions refer to new tags/branches that your local copy of the repositories will not recognize as they are out-of-date. Because of this, you should update the repositories that you have already checked out with the following command:

```bash
vcs custom --args remote update
```

Download the new source code

You should now be able to download the sources associated with the new repository list with:

Linux

```bash
vcs import src < ros2.repos
vcs pull src
```

macOS

```bash
vcs import src < ros2.repos
vcs pull src
```
Windows

```
# CMD
vcs import src < ros2.repos
vcs pull src

# PowerShell
vcs import --input ros2.repos src
vcs pull src
```

Rebuild your workspace

Now that the workspace is up to date with the latest sources, remove your previous install and rebuild your workspace with, for example:

```
colcon build --symlink-install
```

Inspect your source checkout

During your development you may have deviated from the original state of your workspace from when you imported the repository list. If you wish to know the versions of the set of repositories in your workspace, you can export the information using the following command:

Linux

```
cd ~/ros2_iron
vcs export src > my_ros2.repos
```

macOS

```
cd ~/ros2_iron
vcs export src > my_ros2.repos
```

Windows

```
cd \dev\ros2_iron
vcs export src > my_ros2.repos
```

This `my_ros2.repos` file can then be shared with others so that they can reproduce the state of the repositories in your workspace.

Testing with pre-release binaries

Many ROS packages are provided as pre-built binaries. Usually, you will get the released version of binaries when following Installation. There are also pre-released versions of binaries that are useful for testing before making an official release. This article describes several options if you would like to try out pre-released versions of ROS binaries.
Debian testing repository

When packages are released into a ROS distribution (using bloom), the buildfarm builds them into debian packages which are stored temporarily in the building apt repository. As dependent packages are rebuilt, an automatic process periodically synchronizes the packages in building to a secondary repository called ros-testing. ros-testing is intended as a soaking area where developers and bleeding-edge users may give the packages extra testing, before they are manually synced into the public ros repository from which users typically install packages.

Approximately every two weeks, the rosdistro’s release manager manually synchronizes the contents of ros-testing into the main ROS repository.

For Debian-based operating systems, you can install binary packages from the ros-testing repository.

1. Make sure you have a working ROS 2 installation from Debian packages (see Installation).
2. Edit (with sudo) the file /etc/apt/sources.list.d/ros2.list and change ros2 with ros2-testing. For example, on Ubuntu Jammy the contents should look like the following:

```
# deb http://packages.ros.org/ros2/ubuntu jammy main
deb http://packages.ros.org/ros2-testing/ubuntu jammy main
```

3. Update the apt index:

```
sudo apt update
```

4. You can now install individual packages from the testing repository, for example:

```
sudo apt install ros-iron-my-just-released-package
```

5. Alternatively, you can move your entire ROS 2 installation to the testing repository:

```
sudo apt dist-upgrade
```

6. Once you are finished testing, you can switch back to the normal repository by changing back the contents of /etc/apt/sources.list.d/ros2.list:

```
deb http://packages.ros.org/ros2/ubuntu jammy main
# deb http://packages.ros.org/ros2-testing/ubuntu jammy main
```

and doing an update and upgrade:

```
sudo apt update
sudo apt dist-upgrade
```

Fat binaries

For core packages, we run nightly packaging jobs for Ubuntu Linux, RHEL, and Windows. These packaging jobs produce archives with pre-built binaries that can be downloaded and extracted to your filesystem.

1. Make sure you have all dependencies installed according to the latest development setup for your platform.
2. Go to https://ci.ros2.org/view/packaging/ and select a packaging job from the list corresponding to your platform.
3. Under the heading “Last Successful Artifacts” you should see a download link (e.g. for Windows, ros2-package-windows-AMD64.zip).
4. Download and extract the archive to your file system.
5. To use the fat binary installation, source the `setup.*` file that can be found in the root of the archive.

   **Ubuntu Linux and RHEL**

   ```bash
   source path/to/extracted/archive/setup.bash
   ```

   **Windows**

   ```bash
   call path\to\extracted\archive\setup.bat
   ```

**Docker**

For Ubuntu Linux, there is also a nightly Docker image based on the nightly fat archive.

1. Pull the Docker image:

   ```bash
   docker pull osrf/ros2:nightly
   ```

2. Start an interactive container:

   ```bash
   docker run -it osrf/ros2:nightly
   ```

For support on running GUI applications in Docker, take a look at the tutorial [User GUI’s with Docker](#) or the tool [rocker](#).

**DDS implementations**

By default, ROS 2 uses DDS as its middleware. It is compatible with multiple DDS or RTPS (the DDS wire protocol) vendors. There is currently support for eProsima’s Fast DDS, RTI’s Connext DDS, Eclipse Cyclone DDS, and GurumNetworks GurumDDS. See [https://ros.org/reps/rep-2000.html](https://ros.org/reps/rep-2000.html) for supported DDS vendors by distribution.

The default DDS vendor is eProsima’s Fast DDS.

- [Working with Eclipse Cyclone DDS](#) explains how to utilize Cyclone DDS.
- [Working with eProsima Fast DDS](#) explains how to utilize Fast DDS.
- [Working with GurumNetworks GurumDDS](#) explains how to utilize GurumDDS.

**Connext security plugins**

The Connext DDS Libraries are included with ROS 2 under a non-commercial license and do not include the security plug-in libraries. These libraries are available in the commercial, university and research license versions of RTI Connext DDS Pro, which is bundled with tools for system debugging, monitoring, record/replay, etc.

The Connext DDS Evaluation Version (6.0.1) includes the security plugins, and can be downloaded via options available for university, purchase or evaluation.

A video walk-thru of this installation (tools and security plug-ins) is available [here](#) at the RTI website. The steps are:

**Install Connext DDS Pro (Host)** This is a host-specific installer application (for Windows, Linux, MacOS) to install a ‘Host’ bundle which includes the Launcher, tools, and other software services. At the end of the installation, the RTI ‘Launcher’ program will be started. The Launcher is used to install target libraries, security plugins, and other layered services.

**Use the Package Installer in Launcher**
Fig. 1: Launcher Image
The `RTI Package Installer` is used to install `.rtipkg` files – target libraries, security plug-ins, etc. Open the Package Installer and select all of the .rtipkg files that were included in the Connext DDS Secure bundle for installation:

- Target Libraries - such as: rti_connext_dds-[version]-pro-target-[toolchain].rtipkg
- Security Plugin Host - such as: rti_securityPlugins-[version]-host-[toolchain].rtipkg
- Security Plugin Target - such as: rti_securityPlugins-[version]-target-[toolchain].rtipkg
- OpenSSL Host - such as: openssl-1.0.2x-[version]-host-[toolchain].rtipkg

**Extract and Install OpenSSL.** This is included as an archive (.zip or otherwise) and can be simply extracted and copied to a convenient location on your host computer. As a suggestion, this could also be installed into the `rti_connext_dds-[version]` directory in your home directory space (this was created during installation of the RTI host tools). Note: this directory location may need to be placed in your PATH environment variable. See the RTI Security Plugins Getting Started Guide for more information.

Installation complete.

**RTI Connext DDS**

A libraries-only version of RTI Connext DDS 6.0.1 may be installed per the *installation instructions* for Debian/Ubuntu Linux (amd64) platforms only, under a non-commercial license.

A full-suite installation of RTI Connext DDS is available for many additional platforms, for universities, evaluation, or purchase. This installation includes diagnostic tools, layered services, and security. See below for installation details.

**RTI University Program**

University researchers and classroom users may be eligible for a free academic license through the RTI University Program. This includes a one-year (renewable) license to the unabridged version of Connext DDS Secure, which includes diagnostic tools and layered services. The university license application can be found here.

**RTI Connext DDS Evaluation**

To install RTI Connext DDS version 6.0.1 Evaluation:

- Visit the RTI Free Trial (6.0.1) site.
- Download the version(s) to match your environment.
- Contact license@rti.com for an evaluation license.
- Install RTI Connext 6.0.1 by running the installation program. When finished, it will run the RTI Launcher.
- Use the RTI Launcher to install the license file (rti_license.dat) if needed. The launcher may also be used to launch the diagnostic tools and services.
Eclipse Cyclone DDS

Eclipse Cyclone DDS is a very performant and robust open-source DDS implementation. Cyclone DDS is developed completely in the open as an Eclipse IoT project. See also: https://projects.eclipse.org/projects/iot.cyclonedds

Prerequisites

Have rosdep installed

Install packages

The easiest way is to install from ROS 2 apt repository.

```
sudo apt install ros-iron-rmw-cyclonedds-cpp
```

Build from source code

Building from source code is also another way to install.

First, clone Cyclone DDS and rmw_cyclonedds in the ROS 2 workspace source directory.

```
cd ros2_ws/src
git clone https://github.com/ros2/rmw_cyclonedds ros2/rmw_cyclonedds -b iron
git clone https://github.com/eclipse-cyclonedds/cyclonedds eclipse-cyclonedds/cyclonedds
```

Then, install necessary packages for Cyclone DDS.

```
cd ..
rosdep install --from src -i
```

Finally, run colcon build.

```
colcon build --symlink-install
```

Switch to rmw_cyclonedds

Switch from other rmw to rmw_cyclonedds by specifying the environment variable.

```
export RMW_IMPLEMENTATION=rmw_cyclonedds_cpp
```

See also: Working with multiple RMW implementations
Run the talker and listener

Now run `talker` and `listener` to test Cyclone DDS.

```bash
ros2 run demo_nodes_cpp talker

ros2 run demo_nodes_cpp listener
```

GurumNetworks GurumDDS

rmw_gurumdds is a implementation of the ROS middleware interface using GurumNetworks GurumDDS. More information about GurumDDS is available on our website: [https://gurum.cc/index_eng](https://gurum.cc/index_eng)

Prerequisites

The following description assumes that you have completed the ‘Environment setup’ process from the *Installing ROS 2 via Debian Packages* or from the *Building ROS 2 on Ubuntu Linux*.

rmw_gurumdds requires version of GurumDDS-2.8.x. Debian packages of GurumDDS are provided in the ROS 2 apt repositories on ubuntu. Windows binary installer of GurumDDS will be supported soon.

GurumDDS requires a license. See the next page: [https://gurum.cc/free_trial_eng.html](https://gurum.cc/free_trial_eng.html)

After requesting a trial license, please download the license from the license homepage. After getting a license, move it to the following location.

<table>
<thead>
<tr>
<th>DDS Version</th>
<th>License Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;= 2.7.2860</td>
<td>/etc/flame</td>
</tr>
<tr>
<td>&gt;= 2.7.2861</td>
<td>/etc/gurumnet</td>
</tr>
<tr>
<td>2.8.x</td>
<td>/etc/gurumnet</td>
</tr>
</tbody>
</table>

Install packages

The easiest way is to install from ROS 2 apt repository. When `ros-iron-rmw-gurumdds-cpp` is installed, `gurumdds-2.8` is also installed.

```bash
sudo apt install ros-iron-rmw-gurumdds-cpp
```

Build from source code

Building from source code is also another way to install.

First, clone `rmw_gurumdds` in the ROS 2 workspace source directory.

```bash
cd ros2_ws/src
git clone https://github.com/ros2/rmw_gurumdds -b iron ros2/rmw_gurumdds
```

Then, install necessary packages for GurumDDS.
Finally, run colcon build.

```bash
colcon build --symlink-install
```

### Switch to rmw_gurumdds

Switch from other rmw to rmw_gurumdds by specifying the environment variable.

```bash
export RMW_IMPLEMENTATION=rmw_gurumdds_cpp
```

See also: *Working with multiple RMW implementations*

### Run the talker and listener

Now run `talker` and `listener` to test GurumDDS. Don’t forget to set up environment by setup script.

```bash
ros2 run demo_nodes_cpp talker
ros2 run demo_nodes_cpp listener
```

### eProsima Fast DDS

eProsima Fast DDS is a complete open-source DDS implementation for real-time embedded architectures and operating systems. See also: [https://www.eprosima.com/index.php/products-all/eprosima-fast-dds](https://www.eprosima.com/index.php/products-all/eprosima-fast-dds)

### Prerequisites

Have rosdep installed

### Install packages

The easiest way is to install from ROS 2 apt repository.

```bash
sudo apt install ros-iron-rmw-fastrtps-cpp
```
**Build from source code**

Building from source code is also another way to install.

First, clone Fast DDS and rmw_fastrtps in the ROS 2 workspace source directory.

```
cd ros2_ws/src
git clone https://github.com/ros2/rmw_fastrtps ros2/rmw_fastrtps -b iron
git clone https://github.com/eProsima/Fast-DDS eProsima/fastrtps
```

Then, install necessary packages for Fast DDS.

```
cd ..
rosdep install --from src -i
```

Finally, run colcon build.

```
colcon build --symlink-install
```

**Switch to rmw_fastrtps**

The eProsima Fast DDS RMW can be selected by specifying the environment variable:

```
export RMW_IMPLEMENTATION=rmw_fastrtps_cpp
```

See also: [*Working with multiple RMW implementations*](#)

**Run the talker and listener**

Now run `talker` and `listener` to test Fast DDS.

```
ros2 run demo_nodes_cpp talker
ros2 run demo_nodes_cpp listener
```

If you would like to use one of the other vendors you will need to install their software separately before building. The ROS 2 build will automatically build support for vendors that have been installed and sourced correctly.

Once you’ve installed a new DDS vendor, you can change the vendor used at runtime: [*Working with Multiple RMW Implementations*](#).

Detailed instructions for installing other DDS vendors are provided below.

**Platforms / Installation types**

- *Ubuntu Linux source install*
- *Ubuntu Linux binary install*
- *OSX source install*
- *OSX binary install*
- *Windows source install*
Ubuntu Linux source install

RTI Connext (version 6.0.1, amd64 only)

Debian packages provided in the ROS 2 apt repositories

You can install a Debian package of RTI Connext available on the ROS 2 apt repositories. You will need to accept a license from RTI.

```
sudo apt update && sudo apt install -q -y rti-connext-dds-6.0.1
```

Source the setup file to set the `NDDSHOME` environment variable.

```
cd /opt/rti.com/rti_connext_dds-6.0.1/resource/scripts && source ./rtisetenv_x64Linux4gcc7.3.0.bash; cd -
```

Note: when using zsh you need to be in the directory of the script when sourcing it to have it work properly

Now you can build as normal and support for RTI will be built as well.

Official binary packages from RTI

You can install the Connext 6.0.1 package for Linux provided by RTI, via options available for university, purchase or evaluation

After downloading, use `chmod +x` on the `.run` executable and then execute it. Note that if you’re installing to a system directory use `sudo` as well.

The default location is `~/rti_connext_dds-6.0.1`

After installation, run RTI launcher and point it to your license file (obtained from RTI).

Add the following line to your `.bashrc` file pointing to your copy of the license.

```
export RTI_LICENSE_FILE=path/to/rti_license.dat
```

Source the setup file to set the `NDDSHOME` environment variable.

```
cd ~/rti_connext_dds-6.0.1/resource/scripts && source ./rtisetenv_x64Linux4gcc7.3.0.bash; cd -
```

Now you can build as normal and support for RTI will be built as well.
Ubuntu Linux binary install

RTI Connext (version 6.0.1, amd64 only)

To use RTI Connext DDS there are full-suite install options available for university, purchase or evaluation or you can install a libraries-only Debian package of RTI Connext 6.0.1, available from the OSRF Apt repository under a non-commercial license.

To install the libs-only Debian package:

```
sudo apt update && sudo apt install -q -y rti-connext-dds-6.0.1
```

You will need to accept a license agreement from RTI, and will find an `rti_license.dat` file in the installation.

Add the following line to your `.bashrc` file pointing to your copy of the license (and source it).

```
export RTI_LICENSE_FILE=path/to/rti_license.dat
```

All options need you to source the setup file to set the `NDDSHOME` environment variable:

```
cd /opt/rti.com/rti_connext.dds-6.0.1/resource/scripts && source ./rtisetenv_x64Linux4gcc7.3.0.bash; cd -
```

Note: the above may need modification to match your RTI installation location

If you want to install the Connext DDS-Security plugins please refer to this page.

OSX source install

RTI Connext (6.0.1)

If you would like to also build against RTI Connext DDS there are options available for university, purchase or evaluation

You also need a Java runtime installed to run the RTI code generator, which you can get here.

After installing, run RTI launcher and point it to your license file.

Source the setup file to set the `NDDSHOME` environment variable before building your workspace.

```
source /Applications/rti_connext.dds-6.0.1/resource/scripts/rtisetenv_x64Darwin17clang9.0.bash
```

You may need to increase shared memory resources following https://community.rti.com/kb/osx510

If you want to install the Connext DDS-Security plugins please refer to this page.
**OSX binary install**

**Enable Connext support**

To use RTI Connext DDS there are options available for *university, purchase or evaluation*

After installing, run RTI launcher and point it to your license file.

Source the setup file to set the `NDDSHOME` environment variable before building your workspace.

```bash
source /Applications/rti_connext_dds-6.0.1/resource/scripts/rtisetenv_x64Darwin17clang9.0.bash
```

You may need to increase shared memory resources following [https://community.rti.com/kb/osx510](https://community.rti.com/kb/osx510).

If you want to install the Connext DDS-Security plugins please refer to [*this page*](#).

**Windows source install**

**RTI Connext 6.0.1**

If you would like to also build against RTI Connext DDS there are options available for *university, purchase or evaluation*

After installing, use the RTI Launcher to load your license file.

Then before building ROS 2, set up the Connext environment:

```bash
call "C:\Program Files\rti_connext_dds-6.0.1\resource\scripts\rtisetenv_x64Win64VS2017.bat"
```

Note that this path might need to be slightly altered depending on where you selected to install RTI Connext DDS, and which version of Visual Studio was selected. The path above is the current default path as of version 6.0.1, but will change as the version numbers increment in the future.

If you want to install the Connext DDS-Security plugins please refer to [*this page*](#).

**Windows binary install**

**RTI Connext**

To use RTI Connext DDS there are options available for *university, purchase or evaluation*

After installing, run RTI launcher and point it to your license file.

Then before using ROS 2, set up the Connext environment:

```bash
call "C:\Program Files\rti_connext_dds-6.0.1\resource\scripts\rtisetenv_x64Win64VS2017.bat"
```

If you want to install the Connext DDS-Security plugins please refer to [*this page*](#).
**Binary packages**

Binaries are only created for the Tier 1 operating systems listed in REP-2000. Given the nature of Rolling, this list may be updated at any time. If you are not running any of the following operating systems you may need to build from source or use a container solution to run ROS 2 on your platform.

We provide ROS 2 binary packages for the following platforms:

- Ubuntu Linux - Jammy Jellyfish (22.04)
  - Debian packages (recommended)
  - “fat” archive
- RHEL 9
  - RPM packages (recommended)
  - “fat” archive
- Windows (VS 2019)

**Building from source**

We support building ROS 2 from source on the following platforms:

- Ubuntu Linux
- Windows
- RHEL/Fedora
- macOS

**Which install should you choose?**

Installing from binary packages or from source will both result in a fully-functional and usable ROS 2 install. Differences between the options depend on what you plan to do with ROS 2.

**Binary packages** are for general use and provide an already-built install of ROS 2. This is great for people who want to dive in and start using ROS 2 as-is, right away.

Linux users have two options for installing binary packages:

- Debian packages
- “fat” archive

Installing from Debian packages is the recommended method. It’s more convenient because it installs its necessary dependencies automatically. It also updates alongside regular system updates.

However, you need root access in order to install Debian packages. If you don’t have root access, the “fat” archive is the next best choice.

macOS and Windows users who choose to install from binary packages only have the “fat” archive option (Debian packages are exclusive to Ubuntu/Debian).

**Building from source** is meant for developers looking to alter or explicitly omit parts of ROS 2’s base. It is also recommended for platforms that don’t support binaries. Building from source also gives you the option to install the absolute latest version of ROS 2.
Contributing to ROS 2 core?

If you plan to contribute directly to ROS 2 core packages, you can install the latest development from source which shares installation instructions with the Rolling distribution.

4.8.2 Distributions

What is a Distribution?

A ROS distribution is a versioned set of ROS packages. These are akin to Linux distributions (e.g. Ubuntu). The purpose of the ROS distributions is to let developers work against a relatively stable codebase until they are ready to roll everything forward. Therefore once a distribution is released, we try to limit changes to bug fixes and non-breaking improvements for the core packages (every thing under ros-desktop-full). That generally applies to the whole community, but for “higher” level packages, the rules are less strict, and so it falls to the maintainers of a given package to avoid breaking changes.

List of Distributions

Below is a list of current and historic ROS 2 distributions. Rows in the table marked in green are the currently supported distributions.

Iron Irwini (iron)

Iron Irwini Changelog

This page is a list of the complete changes in all ROS 2 core packages since the previous release.

Table of Contents

- action_msgs
- action_tutorials_cpp
- action_tutorials_interfaces
- action_tutorials_py
- actionlib_msgs
- ament_clang_format
- ament_clang_tidy
- ament_cmake
- ament_cmake_auto
- ament_cmake_clang_format
- ament_cmake_clang_tidy
- ament_cmake_copy
- ament_cmake_core
- ament_cmake_cppcheck
• ament_cmake_cpplint
• ament_cmake_export_definitions
• ament_cmake_export_dependencies
• ament_cmake_export_include_directories
• ament_cmake_export_interfaces
• ament_cmake_export_libraries
• ament_cmake_export_link_flags
• ament_cmake_export_targets
• ament_cmake_flake8
• ament_cmake_gen_version_h
• ament_cmake_gmock
• ament_cmake_google_benchmark
• ament_cmake_gtest
• ament_cmake_include_directories
• ament_cmake_libraries
• ament_cmake_lint_cmake
• ament_cmake_mypy
• ament_cmake_pclint
• ament_cmake_pep257
• ament_cmake_pycodestyle
• ament_cmake_pyflakes
• ament_cmake_pytest
• ament_cmake_python
• ament_cmake_ros
• ament_cmake_target_dependencies
• ament_cmake_test
• ament_cmake_uncrustify
• ament_cmake_vendor_package
• ament_cmake_version
• ament_cmake_xmllint
• ament_copyright
• ament_cppcheck
• ament_cpplint
• ament_flake8
• ament_index_cpp
• *ament_index_python*
• *ament_lint*
• *ament_lint_auto*
• *ament_lint_cmake*
• *ament_lint_common*
• *ament_mypy*
• *ament_package*
• *ament_pclint*
• *ament_pep257*
• *ament_pycodestyle*
• *ament_pyflakes*
• *ament_uncrustify*
• *ament_xmllint*
• *builtin_interfaces*
• *camera_calibration_parsers*
• *camera_info_manager*
• *class_loader*
• *common_interfaces*
• *composition*
• *composition_interfaces*
• *demo_nodes_cpp*
• *demo_nodes_cpp_native*
• *demo_nodes_py*
• *diagnostic_msgs*
• *domain_coordinator*
• *dummy_map_server*
• *dummy_robot_bringup*
• *dummy_sensors*
• *eigen3_cmake_module*
• *example_interfaces*
• *examples_rclcpp_async_client*
• *examples_rclcpp_cbg_executor*
• *examples_rclcpp_minimal_action_client*
• *examples_rclcpp_minimal_action_server*
• *examples_rclcpp_minimal_client*
- examples_rclcpp_minimal_composition
- examples_rclcpp_minimal_publisher
- examples_rclcpp_minimal_service
- examples_rclcpp_minimal_subscriber
- examples_rclcpp_minimal_timer
- examples_rclcpp_multithreaded_executor
- examples_rclcpp_wait_set
- examples_rclpy_executors
- examples_rclpy_guard_conditions
- examples_rclpy_minimal_action_client
- examples_rclpy_minimal_action_server
- examples_rclpy_minimal_client
- examples_rclpy_minimal_publisher
- examples_rclpy_minimal_service
- examples_rclpy_minimal_subscriber
- examples_rclpy_pointcloud_publisher
- examples_tf2_py
- fastrtps_cmake_module
- foonathan_memory_vendor
- geometry2
- geometry_msgs
- google_benchmark_vendor
- ignition_cmake2_vendor
- ignition_math6_vendor
- image_tools
- image_transport
- interactive_markers
- intra_process_demo
- kdl_parser
- keyboard_handler
- laser_geometry
- launch
- launch_pytest
- launch_ros
- launch_testing
• `launch_testing_ament_cmake`
• `launch_testing_examples`
• `launch_testing_ros`
• `launch_xml`
• `launch_yaml`
• `libcurl_vendor`
• `libstatistics_collector`
• `libyaml_vendor`
• `lifecycle`
• `lifecycle_msgs`
• `lifecycle_py`
• `logging_demo`
• `map_msgs`
• `mcap_vendor`
• `message_filters`
• `mimick_vendor`
• `nav_msgs`
• `orocos_kdl_vendor`
• `osrf_pycommon`
• `osrf_testing_tools_cpp`
• `pendulum_control`
• `pendulum_msgs`
• `performance_test_fixture`
• `pluginlib`
• `pybind11_vendor`
• `python_cmake_module`
• `python_orocos_kdl_vendor`
• `python_qt_binding`
• `qt_dotgraph`
• `qt_gui`
• `qt_gui_app`
• `qt_gui_core`
• `qt_gui_cpp`
• `qt_gui_py_common`
• `quality_of_service_demo_cpp`
• quality_of_service_demo_py
• rcl
• rcl_action
• rcl_interfaces
• rcl_lifecycle
• rcl_logging_interface
• rcl_logging_noop
• rcl_logging_spdlog
• rcl_yaml_param_parser
• rclepp
• rclepp_action
• rclepp_components
• rclepp_lifecycle
• rclpy
• rcpputils
• rcutils
• rmw
• rmw_connextdds
• rmw_connextdds_common
• rmw_connextddsmicro
• rmw_cyclonedds_cpp
• rmw.dds_common
• rmw_fastrtpspcpp
• rmw_fastrtps_dynamic_cpp
• rmw_fastrtps_shared_cpp
• rmw_implementation
• rmw_implementation_cmake
• robot_state_publisher
• ros2action
• ros2bag
• ros2cli
• ros2cli_common_extensions
• ros2cli_test_interfaces
• ros2component
• ros2doctor
• ros2interface
• ros2launch
• ros2lifecycle
• ros2lifecycle_test_fixtures
• ros2multicast
• ros2node
• ros2param
• ros2pkg
• ros2run
• ros2service
• ros2test
• ros2topic
• ros2trace
• ros_testing
• rosbag2
• rosbag2_compression
• rosbag2_compression_zstd
• rosbag2_cpp
• rosbag2_examples_cpp
• rosbag2_examples_py
• rosbag2_interfaces
• rosbag2_performance_benchmarking
• rosbag2_performance_benchmarking_msgs
• rosbag2_py
• rosbag2_storage
• rosbag2_storage_default_plugins
• rosbag2_storage_mcap
• rosbag2_storage_sqlite3
• rosbag2_test_common
• rosbag2_test_msgdefs
• rosbag2_tests
• rosbag2_transport
• rosgraph_msgs
• rosidl_adapter
• rosidl_cli
• rosidl_cmake
• rosidl_core_generators
• rosidl_core_runtime
• rosidl_default_generators
• rosidl_default_runtime
• rosidl_dynamic_typesupport
• rosidl_dynamic_typesupport_fastrtps
• rosidl_generator_c
• rosidl_generator_cpp
• rosidl_generator_dds_idl
• rosidl_generator_py
• rosidl_generator_tests
• rosidl_generator_type_description
• rosidl_parser
• rosidl_pycommon
• rosidl_runtime_c
• rosidl_runtime_cpp
• rosidl_runtime_py
• rosidl_typesupport_c
• rosidl_typesupport_c
• rosidl_typesupport_fastrtps_c
• rosidl_typesupport_fastrtps_cpp
• rosidl_typesupport_interface
• rosidl_typesupport_introspection_c
• rosidl_typesupport_introspection_cpp
• rosidl_typesupport_introspection_tests
• rosidl_typesupport_tests
• rpyutils
• rqt
• rqt_action
• rqt_bag
• rqt_bag_plugins
• rqt_console
• rqt_graph
• rqt_gui
• rqt_gui_cpp
• rqt_gui_py
• rqt_msg
• rqt_plot
• rqt_publisher
• rqt_py_common
• rqt_py_console
• rqt_reconfigure
• rqt_service_caller
• rqt_shell
• rqt_srv
• rqt_topic
• rti_connext_dds_cmake_module
• rttest
• rviz2
• rviz_assimp_vendor
• rviz_common
• rviz_default_plugins
• rviz_ogre_vendor
• rviz_rendering
• rviz_rendering_tests
• rviz_visual_testing_framework
• sensor_msgs
• sensor_msgs_py
• service_msgs
• shape_msgs
• shared_queues_vendor
• spdlog_vendor
• sqlite3_vendor
• sros2
• statistics_msgs
• std_msgs
• std_srvs
• stereo_msgs
• tango_icons_vendor
• test_cli
• test_cli_remapping
• test_communication
• test_interface_files
• test_launch_ros
• test_launch_testing
• test_msgs
• test_osrf_testing_tools_cpp
• test_quality_of_service
• test_rclcpp
• test_rmw_implementation
• test_ros2trace
• test_security
• test_tf2
• test_tracetools
• test_tracetools_launch
• tf2
• tf2_bullet
• tf2_eigen
• tf2_eigen_kdl
• tf2_geometry_msgs
• tf2_kdl
• tf2_msgs
• tf2_py
• tf2_ros
• tf2_ros_py
• tf2_sensor_msgs
• tf2_tools
• tlsf
• tlsf_cpp
• topic_monitor
• topic_statistics_demo
• tracetools
• tracetools_launch
• tracetools_trace
• trajectory_msgs
• turtlesim
• type_description_interfaces
• unique_identifier_msgs
• urdf
• urdf_parser_plugin
• visualization_msgs
• yaml_cpp_vendor
• zstd_vendor

action_msgs

• Update common_interfaces to C++17. (#215) (#151)
• Add service_msgs package (#143)
• [rolling] Update maintainers - 2022-11-07 (#150)
• Depend on rosidl_core_generators for packages required by actions (#144)
• Contributors: Audrow Nash, Brian, Chris Lalancette, Jacob Perron

action_tutorials_cpp

• Change all ROS2 -> ROS 2. (#610)
• Update the demos to C++17. (#594)
• Add README’s for action_tutorials. (#576)
• [rolling] Update maintainers - 2022-11-07 (#589)
• Fix two small bugs in the fibonacci C++ tutorial. (#564)
• Contributors: Audrow Nash, Chris Lalancette, kagibson

action_tutorials_interfaces

• Change all ROS2 -> ROS 2. (#610)
• A couple more upgrades to C++17. (#609)
• Add README’s for action_tutorials. (#576)
• [rolling] Update maintainers - 2022-11-07 (#589)
• Remove action_msgs dependency (#580)
• Contributors: Audrow Nash, Chris Lalancette, Jacob Perron, kagibson
action_tutorials_py

- Add README's for action_tutorials. (#576)
- [rolling] Update maintainers - 2022-11-07 (#589)
- Contributors: Audrow Nash, kagibson

actionlib_msgs

- Update common_interfaces to C++17. (#215)
- [rolling] Update maintainers - 2022-11-07 (#210)
- Contributors: Audrow Nash, Chris Lalancette

ament_clang_format

- ament_clang_format: use open braces for enum definitions (#426)
- [rolling] Update maintainers - 2022-11-07 (#421)
- Update maintainers (#379)
- Contributors: Audrow Nash, james-rms, methylDragon

ament_clang_tidy

- [rolling] Update maintainers - 2022-11-07 (#421)
- Update maintainers (#379)
- recommend use of –mixin compile-commands (#371)
- Improve message and avoid missing new lines between reports from files (#373)
- Contributors: Audrow Nash, William Woodall, methylDragon

ament_cmake

- Contributors: Audrow Nash

ament_cmake_auto

- Support INTERFACE on ament_auto_add_library (#420)
- Fix ament_auto_add_gtest’s parameter passing (#421)
- [rolling] Update maintainers - 2022-11-07 (#411)
- Rolling: ament_cmake_auto should include dependencies as SYSTEM (#385)
- Contributors: Audrow Nash, Christopher Wecht, Joshua Whitley, Rin Iwai

4.8. ROS 2 Documentation
ament_cmake_clang_format

- [rolling] Update maintainers - 2022-11-07 (#421)
- Update maintainers (#379)
- Contributors: Audrow Nash, methylDragon

ament_cmake_clang_tidy

- [rolling] Update maintainers - 2022-11-07 (#421)
- Update maintainers (#379)
- Contributors: Audrow Nash, methylDragon

ament_cmake_copyright

- [rolling] Update maintainers - 2022-11-07 (#421)
- [ament_lint_auto] General file exclusion with AMENT_LINT_AUTO_FILE_EXCLUDE (#386)
- Update maintainers (#379)
- Contributors: Abrar Rahman Protyasha, Audrow Nash, methylDragon

ament_cmake_core

- ament_cmake_uninstall_target: Correct location of install_manifest.txt (#432)
- Use file(GENERATE OUTPUT) to create dsv files (#416) Using file(WRITE) and file(APPEND) causes the modification stamp of the file to be changed each time CMake configures, resulting in an ‘Installing’ message rather than an ‘Up-to-date’ message even though the file content is identical. Using file(GENERATE OUTPUT) updates the timestamp of the file only if the content changes.
- Warn when trying to symlink install an INTERFACE_LIBRARY (#417)
- Workaround to exclude Clion’s cmake folders from colcon test (#410) - Add AMENT_IGNORE to CMAKE_BINARY_DIR to avoid picking up cmake specific folders created by CLion in colcon build and colcon test commands
- if (NOT ${UNDEFINED_VAR}) gets evaluated to false, so change to if (NOT UNDEFINED_VAR) so it evaluates to true. (#407)
- Implement ament_add_default_options (#390)
- Contributors: Audrow Nash, Kenji Brameld, Michael Orlov, Scott K Logan, Shane Loretz, Silvio Traversaro, methylDragon
ament_cmake_cppcheck

- [rolling] Update maintainers - 2022-11-07 (#421)
- [ament_lint_auto] General file exclusion with AMENT_LINT_AUTO_FILE_EXCLUDE (#386)
- Update maintainers (#379)
- Contributors: Abrar Rahman Protyasha, Audrow Nash, methylDragon

ament_cmake_cpplint

- [rolling] Update maintainers - 2022-11-07 (#421)
- [ament_lint_auto] General file exclusion with AMENT_LINT_AUTO_FILE_EXCLUDE (#386)
- Update maintainers (#379)
- Contributors: Abrar Rahman Protyasha, Audrow Nash, methylDragon

ament_cmake_export_definitions

- Contributors: Audrow Nash

ament_cmake_export_dependencies

- Contributors: Audrow Nash

ament_cmake_export_include_directories

- Contributors: Audrow Nash

ament_cmake_export_interfaces

- Contributors: Audrow Nash
ament_cmake_export_libraries

- Contributors: Audrow Nash

ament_cmake_export_link_flags

- Contributors: Audrow Nash

ament_cmake_export_targets

- Support new target export template introduced with CMake 3.24 (#395)
- Fix the order in which Export.cmake files are included (#256)
- Contributors: Audrow Nash, Timo Röhling

ament_cmake_flake8

- Add flake8 linter ignore support (#424)
- [rolling] Update maintainers - 2022-11-07 (#421)
- Update maintainers (#379)
- Contributors: Abrar Rahman Protyasha, Audrow Nash, RFRIEDM-Trimble, methylDragon

ament_cmake_gen_version_h

- Changed version gte macro to make it MSVC compatible. Fix #433 (#434)
- Contributors: Audrow Nash, iquarobotics

ament_cmake_gmock

- Fix compiler warnings related to gtest/gmock (#408) * Suppress compiler warnings when building gmock definition of implicit copy constructor ... is deprecated because it has a user-declared copy assignment operator [-Wdeprecated-copy] * Declare gtest/gmock include dirs as SYSTEM PRIVATE for test targets
- Contributors: Audrow Nash, Robert Haschke
ament_cmake_google_benchmark

- Contributors: Audrow Nash

ament_cmake_gtest

- Fix compiler warnings related to gtest/gmock (#408) * Suppress compiler warnings when building gmock definition of implicit copy constructor ... is deprecated because it has a user-declared copy assignment operator [-Wdeprecated-copy] * Declare gtest/gmock include dirs as SYSTEM PRIVATE for test targets
- Contributors: Audrow Nash, Robert Haschke

ament_cmake_include_directories

- Contributors: Audrow Nash

ament_cmake_libraries

- Contributors: Audrow Nash

ament_cmake_lint_cmake

- [rolling] Update maintainers - 2022-11-07 (#421)
- Update maintainers (#379)
- Contributors: Audrow Nash, methylDragon

ament_cmake_mypy

- [rolling] Update maintainers - 2022-11-07 (#421)
- Update maintainers (#379)
- Contributors: Audrow Nash, methylDragon
ament_cmake_pclint

- [rolling] Update maintainers - 2022-11-07 (#421)
- Update maintainers (#379)
- Contributors: Audrow Nash, methylDragon

ament_cmake_pep257

- [rolling] Update maintainers - 2022-11-07 (#421)
- Update maintainers (#379)
- Contributors: Audrow Nash, methylDragon

ament_cmake_pycodestyle

- [rolling] Update maintainers - 2022-11-07 (#421)
- Update maintainers (#379)
- Contributors: Audrow Nash, methylDragon

ament_cmake_pyflakes

- [rolling] Update maintainers - 2022-11-07 (#421)
- Update maintainers (#379)
- Contributors: Audrow Nash, methylDragon

ament_cmake_pytest

- Fix test skipping logic for missing pytest module (#441)
- Add missing buildtool_depend on python3-pytest (#440)
- ament_cmake_pytest needs a buildtool_depend on ament_cmake_test. (#439)
- Fix pytest-cov version detection with pytest >=7.0.0 (#436)
- use the error handler replace to allow non-utf8 to be decoded (#381)
- Add NOCAPTURE option to ament_add_pytest_test (#393)
- Contributors: Audrow Nash, Chris Lalancette, Christophe Bedard, El Jawad Alaa, Jacob Perron, Scott K Logan
ament_cmake_python

- Support Debian-specific install dir for ament_cmake_python (#431)
- Document ament_cmake_python (#387)
- Contributors: Audrow Nash, Shane Loretz, Timo Röhling

ament_cmake_ros

- [rolling] Update maintainers - 2022-11-07 (#16)
- Update maintainers (#15)
- Contributors: Audrow Nash, methylDragon

ament_cmake_target_dependencies

- Contributors: Audrow Nash

ament_cmake_test

- use the error handler replace to allow non-utf8 to be decoded (#381)
- Contributors: Audrow Nash, El Jawad Alaa

ament_cmake_uncrustify

- [rolling] Update maintainers - 2022-11-07 (#421)
- [ament_lint_auto] General file exclusion with AMENT_LINT_AUTO_FILE_EXCLUDE (#386)
- Update maintainers (#379)
- Contributors: Abrar Rahman Protyasha, Audrow Nash, methylDragon

ament_cmake_vendor_package

- Fix the version number of ament_cmake_vendor_package.
- Add ament_cmake_vendor_package package (#429)
- Contributors: Chris Lalancette, Scott K Logan
ament_cmake_version

- Contributors: Audrow Nash

ament_cmake_xmllint

- [rolling] Update maintainers - 2022-11-07 (#421)
- Update maintainers (#379)
- Contributors: Audrow Nash, methylDragon

ament_copyright

- [rolling] Update maintainers - 2022-11-07 (#421)
- Support for matching license header within multiline comment block (#361)
- Improved licence matching (#358)
- Updated regex and adding test cases for copyright search (#363)
- Update maintainers (#379)
- Contributors: Audrow Nash, Will, methylDragon

ament_cppcheck

- [rolling] Update maintainers - 2022-11-07 (#421)
- Update maintainers (#379)
- Contributors: Audrow Nash, methylDragon

ament_cpplint

- [rolling] Update maintainers - 2022-11-07 (#421)
- [ament_cpplint] Process errors without linenums (#385)
- Update maintainers (#379)
- Consider files with ‘.hh’ extension as C++ headers (#374)
- Contributors: Abrar Rahman Protyasha, Audrow Nash, Jacob Perron, methylDragon
ament_flake8

• [rolling] Update maintainers - 2022-11-07 (#421)
• Fix exclude regression (#387)
• Update maintainers (#379)
• Contributors: Audrow Nash, methylDragon

ament_index_cpp

• [rolling] Update maintainers - 2022-11-07 (#89)
• Contributors: Audrow Nash

ament_index_python

• [rolling] Update maintainers - 2022-11-07 (#89)
• Contributors: Audrow Nash

ament_lint

• [rolling] Update maintainers - 2022-11-07 (#421)
• Update maintainers (#379)
• Contributors: Audrow Nash, methylDragon

ament_lint_auto

• Add flake8 linter ignore support (#424)
• [rolling] Update maintainers - 2022-11-07 (#421)
• [ament_lint_auto] General file exclusion with AMENT_LINT_AUTO_FILE_EXCLUDE (#386)
• Update maintainers (#379)
• Contributors: Abrar Rahman Protyasha, Audrow Nash, RFRIEDM-Trimble, methylDragon

ament_lint_cmake

• [rolling] Update maintainers - 2022-11-07 (#421)
• Update maintainers (#379)
• Contributors: Audrow Nash, methylDragon
ament_lint_common

- [rolling] Update maintainers - 2022-11-07 (#421)
- Update maintainers (#379)
- Contributors: Audrow Nash, methylDragon

ament_mypy

- [rolling] Update maintainers - 2022-11-07 (#421)
- Update maintainers (#379)
- Contributors: Audrow Nash, methylDragon

ament_package

- Add support for comment lines in dsv files (#139)
- [rolling] Update maintainers - 2022-11-07 (#138)
- Mirror rolling to master
- Remove unused isolated prefix level templates (#133)
- Contributors: Audrow Nash, Scott K Logan, Shane Loretz

ament_pclint

- [rolling] Update maintainers - 2022-11-07 (#421)
- Update maintainers (#379)
- Contributors: Audrow Nash, methylDragon

ament_pep257

- updating ref to pep257 docs (#433)
- Added underscore to ignore new pydocstyle item (#428)
- [rolling] Update maintainers - 2022-11-07 (#421)
- [ament_pep257][master] redirecting error prints to stderr (#390)
- Update maintainers (#379)
- Contributors: Audrow Nash, Christian Henkel, Cristóbal Arroyo, Mirco Colosi (CR/AAS3), methylDragon
ament_pycodestyle

- ament_pycodestyle - fix crash caused by reporting on ignored errors (#435)
- [rolling] Update maintainers - 2022-11-07 (#421)
- Update maintainers (#379)
- Contributors: Audrow Nash, Shane Loretz, methylDragon

ament_pyflakes

- [rolling] Update maintainers - 2022-11-07 (#421)
- Update maintainers (#379)
- Contributors: Audrow Nash, methylDragon

ament_uncrustify

- [rolling] Update maintainers - 2022-11-07 (#421)
- Update maintainers (#379)
- Contributors: Audrow Nash, methylDragon

ament_xmlint

- [rolling] Update maintainers - 2022-11-07 (#421)
- Update maintainers (#379)
- Contributors: Audrow Nash, methylDragon

builtin_interfaces

- Update common_interfaces to C++17. (#215) (#151)
- [rolling] Update maintainers - 2022-11-07 (#150)
- Depend on rosidl_core_generators for packages required by actions (#144)
- Fix documented range (#139)
- Contributors: Audrow Nash, Chris Lalancette, Jacob Perron, Tully Foote
camera_calibration_parsers

- Update image_common to C++17. (#267)
- Add alias library targets for all libraries (#259)
- Add support for missing ROI and binning fields (#254)
- Contributors: AndreasR30, Chris Lalancette, RFRIEDM-Trimble

camera_info_manager

- Update image_common to C++17. (#267)
- Add alias library targets for all libraries (#259)
- Add lifecycle node compatibility to camera_info_manager (#190)
- Contributors: Chris Lalancette, RFRIEDM-Trimble, Ramon Wijnands

class_loader

- make sanitizer happy (#205)
- [rolling] Update maintainers - 2022-11-07 (#206)
- Remove appveyor configuration. (#204)
- Just fix a typo in a comment. (#203)
- make the meta-object alive in the lifecycle of the registered plugin (#201)
- Mirror rolling to ros2
- Contributors: Audrow Nash, Chen Lihui, Chris Lalancette

common_interfaces

- [rolling] Update maintainers - 2022-11-07 (#210)
- Contributors: Audrow Nash

composition

- Change all ROS2 -> ROS 2. (#610)
- update launch file name format to match documentation (#588)
- Added README.md for composition (#598)
- Update the demos to C++17. (#594)
- [rolling] Update maintainers - 2022-11-07 (#589)
- fix memory leak (#585)
- Contributors: Audrow Nash, Chen Lihui, Chris Lalancette, Gary Bey, Patrick Wspanialy
composition_interfaces

- Update common_interfaces to C++17. (#215) (#151)
- [rolling] Update maintainers - 2022-11-07 (#150)
- Contributors: Audrow Nash, Chris Lalancette

demo_nodes_cpp

- Change all ROS2 -> ROS 2. (#610)
- Add matched event demo for rclcpp and rclpy (#607)
- Fix the set_parameters_callback example program. (#608)
- [demo_nodes_cpp] Add YAML launch demos for topics (#605)
- update launch file name format to match documentation (#588)
- Service introspection (#602) * Add in a rclcpp and rclpy demo of introspection.
- Added README.md for demo_cpp_nodes (#599)
- Update the demos to C++17. (#594)
- [rolling] Update maintainers - 2022-11-07 (#589)
- Demo for pre and post set parameter callback support (#565) * local parameter callback support
- counter starts from 1, not 2. (#562)
- add a demo of content filter listener (#557)
- Contributors: Audrow Nash, Barry Xu, Chen Lihui, Chris Lalancette, Damien LaRocque, Deepanshu Bansal, Gary Bey, Patrick Wspanialy, Tomoya Fujita

demo_nodes_cpp_native

- Change all ROS2 -> ROS 2. (#610)
- Added README.md for demo_cpp_nodes_native (#597)
- Update the demos to C++17. (#594)
- [rolling] Update maintainers - 2022-11-07 (#589)
- Make demo_nodes_cpp_native install stuff only when it builds (#590)
- Contributors: Audrow Nash, Chris Lalancette, Gary Bey, Shane Loretz
demo_nodes_py

- Change all ROS2 -> ROS 2. (#610)
- Add matched event demo for rclcpp and rclpy (#607)
- Enable document generation using rosdoc2 (#606)
- Service introspection (#602)
- Added README.md for demo_nodes_py (#600)
- [rolling] Update maintainers - 2022-11-07 (#589)
- Demo for pre and post set parameter callback support (#565)
- Add demo for rclpy parameter client (#566)
- Exit with code 0 if ExternalShutdownException is raised (#581)
- Contributors: Audrow Nash, Barry Xu, Brian, Chris Lalancette, Deepanshu Bansal, Gary Bey, Jacob Perron, Yadu

diagnostic_msgs

- Update common_interfaces to C++17. (#215)
- [rolling] Update maintainers - 2022-11-07 (#210)
- Contributors: Audrow Nash, Chris Lalancette

domain_coordinator

- [rolling] Update maintainers - 2022-11-07 (#16)
- Update maintainers (#15)
- Contributors: Audrow Nash, methylDragon

dummy_map_server

- Change all ROS2 -> ROS 2. (#610)
- Update the demos to C++17. (#594)
- [rolling] Update maintainers - 2022-11-07 (#589)
- Added README.md for dummy_map_server (#572)
- Contributors: Audrow Nash, Chris Lalancette, Gary Bey
**dummy_robotbringup**

- update launch file name format to match documentation (#588)
- [rolling] Update maintainers - 2022-11-07 (#589)
- Contributors: Audrow Nash, Patrick Wspanialy

**dummy_sensors**

- Fix unstable LaserScan status for rviz2 (#616)
- Added README.md for dummy_sensors (#573)
- Update the demos to C++17. (#594)
- [rolling] Update maintainers - 2022-11-07 (#589)
- Contributors: Audrow Nash, Chen Lihui, Chris Lalancette, Gary Bey

**eigen3_cmake_module**

- [rolling] Update maintainers - 2022-11-07 (#6)
- Mirror rolling to master
- Update maintainers (#4)
- Contributors: Audrow Nash, methylDragon

**example_interfaces**

- [rolling] Update maintainers - 2022-11-07 (#17)
- Remove action_msgs dependency (#16)
- Mirror rolling to master
- Contributors: Audrow Nash, Jacob Perron

**examples_rclcpp_async_client**

- Update the examples to C++17. (#353)
- [rolling] Update maintainers - 2022-11-07 (#352)
- Contributors: Audrow Nash, Chris Lalancette
examples_rclcpp_cbg_executor

- Update the examples to C++17. (#353)
- [rolling] Update maintainers - 2022-11-07 (#352)
- Contributors: Audrow Nash, Chris Lalancette

examples_rclcpp_minimal_action_client

- Update the examples to C++17. (#353)
- [rolling] Update maintainers - 2022-11-07 (#352)
- Contributors: Audrow Nash, Chris Lalancette

examples_rclcpp_minimal_action_server

- Update the examples to C++17. (#353)
- [rolling] Update maintainers - 2022-11-07 (#352)
- Contributors: Audrow Nash, Chris Lalancette

examples_rclcpp_minimal_client

- Update the examples to C++17. (#353)
- [rolling] Update maintainers - 2022-11-07 (#352)
- Contributors: Audrow Nash, Chris Lalancette

examples_rclcpp_minimal_composition

- Update the examples to C++17. (#353)
- [rolling] Update maintainers - 2022-11-07 (#352)
- Contributors: Audrow Nash, Chris Lalancette

examples_rclcpp_minimal_publisher

- Update the examples to C++17. (#353)
- [rolling] Update maintainers - 2022-11-07 (#352)
- Contributors: Audrow Nash, Chris Lalancette
examples_rclcpp_minimal_service

- Update the examples to C++17. (#353)
- [rolling] Update maintainers - 2022-11-07 (#352)
- Contributors: Audrow Nash, Chris Lalancette

examples_rclcpp_minimal_subscriber

- Update the examples to C++17. (#353)
- [rolling] Update maintainers - 2022-11-07 (#352)
- add ContentFilteredTopic example. (#341)
- Contributors: Audrow Nash, Chris Lalancette, Tomoya Fujita

examples_rclcpp_minimal_timer

- Update the examples to C++17. (#353)
- [rolling] Update maintainers - 2022-11-07 (#352)
- Contributors: Audrow Nash, Chris Lalancette

examples_rclcpp_multithreaded_executor

- Update the examples to C++17. (#353)
- [rolling] Update maintainers - 2022-11-07 (#352)
- Contributors: Audrow Nash, Chris Lalancette

examples_rclcpp_wait_set

- Update the examples to C++17. (#353)
- [rolling] Update maintainers - 2022-11-07 (#352)
- Add test linting to wait_set and fix issues. (#346) (#347)
- Contributors: Audrow Nash, Chris Lalancette, mergify[bot]

examples_rclpy_executors

- [rolling] Update maintainers - 2022-11-07 (#352)
- Contributors: Audrow Nash
examples_rclpy_guard_conditions

- [rolling] Update maintainers - 2022-11-07 (#352)
- Contributors: Audrow Nash

examples_rclpy_minimal_action_client

- Enable document generation using rosdoc2 for ament_python pkgs (#357) * Add missing action_msgs dep*
- Add exec_deps for launch_testing_examples
- [rolling] Update maintainers - 2022-11-07 (#352)
- Contributors: Audrow Nash, Yadu

examples_rclpy_minimal_action_server

- [rolling] Update maintainers - 2022-11-07 (#352)
- Contributors: Audrow Nash

examples_rclpy_minimal_client

- [rolling] Update maintainers - 2022-11-07 (#352)
- Contributors: Audrow Nash

examples_rclpy_minimal_publisher

- [rolling] Update maintainers - 2022-11-07 (#352)
- Contributors: Audrow Nash

examples_rclpy_minimal_service

- [rolling] Update maintainers - 2022-11-07 (#352)
- Contributors: Audrow Nash

examples_rclpy_minimal_subscriber

- [rolling] Update maintainers - 2022-11-07 (#352)
- Contributors: Audrow Nash
examples_rclpy_pointcloud_publisher

- [rolling] Update maintainers - 2022-11-07 (#352)
- Contributors: Audrow Nash

examples_tf2_py

- Enable document generation using rosdoc2 for ament_python pkgs (#587)
- Update maintainers (#560)
- Contributors: Audrow Nash, Yadu

fastrtps_cmake_module

- [rolling] Update maintainers - 2022-11-07 (#93)
- Contributors: Audrow Nash

foonathan_memory_vendor

- Added support for QNX 7.1 build (#65)
- Update upstream to release 0.7-3 (#62)(#63)
- Fix CMake minimum required version (#60)

geometry2

- Update maintainers (#560)
- Contributors: Audrow Nash

geometry_msgs

- Update common_interfaces to C++17. (#215)
- [rolling] Update maintainers - 2022-11-07 (#210)
- Contributors: Audrow Nash, Chris Lalancette

google_benchmark_vendor

- Actually update to 1.6.1. (#25) We claimed we were, but in fact we were pinned to the 1.5.3 git hash.
- Remove set but unused variable (#24) Clang checks -Wunused-but-set-variable. This fails the build with -Werror also enabled.
- [rolling] Update maintainers - 2022-11-07 (#22)
- Mirror rolling to main
- Contributors: Audrow Nash, Chris Lalancette, Michael Carroll
ignition_cmake2_vendor

- Set target version to 2.14.0 (#5)
- Mirror rolling to main
- Contributors: Audrow Nash, Yadu

ignition_math6_vendor

- Forward CMAKE_PREFIX_PATH when building vendor package (#8)
- Contributors: Scott K Logan

image_tools

- Added README.md for image_tools - [Clean] (#596)
- Update the demos to C++17. (#594)
- [rolling] Update maintainers - 2022-11-07 (#589)
- Contributors: Audrow Nash, Chris Lalancette, Gary Bey

image_transport

- Update image_common to C++17. (#267)
- Add alias library targets for all libraries (#259)
- Remove subscriber and publisher impl methods without options (#252)
- Deprecate impl without options (#249)
- opt-in to qos overriding for publisher (#246)
- Add qos option to override qos (#208)
- Contributors: Brian, Chris Lalancette, Daisuke Nishimatsu, Kenji Brameld, RFRIEDM-Trimble

interactive_markers

- Update interactive_markers to C++17. (#99)
- Update maintainers (#98)
- Mirror rolling to ros2
- update maintainer (#92)
- Contributors: Audrow Nash, Chris Lalancette, Dharini Dutia
intra_process_demo

- Fix executable name in README (#619)
- Change all ROS2 -> ROS 2. (#610)
- Added README.md for intra_process_demo (#595)
- Update the demos to C++17. (#594)
- [rolling] Update maintainers - 2022-11-07 (#589)
- Contributors: Audrow Nash, Chris Lalancette, Gary Bey, Yadunund

kdl_parser

- Switch some tests to use unique pointers instead of raw pointers. (#74)
- log link children as DEBUG instead of INFO (#71)
- Enable the kdl_parser tests in ROS 2 (#68)
- Add in a LICENSE file and fix up copyright headers (#66)
- Use orocos_kdl_vendor and orocos-kdl target (#64)
- Use the rcutils logger instead of printf (#65)
- Contributors: Chris Lalancette, Joseph Schornak, Scott K Logan, yuraSomatic

keyboard_handler

- Force exit from main thread on signal handling in keyboard_handler (#23)
- Contributors: Michael Orlov

laser_geometry

- Update laser_geometry to C++17. (#90)
- Update Maintainers (#88)
- Mirror rolling to ros2
- Contributors: Audrow Nash, Chris Lalancette

launch

- Document LaunchService.{run,run_async}() return value (#702)
- [rosdoc2] Fix document generation on buildfarm (#701)
- Enable document generation using rosdog2 for ament_python pkgs (#697)
- Remove the import of Literal from entity.py. (#694)
- Fix flake8 errors. (#695)
- add symlink to latest log directory (#686)
- Improve type checking (#679)
- Fixed typos (#692)
- Pass modules to PythonExpression (#655)
- Allow ReadyToTest() usage in event handler (#665)
- Expose emulate_tty to xml and yaml launch (#669)
- Expose sigterm_timeout and sigkill_timeout to xml frontend (#667)
- [rolling] Update maintainers - 2022-11-07 (#671)
- Expect deprecation warnings in tests (#657)
- Fix the restoring of os.environ to maintain type. (#656)
- Implement Any, All, Equals, and NotEquals substitutions (#649)
- add LaunchLogDir substitution, replacing log_dir frontend only substitution (#652)
- Add special cases to coerce “1” and “0” to bool when using bool coercion only (#651)
- Update launch/test/launch/test_execute_local.py
- Added unit test ensuring that output dictionary works with ExecuteLocal
- Addresses issue #588, allowing dict for ‘output’
- Remove unused variables. (#612)
- Expose shutdown action to xml frontend (#611)
- Contributors: Aditya Pande, Alejandro Hernández Cordero, Audrow Nash, Blake Anderson, Chris Lalancette, Christophe Bedard, Hervé Audren, Jacob Perron, Matthew Elwin, Michael Jeronimo, Nikolai Morin, Welte, William Woodall, Yadu, methylDragon

**launch_pytest**

- Fixed typos (#692)
- Drop unused data_files entry for example_processes (#680)
- Spelling correction
- [rolling] Update maintainers - 2022-11-07 (#671)
- Contributors: Alejandro Hernández Cordero, Audrow Nash, Geoffrey Biggs, Scott K Logan

**launch_ros**

- Use SomeEntitiesType for type checking. (#358)
- Fix normalize_parameters_dict for multiple nodes in the same namespace (#347)
- Implement None check for ComposableNodeContainer (#341)
- Add LifecyleTransition action (#317)
- Improve evaluate parameter_dict exceptions error message (#320)
- Ensure load_composable_nodes respects condition (#339)
- fix: return text value to avoid exception (#338)
• [rolling] Update maintainers - 2022-11-07 (#331)
• RosTimer -> ROSTimer and PushRosNamespace -> PushROSNamespace, to follow PEP8 (#326)
• add SetROSLogDir action (#325)
• Support default values in parameter substitution (#313)
• Run condition for composable nodes (#311)
• Contributors: Aditya Pande, Alexey Merzlyakov, Audrow Nash, Chris Lalancette, Christoph Hellmann Santos, Daisuke Nishimatsu, Felipe Gomes de Melo, Kenji Miyake, William Woodall, methylDragon

launch_testing

• Improve type checking (#679)
• Fixed typos (#692)
• Allow ReadyToTest() usage in event handler (#665)
• Inherit markers from generate_test_description (#670)
• [rolling] Update maintainers - 2022-11-07 (#671)
• Fix Typo (#641)
• ReadyToTest action timeout using decorator (#625)
• Switch to using a comprehension for process_names. (#614)
• Contributors: Alejandro Hernández Cordero, Audrow Nash, Chris Lalancette, Deepanshu Bansal, Hervé Audren, Kenji Brameld, Nikolai Morin, Scott K Logan

launch_testing_ament_cmake

• [rolling] Update maintainers - 2022-11-07 (#671)
• Contributors: Audrow Nash

launch_testing_examples

• Enable document generation using rosdoc2 for ament_python pkgs (#357)
• increase the timeout for window platform to avoid flaky test (#355)
• [rolling] Update maintainers - 2022-11-07 (#352)
• Increase the WaitForNode timeout. (#350)
• Contributors: Audrow Nash, Chen Lihui, Chris Lalancette, Yadu
launch_testing_ros

- Increase the timeouts in wait_for_topic_launch_test. (#360)
- Enable document generation using rosdoc2 (#359)
- exit() methods should not reraise the passed-in exception (#357)
- Inherit markers from generate_test_description (#330)
- [rolling] Update maintainers - 2022-11-07 (#331)
- Fix long wait during shutdown in WaitForTopics (#314)
- Contributors: Audrow Nash, Chris Lalancette, Giorgio Pintaudi, Keng12, Scott K Logan, Yadu

launch_xml

- Fixed typos (#692)
- Expose emulate_tty to xml and yaml launch (#669)
- Expose sigterm_timeout and sigkill_timeout to xml frontend (#667)
- [rolling] Update maintainers - 2022-11-07 (#671)
- Contributors: Aditya Pande, Alejandro Hernández Cordero, Audrow Nash

launch_yaml

- Expose emulate_tty to xml and yaml launch (#669)
- Expose sigterm_timeout and sigkill_timeout to xml frontend (#667)
- [rolling] Update maintainers - 2022-11-07 (#671)
- Contributors: Aditya Pande, Audrow Nash

libcurl_vendor

- merge libcurl_vendor build instructions (#81)
- Sets CMP0135 policy behavior to NEW (#79)
- Fixes policy CMP0135 warning for CMake >= 3.24
- Contributors: Cristóbal Arroyo, Crola1702, schrodinbug

libstatistics_collector

- Bump hmarr/auto-approve-action from 3.2.0 to 3.2.1
- Mark benchmark _ as unused. (#158)
- Bump hmarr/auto-approve-action from 3.1.0 to 3.2.0
- Bump ros-tooling/action-ros-ci from 0.2 to 0.3
- Bump pascalgn/automerge-action from 0.15.5 to 0.15.6
- Update libstatistics_collector to C++17. (#154)
- Remove unnecessary build dependency on std_msgs. (#145)
- Bump pascalgn/automerge-action from 0.15.2 to 0.15.3
- Cleanup the CI jobs on this repository. (#146)
- Check if message has a “header” field with a stamp subfield of type builtin_interfaces::msg::Time (#54)
- Mirror rolling to master
- Contributors: Audrow Nash, Chris Lalancette, Scott Mende, dependabot[bot]

**libyaml_vendor**

- Fix system package dependency (#54)
- Update libyaml_vendor to C++17. (#55)
- [rolling] Update maintainers - 2022-11-07 (#53)
- Remove a warning message. (#51)
- check if libyaml is already present before building it (take 2) (#45)
- Mirror rolling to master
- Support WindowsStore builds for ROS2 (#50) * libyaml for uwp
- Contributors: Audrow Nash, Chris Lalancette, Lou Amadio, Scott K Logan, Silvio Traversaro

**lifecycle**

- update launch file name format to match documentation (#588)
- Update the demos to C++17. (#594)
- [rolling] Update maintainers - 2022-11-07 (#589)
- Contributors: Audrow Nash, Chris Lalancette, Patrick Wspanialy

**lifecycle_msgs**

- Update common_interfaces to C++17. (#215) (#151)
- [rolling] Update maintainers - 2022-11-07 (#150)
- lifecycle_msgs: remove non-ASCII chars from field comments (#147)
- Contributors: Audrow Nash, Chris Lalancette, G.A. vd. Hoorn
**lifecycle_py**

- Enable document generation using rosdoc2 (#606)
- update launch file name format to match documentation (#588)
- Cleanup lifecycle_py to conform to ROS 2 standards. (#604)
- [rolling] Update maintainers - 2022-11-07 (#589)
- Install the launch file for lifecycle_py. (#586)
- Contributors: Audrow Nash, Chris Lalancette, Patrick Wspanialy, Yadu

**logging_demo**

- Update the demos to C++17. (#594)
- [rolling] Update maintainers - 2022-11-07 (#589)
- Change dependency from ‘rosidl_cmake’ to ‘rosidl_default_generators’ (#578)
- Contributors: Audrow Nash, Chris Lalancette, Jacob Perron

**map_msgs**

- Update maintainers
- Contributors: Audrow Nash, Steve Macenski

**mcap_vendor**

- mcap_vendor: add readme with versioning procedure (#1230)
- Add Michael Orlov as maintainer in rosbag2 packages (#1215)
- mcap_vendor: only install public headers (#1207)
- Fixes policy CMP0135 warning for CMake >= 3.24 for mcap_vendor (#1208)
- mcap_vendor: download MCAP source via tarball (#1204)
- rosbag2_cpp: test more than one storage plugin (#1196)
- rosbag2_storage_mcap: merge into rosbag2 repo (#1163)
- Fix Windows build (#73) Update mcap version to newest windows-compatible release. Add visibility macros for tests. Add clang-format preprocessor indentation for visibility_control to be readable.
- mcap_vendor: update to v0.6.0 (#69)
- Cleanup in mcap_vendor package (#62)
- Switch to using the vendored zstd library. (#59)
- Support timestamp-ordered playback (#50)
- Support regex topic filtering
- Add all lz4 sources to fix undefined symbols at runtime (#46)
• Upgrade mcap to fix LZ4 error and segfault (#42) Incorporates fixes from https://github.com/foxglove/mcap/pull/478 and https://github.com/foxglove/mcap/pull/482

• Add missing buildtool_depend on git (#37) This vendor package uses git to fetch sources for other packages. It should declare a dependency on that build tool. This should address the current cause of RPM build failures for RHEL: https://build.ros2.org/view/Rbin_rhel_el864/job/Rbin_rhel_el864__mcap_vendor__rhel_8_x86_64__binary/

• Test Foxy & Galactic in CI, fix missing test_depends in mcap_vendor/package.xml (#33)

• fix: minor issues (#31) * remove unnecessary block * use target_link_libraries instead of ament_target_dependencies * remove ros environment * add prefix to compile definition

• Update email address for Foxglove maintainers (#32)

• Added mcap_vendor package. Updated CMakeLists.txt to fetch dependencies with FetchContent rather than Conan.

• Contributors: Chris Lalancette, Cristóbal Arroyo, Daisuke Nishimatsu, Emerson Knapp, Jacob Bandes-Storch, James Smith, Michael Orlov, Scott K Logan, james-rms

message_filters

• Update message_filters to C++17. (#88)

• Fix cache.h std::placeholder namespace (#87)

• [rolling] Update maintainers - 2022-11-07 (#85)

• Add a simpler approximate time sync policy: ApproximateEpsilonTime (#84)

• Add latest time zero-order-hold sync policy (#73)

• Fix python examples and add a new example in documentation (#79)

• Mirror rolling to master

• Adding fix to subscribe() call with raw node pointer and subscriber options (#76)

• Corrected function arguments in example description (#35)

• Changed invocation to add to conform template syntax (#1388)

• fix sphinx warning (#1371)

• change invocation to add to conform template syntax (#1388)

• fix sphinx warning (#1371)

• Contributors: Audrow Nash, Carlos Andrés Álvarez Restrepo, Chris Lalancette, Haoru Xue, Ivan Santiago Paunovic, Martin Ganeff, Steve Macenski, andermi
mimick_vendor

- [rolling] Update maintainers - 2022-11-07 (#29)
- Mirror rolling to master
- Contributors: Audrow Nash

nav_msgs

- Update common_interfaces to C++17. (#215)
- [rolling] Update maintainers - 2022-11-07 (#210)
- Contributors: Audrow Nash, Chris Lalancette

orocos_kdl_vendor

- Make sure to quote orocos variables when setting targets. (#12)
- Ensure orocos-kdl is available as a target (#10)
- Ensure orocos-kdl target references Eigen (#6)
- Contributors: Chris Lalancette, Scott K Logan

osrf_pycommon

- [master] Update maintainers - 2022-11-07 (#89)
- Declare test dependencies in [test] extra (#86)
- Contributors: Audrow Nash, Scott K Logan

osrf_testing_tools_cpp

- Fix mpark/variant conditional for MSVC (#77)
- Changing C++ Compile Version (#76)
- Update maintainers (#74)
- Sets CMP0135 policy behavior to NEW (#73)
- Fixes policy CMP0135 warning in CMake 3.24 (#71)
- Add cstring include. (#70)
- Contributors: Audrow Nash, Chris Lalancette, Cristóbal Arroyo, Lucas Wendland, Scott K Logan
pendulum_control

- Update the demos to C++17. (#594)
- [rolling] Update maintainers - 2022-11-07 (#589)
- Contributors: Audrow Nash, Chris Lalancette

pendulum_msgs

- Change all ROS2 -> ROS 2. (#610)
- A couple more upgrades to C++17. (#609)
- Added README.md for pendulum_msgs. (#577)
- [rolling] Update maintainers - 2022-11-07 (#589)
- Contributors: Audrow Nash, Chris Lalancette, Gary Bey

performance_test_fixture

- Resolve use-after-free compiler warnings (#24)
- Update performance_test_fixture to C++17. (#21)
- [rolling] Update maintainers - 2022-11-07 (#20)
- Mirror rolling to main
- Add “cstring” to the list of includes (#19)
- Contributors: Audrow Nash, Chris Lalancette, Scott K Logan

pluginlib

- Update maintainers
- Contributors: Audrow Nash

pybind11_vendor

- Add a modified patch from upstream to support Python 3.11 (#22)
- Add missing buildtool dependency on git (#19)
- Update maintainers (#17)
- Force pybind11 to find Python 3. (#15)
- Mirror rolling to master
- Update maintainers (#14)
- Update to pybind11 2.9.1.
- Rename patch file for history continuity.
- Contributors: Audrow Nash, Chris Lalancette, Scott K Logan, Steven! Ragnarök, methylDragon
python_cmake_module

- [rolling] Update maintainers - 2022-11-07 (#13)
- Mirror rolling to master
- Contributors: Audrow Nash

python_orocos_kdl_vendor

- Fixes policy CMP0135 warning for CMake >= 3.24 (#16)
- Workaround pybind11 CMake error (#9)
- Contributors: Cristóbal Arroyo, Jacob Perron

python_qt_binding

- Fix to allow ninja to use make for generators (#123)
- Fix flake8 linter regression (#125)
- Remove pyqt from default binding order for macOS (#118)
- Demote missing SIP message from WARNING to STATUS (#122)
- [rolling] Update maintainers - 2022-11-07 (#120)
- Contributors: Audrow Nash, Christoph Hellmann Santos, Cristóbal Arroyo, Michael Carroll, Rhys Mainwaring, Scott K Logan

qt_dotgraph

- Add in LICENSE file
- Cast drawLine input arguments to int (#264) (#265)
- Contributors: Chris Lalancette, mergify[bot]

qt_gui

- Add in LICENSE file
- Fix flake8 errors introduced by the previous commit. (#262)
- Enable basic help information if no plugins are running (#261)
- Contributors: Chris Lalancette, Michael Jeronimo
qt_gui_app

- Add in LICENSE file
  - Contributors: Chris Lalancette

qt_gui_core

- Add in LICENSE file
  - Contributors: Chris Lalancette

qt_gui_cpp

- Fix ClassLoader warning by unloading plugin providers. (#275)
- Chen Lihui
- fix shiboken error (#267)
- Conditionally run import tests when generators are built (#269)
- Add in LICENSE file
  - Contributors: Chris Lalancette, Christoph Hellmann Santos, Michael Carroll, Rhys Mainwaring, Scott K Logan

qt_gui_py_common

- Add in LICENSE file
  - Contributors: Chris Lalancette

quality_of_service_demo_cpp

- Update the demos to C++17. (#594)
- [rolling] Update maintainers - 2022-11-07 (#589)
  - Contributors: Audrow Nash, Chris Lalancette

quality_of_service_demo_py

- Use non-deprecated rclpy import. (#617)
- Change all ROS2 -> ROS 2. (#610)
- Enable document generation using rosdoc2 (#606)
- [rolling] Update maintainers - 2022-11-07 (#589)
- Exit with code 0 if ExternalShutdownException is raised (#581)
  - Contributors: Audrow Nash, Chris Lalancette, Jacob Perron, Yadu
rcl

- Honor ROS_LOCALHOST_ONLY if enabled. (#1071)
- fix flaky test (#1063)
- Add enable_type_description_service node option - API only (#1060)
- Dynamic Subscription (BONUS: Allocators): rcl (#1057)
- Runtime Interface Reflection: rcl (#1025)
- [rcl] Improve handling of dynamic discovery (#1023)
- Use get_type_hash_func for typesupports (#1055)
- publish for rosout topic multiple times to avoid flaky test (#1054)
- Switch to target_link_libraries in rcl. (#1051)
- Calculate type hash from TypeDescription (rep2011) (#1027)
- Implement matched event (#1033)
- use user-defined allocator to configure logging. (#1047)
- user defined allocator should be used for rosout publisher. (#1044)
- Add in inconsistent_topic implementation. (#1024)
- doc update, ROS message accessibility depends on RMW implementation. (#1043)
- Fix some warnings from clang. (#1042)
- avoid unnecessary copy for rcutils_char_array_vsprintf. (#1035)
- Service introspection (#997)
- Cache disable flag to avoid reading environmental variable. (#1029)
- use parent logger (#921)
- Add timer on reset callback (#995)
- Update rcl to C++17. (#1031)
- Make sure to check the return value of rcl_clock_init in tests. (#1030)
- Implement rcl_clock_time_started (#1021)
- Make sure to reset errors more places in the tests. (#1020) This makes it so we don’t get as many warnings when the tests are running.
- [rolling] Update maintainers - 2022-11-07 (#1017)
- Small cleanups to rcl (#1013)
- use int64_t for period. (#1010)
- fixed rcl_wait return error when timer cancelled (#1003)
- remove duplicate packages in find_package and reorder (#994)
- Fix buffer overflow in argument parsing caused by lexer returning length beyond length of string (#979)
- Fix leak in test_subscription_content_filter_options.cpp (#978)
- Contributors: Audrow Nash, Barry Xu, Brian, Chen Lihui, Chris Lalancette, Emerson Knapp, Geoffrey Biggs, Shane Loretz, Tomoya Fujita, mauropasse, methylDragon,
rcl_action

- doc update, ROS message accessibility depends on RMW implementation. (#1043)
- Update rcl to C++17. (#1031)
- Reduce result_timeout to 10 seconds. (#1012)
- [rolling] Update maintainers - 2022-11-07 (#1017)
- Contributors: Audrow Nash, Chris Lalancette, Tomoya Fujita

rcl_interfaces

- Add interfaces for logging service. (#154)
- Update common_interfaces to C++17. (#215) (#151)
- [rolling] Update maintainers - 2022-11-07 (#150)
- Contributors: Audrow Nash, Chris Lalancette, Lei Liu

rcl_lifecycle

- Update rcl to C++17. (#1031)
- [rolling] Update maintainers - 2022-11-07 (#1017)
- Contributors: Audrow Nash, Chris Lalancette

rcl_logging_interface

- Update rcl_logging to C++17. (#98)
- Updated maintainers - 2022-11-07 (#96)
- Contributors: Audrow Nash, Chris Lalancette

rcl_logging_noop

- Update rcl_logging to C++17. (#98)
- Updated maintainers - 2022-11-07 (#96)
- Contributors: Audrow Nash, Chris Lalancette
rcl_logging_spdlog

- Mark the benchmark `_` as unused. (#99)
- Update rcl_logging to C++17. (#98)
- change flushing behavior for spdlog log files, and add env var to use old style (no explicit flush- ing) (#95) * now flushes every ERROR message and periodically every 5 seconds * can set RCL_LOGGING_SPDLOG_EXPERIMENTAL_OLD_FLUSHING_BEHAVIOR=1 to get old behavior
- Updated maintainers - 2022-11-07 (#96)
- Disable cppcheck for rcl_logging_spdlog. (#93)
- ament_export_dependencies any package with targets we linked against (#89)
- Contributors: Audrow Nash, Chris Lalancette, Shane Loretz, William Woodall

rcl_yaml_param_parser

- Fix some warnings from clang. (#1042)
- Cleanup the dependencies in rcl_yaml_param_parser. (#1014)
- Update rcl to C++17. (#1031)
- Support yaml string tag ‘!!str’ (#999)
- [rolling] Update maintainers - 2022-11-07 (#1017)
- Contributors: Audrow Nash, Barry Xu, Chris Lalancette

rclcpp

- Fix delivered message kind (#2175) (#2178)
- Add support for logging service. (#2122)
- Picking ABI-incompatible executor changes (#2170)
- add events-executor and timers-manager in rclcpp (#2155)
- Create common structures for executors to use (#2143)
- Implement deliver message kind (#2168)
- applied tracepoints for ring_buffer (#2091)
- Dynamic Subscription (REP-2011 Subset): Stubs for rclcpp (#2165)
- Add type_hash to cpp TopicEndpointInfo (#2137)
- Trigger the intraprocess guard condition with data (#2164)
- Minor grammar fix (#2149)
- Fix unnecessary allocations in executor.cpp (#2135)
- add Logger::get_effective_level(). (#2141)
- Remove deprecated header (#2139)
- Implement matched event (#2105)
- use allocator via init_options argument. (#2129)
- Fixes to silence some clang warnings. (#2127)
- Documentation improvements on the executor (#2125)
- Avoid losing waitable handles while using MultiThreadedExecutor (#2109)
- Hook up the incompatible type event inside of rclcpp (#2069)
- Update all rclcpp packages to C++17. (#2121)
- Fix clang warning: bugprone-use-after-move (#2116)
- Fix memory leak in tracetools::get_symbol() (#2104)
- Service introspection (#1985)
- Allow publishing borrowed messages with intra-process enabled (#2108)
- to fix flaky test about TestTimeSource.callbacks (#2111)
- to create a sublogger while getting child of Logger (#1717)
- Fix documentation of Context class (#2107)
- fixes for rmw callbacks in qos_event class (#2102)
- Add support for timers on reset callback (#1979)
- Topic node guard condition in executor (#2074)
- Fix bug on the disorder of calling shutdown callback (#2097)
- Add default constructor to NodeInterfaces (#2094)
- Fix clock state cached time to be a copy, not a reference. (#2092)
- Fix -Wmaybe-uninitialized warning (#2081)
- Fix the keep_last warning when using system defaults. (#2082)
- Add in a fix for older compilers. (#2075)
- Implement Unified Node Interface (NodeInterfaces class) (#2041)
- Do not throw exception if trying to dequeue an empty intra-process buffer (#2061)
- Move event callback binding to PublisherBase and SubscriptionBase (#2066)
- Implement validity checks for rclcpp::Clock (#2040)
- Explicitly set callback type (#2059)
- Fix logging macros to build with msvc and cpp20 (#2063)
- Add clock type to node_options (#1982)
- Fix nullptr dereference in prune_requests_older_than (#2008)
- Remove templating on to_rcl_subscription_options (#2056)
- Fix SharedFuture from async_send_request never becoming valid (#2044)
- Add in a warning for a KeepLast depth of 0. (#2048)
- Mark rclcpp::Clock::now() as const (#2050)
- Fix a case that did not throw ParameterUninitializedException (#2036)
- Update maintainers (#2043)
- MultiThreadExecutor number of threads is at least 2+ in default. (#2032)
• Fix bug that a callback not reached (#1640)
• Set the minimum number of threads of the Multithreaded executor to 2 (#2030)
• check thread whether joinable before join (#2019)
• Set cpplint test timeout to 3 minutes (#2022)
• Make sure to include-what-you-use in the node_interfaces. (#2018)
• Do not clear entities callbacks on destruction (#2002)
• fix mismatched issue if using zero_allocate (#1995)
• Make ParameterService and Sync/AsyncParameterClient accept rclcpp::QoS (#1978)
• support regex match for parameter client (#1992)
• operator+= and operator-= for Duration (#1988)
• Revert “Revert “Add a create_timer method to Node and LifecycleNode classes (#1975)” (#2009) (#2010)
• force compiler warning if callback handles not captured (#2000)
• Revert “Add a create_timer method to Node and LifecycleNode classes (#1975)” (#2009)
• Add a create_timer method to Node and LifecycleNode classes (#1975)
• [docs] add note about callback lifetime for [on, post]_set_parameter_callback (#1981)
• fix memory leak (#1994)
• Support pre-set and post-set parameter callbacks in addition to on-set-parameter-callback. (#1947)
• Make create_service accept rclcpp::QoS (#1969)
• Make create_client accept rclcpp::QoS (#1964)
• Fix the documentation for rclcpp::ok to be accurate. (#1965)
• use regex for wildcard matching (#1839)
• Revert “Introduce executors new spin_for method, replace spin_until_future_complete with spin_until_complete. (#1821) (#1874)” (#1956)
• Introduce executors new spin_for method, replace spin_until_future_complete with spin_until_complete. (#1821) (#1874)
• test adjustment for LoanedMessage. (#1951)
• fix virtual dispatch issues identified by clang-tidy (#1816)
• Remove unused on_parameters_set_callback_ (#1945)
• Fix subscription.is_serialized() for callbacks with message info (#1950)
• wait for subscriptions on another thread. (#1940)
• Fix documentation of RCLCPP\_[INFO, WARN, ...] (#1943)
• Always trigger guard condition waitset (#1923)
• Add statistics for handle_loaned_message (#1927)
• Drop wrong template specialization (#1926)
• Update get_parameter_from_event to follow the function description (#1922)
• Add ‘best available’ QoS enum values and methods (#1920)
• use reinterpret_cast for function pointer conversion. (#1919)
• Contributors: Alberto Soragna, Alexander Hans, Alexis Paques, Andrew Symington, Audrow Nash, Barry Xu, Brian, Chen Lihui, Chris Lalancette, Christophe Bedard, Christopher Wecht, Cristóbal Arroyo, Daniel Reuter, Deepanshu Bansal, Emerson Knapp, Hubert Liberacki, Ivan Santiago Paunovic, Jacob Perron, Jeffery Hsu, Jochen Sprickerhof, Lei Liu, Mateusz Szczygielski, Michael Carroll, Miguel Company, Nikolai Morin, Shane Loretz, Silvio Traversaro, Tomoya Fujita, Tyler Weaver, William Woodall, Yadu, andrei, mauropasse, mergify[bot], methylDragon, schrodinbug, uupks, ymski

**rclcpp_action**

• extract the result response before the callback is issued. (#2132)
• Update all rclcpp packages to C++17. (#2121)
• Fix the GoalUUID to_string representation (#1999)
• Explicitly set callback type (#2059)
• Update maintainers (#2043)
• Do not clear entities callbacks on destruction (#2002)
• Revert “Introduce executors new spin_for method, replace spin_until_future_complete with spin_until_complete. (#1821) (#1874)” (#1956)

• Introduce executors new spin_for method, replace spin_until_future_complete with spin_until_complete. (#1821) (#1874)
• Contributors: Audrow Nash, Chris Lalancette, Hubert Liberacki, Nathan Wiebe Neufeldt, Tomoya Fujita, William Woodall, mauropasse

**rclcpp_components**

• Update all rclcpp packages to C++17. (#2121)
• Improve component_manager_isolated shutdown (#2085)
• Update maintainers (#2043)
• use unique ptr and remove unuseful container (#2013)
• Revert “Introduce executors new spin_for method, replace spin_until_future_complete with spin_until_complete. (#1821) (#1874)” (#1956)

• Introduce executors new spin_for method, replace spin_until_future_complete with spin_until_complete. (#1821) (#1874)
• Contributors: Audrow Nash, Chen Lihui, Chris Lalancette, Hubert Liberacki, Michael Carroll, William Woodall

**rclcpp_lifecycle**

• Add support for logging service. (#2122)
• Support publishing loaned messages in LifecyclePublisher (#2159)
• Fixes to silence some clang warnings. (#2127)
• Update all rclcpp packages to C++17. (#2121)
• Use the correct macro for LifecycleNode::get_fully_qualified_name (#2117)
• add get_fully_qualified_name to rclcpp_lifecycle (#2115)
• Implement Unified Node Interface (NodeInterfaces class) (#2041)
• Add clock type to node_options (#1982)
• Update maintainers (#2043)
• LifecycleNode on_configure doc fix. (#2034)
• Bugfix 20210810 get current state (#1756)
• Make lifecycle impl get_current_state() const. (#2031)
• Cleanup the lifecycle implementation (#2027)
• Cleanup the rclcpp_lifecycle dependencies. (#2021)
• Revert “Revert “Add a create_timer method to Node and LifecycleNode classes (#1975)” (#2009) (#2010)
• Revert “Add a create_timer method to Node and LifecycleNode classes (#1975)” (#2009)
• Add a create_timer method to Node and LifecycleNode classes (#1975)
• Support pre-set and post-set parameter callbacks in addition to on-set-parameter-callback. (#1947)
• Make create_service accept rclcpp::QoS (#1969)
• Make create_client accept rclcpp::QoS (#1964)
• Contributors: Andrew Symington, Audrow Nash, Chris Lalancette, Deepanshu Bansal, Ivan Santiago Paunovic, Jeffery Hsu, Lei Liu, Michael Babenko, Shane Loretz, Steve Macenski, Tomoya Fujita, methylDragon

rclpy

• Fix type in Node init args (#1115) (#1122)
• Logging service support (#1102)
• Use custom sourcedir for conf.py (#1109)
• ServerGoalHandle should be destroyed before removing. (#1113)
• Fix unnecessary list comprehension flake8 (#1112)
• Stub type hash value line in TopicEndpointInfo string (#1110)
• Support documentation generation using rosdoc2 (#1103)
• Fix Time and Duration raising exception when compared to another type (#1007)
• Make rcl_interfaces a build and exec dependency. (#1100)
• Solving Atomic undefined on OSX with clang (#1096)
• Implement matched event (#1083)
• Update service.py documentation (#1094)
• Allow space or empty strings when using ros2 param set (#1093)
• Hook up the incompatible type event inside of rclpy (#1058)
• Switch to using module instead of module_ (#1090)
• Add in subscription.get_publisher_count() (#1089)
• Service introspection (#988)
• to create a sublogger while getting child of Logger (#1084)
• Fix #983 by saving future and checking for + raising any exceptions (#1073)
• Force C++17 support on. (#1076)
• Use RCPPUTILS_SCOPE_EXIT to cleanup unparsed_indices_c. (#1075)
• Explicitly link atomic when building with Clang (#1065)
• Fix test_publisher linter for pydocstyle 6.2.2 (#1063)
• Add default preset qos profile (#1062)
• Add on_parameter_event method to the AsyncParameterClient. (#1061)
• Add documentation page for rclpy.clock (#1055)
• Rewrite test code without depending on parameter client (#1045)
• Add parallel callback test (#1044)
• decorator should not be callable. (#1050)
• typo fix. (#1049)
• Add in a warning for a depth of 0 with KEEP_LAST. (#1048)
• Add feature of wait for message (#953). (#960)
• Document rclpy.time.Time class (#1040)
• Deal with ParameterUninitializedException for parameter service (#1033)
• Improve documentation in rclpy.utilities (#1038)
• Document rclpy.utilities.remove_ros_args (#1036)
• Fix incorrect comparison on whether parameter type is NOT_SET (#1032)
• [rolling] Update maintainers (#1035)
• Set the default number of threads of the MultiThreadedExecutor to 2 (#1031)
• Update the rclpy method documentation. (#1026)
• Revert “Raise user handler exception in MultiThreadedExecutor. (#984)” (#1017)
• Waitable should check callback_group if it can be executed. (#1001)
• support wildcard matching for params file (#987)
• Raise user handler exception in MultiThreadedExecutor. (#984)
• Add wait_for_node method (#930)
• Create sublogger for action server and action client (#982)
• Support for pre-set and post-set parameter callback. (#966)
• fix gcc 7.5 build errors (#977)
• make _on_parameter_event return result correctly (#817)
• Fix a small typo in documentation. (#967)
• Add Parameter Client (#959)
• Change sphinx theme to readthedocs (#950)
• Name and type in descriptor(s) is ignored via declare_parameter(s). (#957)
• Typo fix (#951)
• Add py.typed to package (#946)
• Fix rclpy.shutdown() from hanging when triggered from callback (#947)
• Check if the context is already shutdown. (#939)
• Avoid causing infinite loop when message is empty (#935)
• Expose ‘best available’ QoS policies (#928)
• remove feedback callback when the goal has been completed. (#927)
• Allow to create a subscription with a callback that also receives the message info (#922)

rcpputils

• Add missing header for strlen (#169)
• issue-167 (#172)
• [rolling] Update maintainers - 2022-11-07 (#166)
• require C++17 and deprecate the rcppmath namespace (#165)
• Mirror rolling to master
• Fix possible race condition in create_directories() (#162)
• Contributors: Artem Shumov, Audrow Nash, Sebastian Freitag, William Woodall, bijoua29

rcutils

• fix memory leak (#423)
• Add convenience error handling macros (#421)
• Calculate the next power-of-two for the user in hash_map_init. (#420)
• update cast to modern style (#418)
• Remove deprecated header get_env.h (#417)
• Updates to rcutils to make rosdoc2 generation happier. (#416)
• add RCUTILS_LOGGING_AUTOINIT_WITH_ALLOCATOR. (#415)
• Fix memory leak in string_map.c in rcutils (#411)
• avoid unnecessary copy for rcutils_char_array_vsprintf. (#412)
• Add missing stddef include for size_t (#410)
• Add SHA256 utility implementation (#408)
• Upgrade rcutils to C++17. (#392)
• [rolling] Update maintainers - 2022-11-07 (#404)
• Fix build on OpenHarmony (#395)
• regression of thread-safety for logging macros (#393)
• add portable nonnull macros (#382)
• Fix memory leak when adding the same key to the logger hash map multiple times (#391)
• time_unix: uses ZEPHYR_VERSION_CODE instead (#390)
• Cleanup time_unix.c (#389)
• time_unix: namespace zephyr headers (#383)
• Restrict overmatching MACH ifdef to only trigger on OSX and Mach (#386)
• Optimize rcutils_logging_get_logger_effective_level() (#381)
• Change syntax __VAR_ARGS__ to __VA_ARGS__ (#376)
• Fix a bug in hash_map_get_next_key_and_data. (#375)
• More fixes from review.
• Fixes from review.
• Make g_rcutils_logging_output_handler static.
• Make g_rcutils_logging_default_logger_level static.
• Optimize rcutils_find_lastn where possible.
• Don’t bother computing the hash_map key if the hash map is empty.
• Make sure to expand char_array by at least 1.5x.
• Optimize index computation in hash_map_find.
• Improve the performance of rcutils_logging_format_message. (#372)
• Get rid of unnecessary ret variable.
• Get rid of unnecessary ifdef cplusplus checks in the C file.
• Get rid of unnecessary rcutils_custom_add_{gtest,gmock}
• Get rid of unnecessary and unused RMW switching for logging tests.
• Remove unnecessary IS_OUTPUT_COLORIZED macro.
• Rename logging internal structures to use our new convention.
• Make all of the logging ‘expand’ methods static.
• Fix up error checking for RCUTILS_CONSOLE_STDOUT_LINE_BUFFERED.
• Cleanup error handling for the RCUTILS_CONSOLE_OUTPUT_FORMAT checks.
• Revamp error handling in rcutils_logging_initialize_with_allocator.
• Revamp rcutils_logging_initialize_with_allocator.
• Make a few logging global variables static.
• Optimize calls via the RCUTILS_LOG macros. (#369)
• time_unix: add zephyr posix time (#368)
• Optimize the implementation of rcutils_char_array_strncpy. (#367)
• strdup.c: fix arbitrary length overread (#366)
Vulcanexus Documentation, Release 1.0.0

• Mirror rolling to master
• strdup.c: fix 1 byte buffer overread (#363)
• Clarify duration arg description in logging macros (#359)
• Update rctools_steady_time_now to return the same data as std::chrono (#357)
• Contributors: AlxWall, Abrar Rahman Protyasha, Audrow Nash, Chen Lihui, Chris Lalancette, Emerson Knapp, Felipe Neves, Jacob Perron, Mario Prats, Maximilian Downey Twiss, Nikolai Morin, Tomoya Fujita, William Woodall, Yakumoo, guiian, methylDragon

rmw

• Dynamic Subscription (BONUS: Allocators): rmw (#353)
• Runtime Interface Reflection: rmw (#340)
• [rmw] Improve handling of dynamic discovery (#338)
• rmw_send_response returns RMW_RET_TIMEOUT. (#350)
• Add a note about asynchronicity of discovery. (#352)
• Add matched event support (#331)
• Add type hash to rmw_topic_endpoint_info_t (rep2011) (#348)
• Add in inconsistent topic defines and data structures. (#339)
• Update documented expectations for GIDs (#335)
• Fix rmw->rwm typo (#347)
• Add rmw count clients, services (#334)
• make writer_guid uint8_t[] instead of int8_t for consistency with rmw_gid_t (#329)
• Update rmw to C++17. (#346)
• Reduce GID storage to 16 bytes. (#345)
• Move the RMW_CHECK_TYPE_IDENTIFIERS_MATCH macro to a C header. (#343)
• [rolling] Update maintainers - 2022-11-07 (#337)
• Remove unused test_loaned_message_sequence.cpp (#336)
• callback can be NULL to clear in Listener APIs. (#332)
• Add rmw_get_gid_for_client method (#327)
• Add ‘best available’ QoS policies (#320) The best available policy should select the highest level of service for the QoS setting while matching with the majority of endpoints. For example, in the case of a DDS middleware subscription, this means: * Prefer reliable reliability if all existing publishers on the same topic are reliable, otherwise use best effort. * Prefer transient local durability if all existing publishers on the same topic are transient local, otherwise use volatile. * Prefer manual by topic liveliness if all existing publishers on the same topic are manual by topic, otherwise use automatic. * Use a deadline that is equal to the largest deadline of existing publishers on the same topic. * Use a liveliness lease duration that is equal to the largest lease duration of existing publishers on the same topic.
• Move statuses definitions to rmw/events_statuses/ (#232)
• Contributors: Audrow Nash, Barry Xu, Brian, Chris Lalancette, Emerson Knapp, Geoffrey Biggs, Jacob Perron, Lee, Minju, Nikolai Morin, Tomoya Fujita, William Woodall, methylDragon
**rmw_connextdds**

- Dynamic Subscription (BONUS: Allocators): rmw_connextdds (#115)
- Revert “Refactor serialization support to use allocators and refs”
- Refactor serialization support to use allocators and refs
- Add stubs for new rmw interfaces (#111)
- Add rmw_get_gid_for_client impl (#92)
- Switch ROS2 -> ROS 2 everywhere (#83)
- Contributors: Brian, Chris Lalancette, methylDragon

**rmw_connextdds_common**

- [rmw_connextdds] New RMW discovery options (#108)
- Call get_type_hash_func (#113)
- Type hash distribution during discovery (rep2011) (#104)
- Implement matched event (#101)
- Add in implementation of inconsistent topic. (#103)
- Add rmw_get_gid_for_client impl (#92)
- Fix assert statement to allow the seconds field of a DDS_Duration_t to be zero (#88)
- Handle ‘best_available’ QoS policies in common (#85)
- Resolve build error with RTI Connext DDS 5.3.1 (#82)
- Contributors: Andrea Sorbini, Barry Xu, Brian, Chris Lalancette, Emerson Knapp, Grey, Jose Luis Rivero, Michael Carroll, Michael Jeronimo

**rmw_connextddsmicro**

- Dynamic Subscription (BONUS: Allocators): rmw_connextdds (#115)
- Add stubs for new rmw interfaces (#111)
- Add rmw_get_gid_for_client impl (#92)
- Switch ROS2 -> ROS 2 everywhere (#83)
- Contributors: Brian, Chris Lalancette, methylDragon
rmw_cyclonedds_cpp

- Dynamic Subscription (BONUS: Allocators): rmw_cyclonedds (#451)
- Add stubs for new rmw interfaces (#447)
- [rmw_cyclonedds] Improve handling of dynamic discovery (#429)
- Call get_type_hash_func (#448)
- Type hash distribution in discovery (rep2011) (#437)
- Disable inconsistent topic events. (#444)
- Implement matched event (#435)
- Implement inconsistent topic. (#431)
- Make sure to add semicolons to the CHECK_TYPE_IDENTIFIER_MATCH. (#432)
- [rolling] Update maintainers - 2022-11-07 (#428)
- Export CycloneDDS dependency (#424)
- add NULL check before accessing object. (#423)
- Add rmw_get_gid_for_client impl (#402)
- Makes topic_name a const ref
- Adds topic name to error msg when create_topic fails
- Improve error message when create_topic fails (#405)
- Change wrong use of %d to print uint32_t to PRIu32 (#253)
- Add cstring include. (#393)
- Handle ‘best_available’ QoS policies (#389)
- Contributors: Audrow Nash, Barry Xu, Brian, Chris Lalancette, Emerson Knapp, Geoffrey Biggs, Jose Luis Rivero, Shane Loretz, Tomoya Fujita, Tully Foote, Voldivh, eboasson, methylDragon

rmw_dds_common

- Type hash in GraphCache, user_data encoding tools (#70)
- Mark benchmark _ as unused. (#71)
- Update rmw_dds_common to C++17. (#69)
- Change Gid.msg to be 16 bytes. (#68)
- Minor cleanups of test_qos. (#67)
- [rolling] Update maintainers - 2022-11-07 (#65)
- build shared lib only if BUILD_SHARED_LIBS is set (#62)
- Update maintainers (#61)
- Add functions for resolving ‘best available’ QoS policies (#60) Given a QoS profile and set of endpoints for the same topic, overwrite any policies set to BEST_AVAILABLE with a policy such that it matches all endpoints while maintaining a high level of service. Add testable functions for updating BEST_AVAILABLE policies, * qos_profile_get_best_available_for_subscription * qos_profile_get_best_available_for_publisher
and add convenience functions that actually query the graph for RMW implementations to use,*
qos_profile_get_best_available_for_topic_subscription * qos_profile_get_best_available_for_topic_publisher

- Contributors: Audrow Nash, Chris Lalancette, Emerson Knapp, Jacob Perron, hannes09, methylDragon

### rmw_fastrtps_cpp

- Dynamic Subscription (BONUS: Allocators): rmw_fastrtps (#687)
- Runtime Interface Reflection: rmw_fastrtps (#655)
- [rmw_fastrtps] Improve handling of dynamic discovery (#653)
- Call get_type_hash_func (#680)
- Type hash distribution in discovery (rep2011) (#671)
- Implement inconsistent topic event (#654)
- Update all rmw_fastrtps packages to C++17. (#674)
- Rewrite how Topics are tracked in rmw_fastrtps_cpp. (#669)
- Allow loaned messages without data-sharing (#568)
- Fix incoherent dissociate_writer to dissociate_reader (#647) (#649)
- [rolling] Update maintainers - 2022-11-07 (#643)
- Add rmw_get_gid_for_client impl (#631)
- Use Fast-DDS Waitsets instead of listeners (#619)
- Remove rosidl_cmake dependency (#629)
- Revert “add line feed for RCUTILS_SAFE_FWRITE_TO_STDERR (#608)” (#612)
- add line feed for RCUTILS_SAFE_FWRITE_TO_STDERR (#608)
- Allow null arguments in the EventsExecutor parameters (#602)
- Add RMW_CHECKS to rmw_fastrtps_cpp EventsExecutor implementation
- Handle ‘best_available’ QoS policies (#598)
- Contributors: Audrow Nash, Brian, Chris Lalancette, Emerson Knapp, Geoffrey Biggs, Jacob Perron, Jose Luis Rivero, Miguel Company, Oscarchoi, Ricardo González, Tomoya Fujita, methylDragon

### rmw_fastrtps_dynamic_cpp

- Dynamic Subscription (BONUS: Allocators): rmw_fastrtps (#687)
- Runtime Interface Reflection: rmw_fastrtps (#655)
- [rmw_fastrtps] Improve handling of dynamic discovery (#653)
- Call get_type_hash_func (#680)
- Type hash distribution in discovery (rep2011) (#671)
- Implement inconsistent topic event (#654)
- Update all rmw_fastrtps packages to C++17. (#674)
- Rewrite how Topics are tracked in rmw_fastrtps_cpp. (#669)
• Allow loaned messages without data-sharing (#568)
• Fix incoherent dissociate_writer to dissociate_reader (#647) (#649)
• [rolling] Update maintainers - 2022-11-07 (#643)
• Add rmw_get_gid_for_client impl (#631)
• Use Fast-DDS Waitsets instead of listeners (#619)
• Revert “add line feed for RCUTILS_SAFE_FWRITE_TO_STDERR (#608)” (#612)
• add line feed for RCUTILS_SAFE_FWRITE_TO_STDERR (#608)
• Allow null arguments in the EventsExecutor parameters (#602)
• Add EventExecutor to rmw_fastrtps_dynamic_cpp
• Fix cpplint error (#601)
• Handle ‘best_available’ QoS policies (#598)
• Contributors: Audrow Nash, Brian, Chris Lalancette, Emerson Knapp, Geoffrey Biggs, Jacob Perron, Jose Luis Rivero, Miguel Company, Oscarchoi, Ricardo González, Tomoya Fujita, methylDragon

**rmw_fastrtps_shared_cpp**

• Fix matched event issues (#683)
• Dynamic Subscription (BONUS: Allocators): rmw_fastrtps (#687)
• Check for triggered guard conditions before waiting (#685)
• Runtime Interface Reflection: rmw_fastrtps (#655)
• [rmw_fastrtps] Improve handling of dynamic discovery (#653)
• Type hash distribution in discovery (rep2011) (#671)
• Implement matched event (#645)
• Implement inconsistent topic event (#654)
• Update all rmw_fastrtps packages to C++17. (#674)
• Rewrite how Topics are tracked in rmw_fastrtps_cpp. (#669)
• Delay lock on message callback setters (#657)
• Make sure to add semicolons to the CHECK_TYPE_IDENTIFIER_MATCH. (#658)
• Allow loaned messages without data-sharing (#568)
• Fix incoherent dissociate_writer to dissociate_reader (#647) (#649)
• [rolling] Update maintainers - 2022-11-07 (#643)
• Remove duplicated code (#637)
• Call callbacks only if unread count > 0 (#634)
• Add rmw_get_gid_for_client impl (#631)
• Use Fast-DDS Waitsets instead of listeners (#619)
• Take all available samples on service/client on_data_available. (#616)
• Revert “add line feed for RCUTILS_SAFE_FWRITE_TO_STDERR (#608)” (#612)
• add line feed for RCUTILS_SAFE_FWRITE_TO_STDERR (#608)
• Contributors: Audrow Nash, Barry Xu, Brian, Chris Lalancette, Emerson Knapp, Geoffrey Biggs, Michael Carroll, Miguel Company, Oscarchoi, Ricardo González, Tomoya Fujita, mauropasse, methylDragon

rmw_implementation

• Dynamic Subscription (BONUS: Allocators): rmw_implementation (#219)
• Runtime Interface Reflection: rmw_implementation (#215)
• Mark the benchmark _ variables as unused. (#218)
• Update rmw_implementation to C++17. (#214)
• [rolling] Update maintainers - 2022-11-07 (#212)
• Build-time RMW selection does not need ament_index_cpp (#210)
• Add rmw_get_gid_for_client & tests (#206)
• Contributors: Audrow Nash, Brian, Chris Lalancette, G.A. vd. Hoorn, methylDragon

rmw_implementation_cmake

• [rolling] Update maintainers - 2022-11-07 (#337)
• Contributors: Audrow Nash

robot_state_publisher

• Update robot_state_publisher to C++17. (#204)
• [rolling] Update maintainers - 2022-11-07 (#203)
• Mirror rolling to ros2
• Contributors: Audrow Nash, Chris Lalancette

ros2action

• Make all of the dependencies in pure Python packages exec_depend. (#823)
• [rolling] Update maintainers - 2022-11-07 (#776)
• Contributors: Audrow Nash, Chris Lalancette
ros2bag

- Cleanup the help text for ros2 bag record. (#1329) (#1333)
- Enable document generation using rosdoc2 for ament_python pkgs (#1260)
- CLI: Get storage-specific values from plugin (#1209)
- Fix up some of the wording in the record help text. (#1228)
- Add topic_name option to info verb (#1217)
- rosbag2_storage: set MCAP as default plugin (#1160)
- rosbag2_py: parametrize tests across storage plugins (#1203)
- Added option to change node name for the recorder from the Python API (#1180)
- rosbag2_cpp: test more than one storage plugin (#1196)
- rosbag2_storage: expose default storage ID as method (#1146)
- Fix for ros2 bag play exit with non-zero code on SIGINT (#1126)
- ros2bag: move storage preset validation to sqlite3 plugin (#1135)
- Add option to prevent message loss while converting (#1058)
- Added support for excluding topics via regular expressions (#1046)
- Readers/info can accept a single bag storage file, and detect its storage id automatically (#1072)
- Add short -v option to ros2 bag list for verbose (#1065)
- Use a single variable for evaluating the filter regex (#1053)
- Add additional mode of publishing sim time updates triggered by replayed messages (#1050)
- Renamed –topics-regex to –regex and -e in Player class to be consistent with Recorder (#1045)
- Use first available writer in recording if default sqlite3 not available. (#1044)
- Add the ability to record any key/value pair in ‘custom’ field in metadata.yaml (#1038)
- Added support for filtering topics via regular expressions on Playback (#1034)
- Fix incorrect boundary check for playback_duration and play_until_timestamp (#1032)
- Adds play until timestamp functionality (#1005)
- Add CLI verb for burst mode of playback (#980)
- Add play-for specified number of seconds functionality (#960)
- Make unpublished topics unrecorded by default (#968)
ros2cli

- Set automatically_declare_parameters_from_overrides in DirectNode. (#813)
- Enable document generation using rosdoc2 (#811)
- Fix linters (#808)
- add timeout option for ros2param to find node. (#802)
- Save method list via connection check to XMLRPC server. (#796)
- ZSH argcomplete: call compinit only if needed (#750)
- Fix network aware node issue (#785)
- [rolling] Update maintainers - 2022-11-07 (#776)
- XMLRPC server accepts request from all local IP addresses. (#729)
- Contributors: Audrow Nash, Chris Lalancette, Cristóbal Arroyo, Ivan Santiago Paunovic, Tomoya Fujita, Yadu, mjbgusz

ros2cli_common_extensions

- [rolling] Update maintainers - 2022-11-07 (#7)
- Update maintainers (#6)
- Contributors: Audrow Nash, methylDragon

ros2cli_test_interfaces

- [rolling] Update maintainers - 2022-11-07 (#776)
- Remove action_msgs dependency (#743)
- Contributors: Audrow Nash, Jacob Perron

ros2component

- Enable document generation using rosdoc2 (#811)
- [rolling] Update maintainers - 2022-11-07 (#776)
- Fix the component load help to mention load, not unload. (#756)
- Remove unused arguments from ros2 component types. (#711)
- Contributors: Audrow Nash, Chris Lalancette, Yadu
ros2doctor

• Shutdown ros2doctor hello when ctrl-c is received (#829)
• Make all of the dependencies in pure Python packages exec_depend. (#823)
• Enable document generation using rosdoc2 (#811) * Fix warnings for ros2component, ros2doctor, ros2interface, and ros2node
• [rolling] Update maintainers - 2022-11-07 (#776)
• Contributors: Audrow Nash, Chris Lalancette, Michael Carroll, Yadu

ros2interface

• Make all of the dependencies in pure Python packages exec_depend. (#823)
• Enable document generation using rosdoc2 (#811)
• [rolling] Update maintainers - 2022-11-07 (#776)
• Contributors: Audrow Nash, Chris Lalancette, Yadu

ros2launch

• [rolling] Update maintainers - 2022-11-07 (#331)
• Contributors: Audrow Nash

ros2lifecycle

• Make all of the dependencies in pure Python packages exec_depend. (#823)
• [rolling] Update maintainers - 2022-11-07 (#776)
• Contributors: Audrow Nash, Chris Lalancette

ros2lifecycle_test_fixtures

• Update the ros2cli test fixture to C++17. (#789)
• [rolling] Update maintainers - 2022-11-07 (#776)
• Contributors: Audrow Nash, Chris Lalancette

ros2multicast

• Make all of the dependencies in pure Python packages exec_depend. (#823)
• [rolling] Update maintainers - 2022-11-07 (#776)
• Add –group and –port options to ros2 multicast (#770)
• Contributors: Audrow Nash, Chris Lalancette, Shane Loretz
ros2node

- Make all of the dependencies in pure Python packages exec_depend. (#823)
- Enable document generation using rosdoc2 (#811) * Fix warnings for ros2component, ros2doctor, ros2interface, and ros2node
- Fix linters (#808)
- add timeout option for ros2param to find node. (#802)
- [rolling] Update maintainers - 2022-11-07 (#776)
- Updated wording in list.py (#775)
- Contributors: Audrow Nash, Chris Lalancette, Cristóbal Arroyo, Michael Wrock, Tomoya Fujita, Yadu

ros2param

- remove deprecated options (#824)
- Make all of the dependencies in pure Python packages exec_depend. (#823)
- add timeout option for ros2param to find node. (#802)
- Fix printing of integer and double arrays. (#804)
- [rolling] Update maintainers - 2022-11-07 (#776)
- refactor: make ros2param use rclpy.parameter_client (#716)
- Contributors: Audrow Nash, Brian, Chris Lalancette, Tomoya Fujita

ros2pkg

- Fix typo in ros2pkg warning message. (#828)
- Make all of the dependencies in pure Python packages exec_depend. (#823)
- resolve #790 (#801)
- Add alias library targets for CMake (#718)
- [rolling] Update maintainers - 2022-11-07 (#776)
- Contributors: Audrow Nash, Chris Lalancette, Kenji Brameld, RFRIEDM-Trimble, Tomoya Fujita

ros2run

- Make all of the dependencies in pure Python packages exec_depend. (#823)
- [rolling] Update maintainers - 2022-11-07 (#776)
- Contributors: Audrow Nash, Chris Lalancette
ros2service

• Make all of the dependencies in pure Python packages exec_depend. (#823)
• [rolling] Update maintainers - 2022-11-07 (#776)
• Contributors: Audrow Nash, Chris Lalancette

ros2test

• [rolling] Update maintainers - 2022-11-07 (#12)
• update maintainer
• Contributors: Audrow Nash, Dharini Dutia, quarkytale

ros2topic

• remove deprecated options (#824)
• Make all of the dependencies in pure Python packages exec_depend. (#823)
• Expect type hash cli output in test (#822)
• Fix the type annotation in pub.py. (#814)
• Switch to using new event_handler instead of qos_event. (#787)
• avoid flaky test that subscriber might not receive the message (#810)
• Adds a --max-wait-time option to ros2 topic pub (#800)
• Fix some flake8 warnings related to style. (#805)
• Adds a timeout feature to rostopic echo (#792)
• Refactor common types (#791)
• Allow configuring liveliness in ros2 topic echo and pub (#788)
• Extend timeout to shutdown the command line process. (#783)
• [rolling] Update maintainers - 2022-11-07 (#776)
• a couple of typo fixes. (#774)
• Add support use_sim_time for ros2 topic hz/bw/pub. (#754)
• Use set_message_fields from rosidl_runtime_py (#761)
• Expand auto to the current time when passed to a Header field (#749)
• Add verbose option to echo that also prints the associated message info (#707)
• update docs for bandwidth functions. (#709)
• Split the bandwidth functions into a get and print. (#708)
• Contributors: Arjo Chakravarty, Audrow Nash, Chen Lihui, Chris Lalancette, Emerson Knapp, Esteve Fernandez, Ivan Santiago Paunovic, Lei Liu, Tomoya Fujita
Vulcanexus Documentation, Release 1.0.0

ros2trace

- Move ros2trace tests to new test_ros2trace package (#63)
- Error out if trace already exists unless 'append' option is used (#58)
- Improve ‘ros2 trace’ command error handling & add end-to-end tests (#54)
- Contributors: Christophe Bedard

ros_testing

- [rolling] Update maintainers - 2022-11-07 (#12)
- update maintainer
- Contributors: Audrow Nash, Dharini Dutia, quarkytale

rosbag2

- Add Michael Orlov as maintainer in rosbag2 packages (#1215)
- Move sqlite3 storage implementation to rosbag2_storage_sqlite3 package (#1113)
- Contributors: Emerson Knapp, Michael Orlov

rosbag2_compression

- Add in a missing cstdint include. (#1321) (#1322)
- Fix warning from ClassLoader in sequential compression reader and writer (#1299) (#1316)
- Add message definition read API (#1292)
- rosbag2_storage: add type description hash to topic metadata (#1272)
- rosbag2_cpp: move local message definition source out of MCAP plugin (#1265)
- Update rosbag2 to C++17. (#1238)
- Use target_link_libraries instead of ament_target_dependencies (#1202)
- set_read_order: return success (#1177)
- Add update_metadata(BagMetadata) API for storage plugin interface (#1149)
- Reverse read order API and sqlite storage implementation (#1083)
- Add option to prevent message loss while converting (#1058)
- set default metadata of compressed message (in case compressor does not set it) (#1060)
- Speed optimization: Preparing copyless publish/subscribing by using const message for writing (#1010)
- Add the ability to record any key/value pair in ‘custom’ field in metadata.yaml (#1038)
- Contributors: Chris Lalancette, Daisuke Nishimatsu, DensoADAS, Emerson Knapp, Hunter L. Allen, Joshua Hampp, Michael Orlov, Tony Peng, james-rms, mergify[bot]
rosbag2_compression_zstd

- Update rosbag2 to C++17. (#1238)
- Use target_link_libraries instead of ament_target_dependencies (#1202)
- Add Michael Orlov as maintainer in rosbag2 packages (#1215)
- Speed optimization: Preparing copyless publish/subscribing by using const message for writing (#1010)
- Contributors: Chris Lalancette, Daisuke Nishimatsu, DensoADAS, Joshua Hampp, Michael Orlov

rosbag2_cpp

- Add recorder stop() API (#1300) (#1344)
- Add type_hash in MessageDefinition struct (#1296)
- Add message definition read API (#1292)
- rosbag2_storage: add type description hash to topic metadata (#1272)
- Fix for flaky TimeControllerClockTest::unpaused_sleep_returns_true test (#1290)
- rosbag2_cpp: move local message definition source out of MCAP plugin (#1265)
- Update rosbag2 to C++17. (#1238)
- Use target_link_libraries instead of ament_target_dependencies (#1202)
- Fix rwm->rmw spelling (#1249)
- Expose more Writer methods in python interface (#1220)
- rosbag2_storage: set MCAP as default plugin (#1160)
- Parametrize all rosbag2_tests for both supported storage plugins (#1221)
- rosbag2_cpp: test more than one storage plugin (#1196)
- Replace language for “db3”/”db”/”database” (#1194)
- set_read_order: return success (#1177)
- Remove explicit sqlite3 from code (#1166)
- Add update_metadata(BagMetadata) API for storage plugin interface (#1149)
- Reader and writer can use default storage by not specifying (#1167)
- rosbag2_storage: expose default storage ID as method (#1146)
- Don’t reopen file for every seek if we don’t have to. Search directionally for the correct file (#1117)
- Add SplitBagfile recording service. (#1115)
- Reverse read order API and sqlite storage implementation (#1083)
- Replace std::filesystem::path(..) with rcpputils::fs::path(..) (#1104)
- Fix issue where sequentialwriter only sets metadata duration to the duration of the final file (#1098)
- Delete obsolete compression_options.cpp from rosbag2_cpp (#1078)
- Readers/info can accept a single bag storage file, and detect its storage id automatically (#1072)
- Remove deprecated rosbag2_cpp/storage_options.hpp, for post-Humble releases (#1064)
- Speed optimization: Preparing copyless publish/subscribing by using const message for writing (#1010)
- Add the ability to record any key/value pair in ‘custom’ field in metadata.yaml (#1038)
- Notification of significant events during bag recording and playback (#908)
- Bugfix for “Playing the bags recorded with split by duration/size is playing only the last recorded .db3.” (#1022)
- Improve test_time_controller test (#1012)
- Contributors: Chris Lalancette, Daisuke Nishimatsu, DensoADAS, Emerson Knapp, Geoffrey Biggs, Hunter L. Allen, Jorge Perez, Joshua Hampp, Kaju-Bubanja, Michael Orlov, Tony Peng, james-rms, mergify[bot], rshanor

### rosbag2_examples_cpp

- rosbag2_storage: add type description hash to topic metadata (#1272)
- Update rosbag2 to C++17. (#1238)
- Use target_link_libraries instead of ament_target_dependencies (#1202)
- Add Michael Orlov as maintainer in rosbag2 packages (#1215)
- Add API samples on main branch - Rolling C++ API examples (#1068)
- Contributors: Chris Lalancette, Daisuke Nishimatsu, Emerson Knapp, Michael Orlov, james-rms

### rosbag2_examples_py

- Fix a warning from python setuptools. (#1312) (#1314)
- Add API samples for Python [rebased] (#1253) * Add API samples for Python * Package Renaming and Move * linting + copyright * more linting ——— Co-authored-by: Geoffrey Biggs <gbiggs@killbots.net>
- Contributors: David V. Lu!!, mergify[bot]

### rosbag2_interfaces

- Update rosbag2 to C++17. (#1238)
- Add Michael Orlov as maintainer in rosbag2 packages (#1215)
- Add SplitBagfile recording service. (#1115)
- Adds stop operation for rosbag2::Player (#1007)
- Notification of significant events during bag recording and playback (#908)
- Adds play until timestamp functionality (#1005)
- Add CLI verb for burst mode of playback (#980)
- Add play-for specified number of seconds functionality (#960)
- Contributors: Agustin Alba Chicar, Chris Lalancette, Geoffrey Biggs, Michael Orlov, Misha Shalem, rshanor
rosbag2_performance_benchmarking

- Add tests for rosbag2_performance_benchmarking pkg (#1268)
- Fix expectations for rosbag2 return code in rosbag2_performance_benchmarking (#1267)
- Update rosbag2 to C++17. (#1238)
- Use thread pool to run benchmark publishers in rosbag2_performance_benchmarking (#1250)
- Use target_link_libraries instead of ament_target_dependencies (#1202)
- Skip ament_package() call when not building rosbag2_performance_benchmarking (#1242)
- Add option to specify a message type (#1153)
- Add Michael Orlov as maintainer in rosbag2 packages (#1215)
- Replace language for “db3”/“db”/“database” (#1194)
- Remove explicit sqlite3 from code (#1166)
- Contributors: Chris Lalancette, Daisuke Nishimatsu, Emerson Knapp, Michael Orlov, Shane Loretz, carlossvg

rosbag2_performance_benchmarking_msgs

- Add tests for rosbag2_performance_benchmarking_pkg (#1268)
- Skip ament_package() call when not building rosbag2_performance_benchmarking (#1242)
- [rolling] Bump to 0.19.0 (#1232)
- Add option to specify a message type (#1153)
- Contributors: Audrow Nash, Michael Orlov, Shane Loretz, carlossvg

rosbag2_py

- Add binding to close the writer (#1339) (#1340)
- Add type_hash in MessageDefinition struct (#1296)
- Store message definitions in SQLite3 storage plugin (#1293)
- Add message definition read API (#1292)
- rosbag2_storage: add type description hash to topic metadata (#1272)
- rosbag2_cpp: move local message definition source out of MCAP plugin (#1265)
- Update rosbag2 to C++17. (#1238)
- Use target_link_libraries instead of ament_target_dependencies (#1202)
- Expose more Writer methods in python interface (#1220)
- rosbag2_storage: set MCAP as default plugin (#1160)
- Add Michael Orlov as maintainer in rosbag2 packages (#1215)
- rosbag2_py: parametrize tests across storage plugins (#1203)
- Added option to change node name for the recorder from the Python API (#1180)
- Replace language for “db3”/“db”/“database” (#1194)
- Remove explicit sqlite3 from code (#1166)
- Move python get_default_storage_id to storage module instead of writer (#1165)
- rosbag2_storage: expose default storage ID as method (#1146)
- rosbag2_py: set defaults for config when bag rewriting (#1121)
- Reverse read order API and sqlite storage implementation (#1083)
- expose py Reader metadata, improve rosbag2_py.BagMetadata usability (#1082)
- Added support for excluding topics via regular expressions (#1046)
- Use a single variable for evaluating the filter regex (#1053)
- Add additional mode of publishing sim time updates triggered by replayed messages (#1050)
- Renamed –topics-regex to –regex and -e in Player class to be consistent with Recorder (#1045)
- Add the ability to record any key/value pair in ‘custom’ field in metadata.yaml (#1038)
- Added support for filtering topics via regular expressions on Playback (#1034)
- Adds play until timestamp functionality (#1005)
- Add CLI verb for burst mode of playback (#980)
- Add play-for specified number of seconds functionality (#960)
- Make unpublished topics unrecorded by default (#968)
- Fix test rosbag2_py test compatibility with Python < 3.8 (#987)

**rosbag2_storage**

- Add type_hash in MessageDefinition struct (#1296)
- Add message definition read API (#1292)
- rosbag2_storage: add type description hash to topic metadata (#1272)
- rosbag2_cpp: move local message definition source out of MCAP plugin (#1265)
- Update rosbag2 to C++17. (#1238)
- Use target_link_libraries instead of ament_target_dependencies (#1202)
- rosbag2_storage: set MCAP as default plugin (#1160)
- Add Michael Orlov as maintainer in rosbag2 packages (#1215)
- set_read_order: return success (#1177)
- Remove explicit sqlite3 from code (#1166)
- Add update_metadata(BagMetadata) API for storage plugin interface (#1149)
- rosbag2_storage: expose default storage ID as method (#1146)
- Don’t reopen file for every seek if we don’t have to. Search directionally for the correct file (#1117)
- Reverse read order API and sqlite storage implementation (#1083)
• Remove YAML_CPP_DLL define (#964)
• Added support for excluding topics via regular expressions (#1046)
• Readers/info can accept a single bag storage file, and detect its storage id automatically (#1072)
• Use a single variable for evaluating the filter regex (#1053)
• Speed optimization: Preparing copyless publish/subscribing by using const message for writing (#1010)
• Renamed –topics-regex to –regex and -e in Player class to be consistent with Recorder (#1045)
• Add the ability to record any key/value pair in ‘custom’ field in metadata.yaml (#1038)
• Added support for filtering topics via regular expressions on Playback (#1034)
• Contributors: Akash, Chris Lalancette, Daisuke Nishimatsu, DensoADAS, Emerson Knapp, Esteve Fernandez, Hunter L. Allen, Joshua Hampp, Michael Orlov, Tony Peng, james-rms

rosbag2_storage_default_plugins

• rosbag2_storage: set MCAP as default plugin (#1160)
• Add Michael Orlov as maintainer in rosbag2 packages (#1215)
• Move sqlite3 storage implementation to rosbag2_storage_sqlite3 package (#1113)
• Reverse read order API and sqlite storage implementation (#1083)
• Add support for old db3 schema used on distros prior to Foxy (#1090)
• Added support for excluding topics via regular expressions (#1046)
• Contributors: Emerson Knapp, Esteve Fernandez, Michael Orlov, james-rms

rosbag2_storage_mcap

• Add type_hash in MessageDefinition struct (#1296)
• Add message definition read API (#1292)
• rosbag2_storage: add type description hash to topic metadata (#1272)
• rosbag2_cpp: move local message definition source out of MCAP plugin (#1265)
• Update rosbag2 to C++17. (#1238)
• Use target_link_libraries instead of ament_target_dependencies (#1202)
• CLI: Get storage-specific values from plugin (#1209)
• Add Michael Orlov as maintainer in rosbag2 packages (#1215)
• rosbag2_cpp: test more than one storage plugin (#1196)
• set_read_order: return success (#1177)
• rosbag2_storage_mcap: merge into rosbag2 repo (#1163)
• mcap_storage: ‘none’ is a valid storage preset profile (#86)
• mcap_storage: handle update_metadata call (#83)
• Update clang-format rules to fit ROS 2 style guide (#80)
• Revert “read_order: throw exception from set_read_order for unsupported orders”
• read_order: throw exception from set_read_order for unsupported orders
• Fix compile flags to work on rosbag_storage:0.17.x (#78)
• Fix Windows build (#73)
• set defaults for SQLite plugin parity (#68)
• rosbag2_storage_mcap: add storage preset profiles (#57)
• rename test_fixture_interfaces package to testdata (#64)
• Switch to using the vendored zstd library. (#59)
• Add set_read_order reader API (#54)
• Some minor improvements in rosbag2_storage_mcap after review (#58)
• Revert “rosbag2_storage_mcap: add storage preset profiles”
• rosbag2_storage_mcap: add storage preset profiles
• Store IDL message definitions in Schema records when no MSG definition is available (#43)
• Support timestamp-ordered playback (#50)
• Support regex topic filtering
• Add all lz4 sources to fix undefined symbols at runtime (#46)
• Upgrade mcap to fix LZ4 error and segfault (#42)
• Fix build for Foxy (#34)
• fix: minor issues (#31) * remove unnecessary block * use target_link_libraries instead of ament_target_dependencies * remove ros environment * add prefix to compile definition
• Update email address for Foxglove maintainers (#32)
• Added mcap_vendor package. Updated CMakeLists.txt to fetch dependencies with FetchContent rather than Conan.
• CMake build script will now execute pip install conan automatically.
• [1.0.0] Use Summary section for get_metadata() and seek(), implement remaining methods (#17)
• feat: add play impl (#16)
• chore: refine package.xml (#15)
• Don’t throw when READ_WRITE mode is used; add .mcap file extension to recorded files (#14)
• Add dynamic message definition lookup (#13)
• Switch C++ formatter to clang-format (#12)
• Merge pull request #7 from ros-tooling/jhurliman/reader-writer
• uninitialized struct
• lint
• lint
• lint
• Reader and writer implementation
• Merge pull request #6 from wep21/add-metadata-impl
• feat: add metadata impl
• Merge pull request #5 from wep21/mcap-storage-impl
• chore: update cmake minimum version
• chore: install mcap header
• chore: include mcap header
• fix: move fetch content into rosbag2 storage mcap
• Merge pull request #3 from ros-tooling/emersonknapp/mcap_plugin_skeleton
• Add rosbag2_storage_mcap skeleton
• Contributors: Andrew Symington, Chris Lalancette, Daisuke Nishimatsu, Emerson Knapp, Jacob Bandes-Storch, James Smith, John Hurliman, Michael Orlov, james-rms, wep21

rosbag2_storage_sqlite3

• Add type_hash in MessageDefinition struct (#1296)
• Store message definitions in SQLite3 storage plugin (#1293)
• Add message definition read API (#1292)
• rosbag2_storage: add type description hash to topic metadata (#1272)
• rosbag2_cpp: move local message definition source out of MCAP plugin (#1265)
• Update rosbag2 to C++17. (#1238)
• Use target_link_libraries instead of ament_target_dependencies (#1202)
• CLI: Get storage-specific values from plugin (#1209)
• Add Michael Orlov as maintainer in rosbag2 packages (#1215)
• Remove sqlite3-specific info from main README, make it more storage agnostic and point to plugin-specific README (#1193)
• set_read_order: return success (#1177)
• Add update_metadata(BagMetadata) API for storage plugin interface (#1149)
• Store db schema version and ROS_DISTRIBUTION name in db3 files (#1156)
• ros2bag: move storage preset validation to sqlite3 plugin (#1135)
• Move sqlite3 storage implementation to rosbag2_storage_sqlite3 package (#1113)
• Use a single variable for evaluating the filter regex (#1053)
• Renamed --topics-regex to --regex and -e in Player class to be consistent with Recorder (#1045)
• Added support for filtering topics via regular expressions on Playback (#1034)
• Contributors: Chris Lalancette, Daisuke Nishimatsu, Emerson Knapp, Esteve Fernandez, Michael Orlov, james-rms
rosbag2_test_common

- Address flakiness in rosbag2_play_end_to_end tests (#1297) (#1330)
- Update rosbag2 to C++17. (#1238)
- Use target_link_libraries instead of ament_target_dependencies (#1202)
- Add Michael Orlov as maintainer in rosbag2 packages (#1215)
- rosbag2_py: parametrize tests across storage plugins (#1203)
- Fix for ros2 bag play exit with non-zero code on SIGINT (#1126)
- Split up the include of rclcpp.hpp (#1027)
- Contributors: Chris Lalancette, Daisuke Nishimatsu, Michael Orlov, james-rms, mergify[bot]

rosbag2_test_msgdefs

- rosbag2_cpp: move local message definition source out of MCAP plugin (#1265) The intention of this PR is to move the message-definition-finding capability outside of rosbag2_storage_mcapi, and allow any rosbag2 storage plugin to store message definitions.
- Contributors: james-rms

rosbag2_tests

- Address flakiness in rosbag2_play_end_to_end tests (#1297) (#1330)
- Add type_hash in MessageDefinition struct (#1296)
- rosbag2_cpp: move local message definition source out of MCAP plugin (#1265)
- Update rosbag2 to C++17. (#1238)
- Use target_link_libraries instead of ament_target_dependencies (#1202)
- rosbag2_storage: set MCAP as default plugin (#1160)
- Add Michael Orlov as maintainer in rosbag2 packages (#1215)
- Parametrize all rosbag2_tests for both supported storage plugins (#1221)
- Make rosbag2_tests agnostic to storage implementation (#1192)
- Get rid from attempt to open DB file in wait_for_db() test fixture (#1141)
- Fix for ros2 bag play exit with non-zero code on SIGINT (#1126)
- Move sqlite3 storage implementation to rosbag2_storage_sqlite3 package (#1113)
- Readers/info can accept a single bag storage file, and detect its storage id automatically (#1072)
- Add the ability to record any key/value pair in ‘custom’ field in metadata.yaml (#1038)
- Contributors: Chris Lalancette, Daisuke Nishimatsu, Emerson Knapp, Hunter L. Allen, Michael Orlov, Tony Peng, james-rms, mergify[bot]
rosbag2_transport

- Change subscriptions from GenericSubscription to SubscriptionBase (#1338)
- Add recorder stop() API (#1300) (#1334)
- Read message definitions from input files in bag_rewrite (#1295)
- Add message definition read API (#1292)
- Move rosbag2_transport::Recorder implementation to pimpl (#1291)
- rosbag2_storage: add type description hash to topic metadata (#1272)
- rosbag2_cpp: move local message definition source out of MCAP plugin (#1265)
- Use RMW methods to initialize endpoint info instead of brace initializer to guard against upcoming struct change (#1257)
- Update rosbag2 to C++17. (#1238)
- Use target_link_libraries instead of ament_target_dependencies (#1202)
- Print “Hidden topics are not recorded” only once. (#1225)
- rosbag2_storage: set MCAP as default plugin (#1160)
- Add Michael Orlov as maintainer in rosbag2 packages (#1215)
- rosbag2_transport: parametrize test_rewrite (#1206)
- rosbag2_cpp: test more than one storage plugin (#1196)
- Replace language for “db3””db””database” (#1194)
- set_read_order: return success (#1177)
- Remove explicit sqlite3 from code (#1166)
- Add pause and resume service calls for rosbag2 recorder (#1131)
- Redesign record_services tests to make them more deterministic (#1122)
- Add SplitBagfile recording service. (#1115)
- Reverse read order API and sqlite storage implementation (#1083)
- make recorder node composable by inheritance (#1093)
- Mark test_play_services as xfail for FastRTPS and CycloneDDS (#1091)
- fixed typo (#1057)
- Fix hangout in rosbag2 player and recorder when pressing CTRL+C (#1081)
- Added support for excluding topics via regular expressions (#1046)
- Use a single variable for evaluating the filter regex (#1053)
- Add additional mode of publishing sim time updates triggered by replayed messages (#1050)
- Speed optimization: Preparing copyless publish/subscribing by using const message for writing (#1010)
- Renamed –topics-regex to –regex and -e in Player class to be consistent with Recorder (#1045)
- Refactor play until and duration tests (#1024)
- Added support for filtering topics via regular expressions on Playback (#1034)
- Adds stop operation for rosbag2::Player (#1007)
• Fix incorrect boundary check for `playback_duration` and `play_until_timestamp` (#1032)
• Split up the include of `rclcpp.hpp` (#1027)
• Notification of significant events during bag recording and playback (#908)
• Adds play until timestamp functionality (#1005)
• Add CLI verb for burst mode of playback (#980)
• Add on play message callbacks to the `robag2::Player` class (#1004)
• Add play-for specified number of seconds functionality (#960)
• Reduce message spam when topics to be recorded do not exist (#1018)
• Address flakiness in record_all_with_sim_time test (#1014)
• Add debug instrumentation for `test_play_services` (#1013)
• Fix for robag2::Player freeze when pressing ctrl+c in pause mode (#1002)
• Add the `/bigobj` flag to Windows Debug builds. (#1009)
• Make unpublished topics unrecorded by default (#968)
• Make `peek_next_message_from_queue` return a `SharedPtr`. (#993)
• Change the topic names in `test_record.cpp` (#988)
• Contributors: Agustin Alba Chicar, Bernardo Taveira, Brian, Chris Lalancette, Cristóbal Arroyo, Daisuke Nishimatsu, DensoADAS, Emerson Knapp, Esteve Fernandez, Geoffrey Biggs, Jorge Perez, Joshua Hampp, Michael Orlov, Misha Shalem, Sean Kelly, Tony Peng, james-rms, kylemarcey, mergify[bot], rshanor

**rosgraph_msgs**

• Update common_interfaces to C++17. (#215) (#151)
• [rolling] Update maintainers - 2022-11-07 (#150)
• Contributors: Audrow Nash, Chris Lalancette

**rosidl_adapter**

• `rosidl_adapter/cmake/rosidl_adapt_interfaces.cmake`: Make ament free (#709)
• [service introspection] generate service_event messages (#700)
• Adding tests for unicode support in message comments. (#720)
• [rolling] Update maintainers - 2022-11-07 (#717)
• Add action2idl script (#654)
• Contributors: Audrow Nash, Brian, Guilherme Henrique Galelli Christmann, John Daktylidis, Yasushi SHOJI
rosidl_cli

- Fix warnings (#726)
- [rolling] Update maintainers - 2022-11-07 (#717)
- Contributors: Audrow Nash, Yadu

rosidl_cmake

- Type Description Codegen and Typesupport (rep2011) (#727)
- Type hash in interface codegen (rep2011) (#722)
- [service introspection] generate service_event messages (#700)
- [rolling] Update maintainers - 2022-11-07 (#717)
- Skip rosidl_generate_interfaces dependency export on SKIP_INSTALL. (#708)
- Move rosidl_cmake Python module to a new package rosidl_pycommon (#696) Deprecate the Python module in rosidl_cmake and move the implementation to the new package rosidl_pycommon.
- Fix comment in camel case conversion function (#683)
- Protect rosidl_target_interfaces from using NOTFOUND in include_directories (#679)
- Contributors: Audrow Nash, Brian, Chris Lalancette, Emerson Knapp, Jacob Perron, Jose Luis Rivero, Shane Loretz

rosidl_core_generators

- [rolling] Update maintainers - 2022-11-07 (#2)
- Add generators and runtime configuration packages (#1) Moved (and renamed) from rosidl_defaults. Related PR: https://github.com/ros2/rosidl_defaults/pull/22
- Contributors: Audrow Nash, Jacob Perron

rosidl_core_runtime

- [rolling] Update maintainers - 2022-11-07 (#2)
- Add generators and runtime configuration packages (#1) Moved (and renamed) from rosidl_defaults. Related PR: https://github.com/ros2/rosidl_defaults/pull/22
- Contributors: Audrow Nash, Jacob Perron
Vulcanexus Documentation, Release 1.0.0

rosidl_default_generators

- add servicemsgs depend (#24)
- Move dependencies to rosidl_core and depend on action_msgs (#22) Move implementation to new packages rosidl_core_generators and rosidl_runtime_generators The new packages are located in a separate repository: https://github.com/ros2/ Rosidl_core.git Rosidl_defaults now depends on the new packages, plus message definitions required for Actions (namely action_msgs). This allows users to avoid having to explicitly depend on action_msgs.
- Contributors: Audrow Nash, Brian, Jacob Perron

rosidl_default_runtime

- add servicemsgs depend (#24)
- Move dependencies to rosidl_core and depend on action_msgs (#22) Move implementation to new packages rosidl_core_generators and rosidl_runtime Generators The new packages are located in a separate repository: https://github.com/ros2/ Rosidl_core.git Rosidl_defaults now depends on the new packages, plus message definitions required for Actions (namely action_msgs). This allows users to avoid having to explicitly depend on action_msgs.
- Contributors: Audrow Nash, Brian, Jacob Perron

rosidl_dynamic_typesupport

- Fix up the exports for rosidl_dynamic_typesupport. (#5)
- Refactor dynamic message type support impl to use allocators (#2)
- Runtime Interface Reflection: rosidl_dynamic_typesupport (#1)
- Contributors: Chris Lalancette, William Woodall, methylDragon

rosidl_dynamic_typesupport_fastrtps

- Remove more unnecessary semicolons (#4)
- Dynamic Subscription (BONUS: Allocators): rosidl_dynamic_typesupport_fastrtps (#3)
- Remove unnecessary semicolons. (#2)
- Runtime Interface Reflection: rosidl_dynamic_typesupport_fastrtps (#1)
- Contributors: Chris Lalancette, methylDragon
rosidl_generator_c

- Type Description Codegen and Typesupport (rep2011) (#727)
- Expose type hash on typesupports (rep2011) (#729)
- Type hash in interface codegen (rep2011) (#722)
- [service introspection] generate service_event messages (#700)
- [rolling] Update maintainers - 2022-11-07 (#717)
- Move rosidl_generator_c/cpp tests to a separate package (#701)
- Move rosidl_cmake Python module to a new package rosidl_pycommon (#696) Deprecate the Python module in rosidl_cmake and move the implementation to the new package rosidl_pycommon.
- Add namespaced ALIAS target to easily consume generated libraries via add_subdirectory (#605)
- Contributors: Audrow Nash, Brian, Emerson Knapp, Jacob Perron, Silvio Traversaro

rosidl_generator_cpp

- Type Description Codegen and Typesupport (rep2011) (#727)
- Expose type hash on typesupports (rep2011) (#729)
- Type hash in interface codegen (rep2011) (#722)
- [service introspection] generate service_event messages (#700)
- [rolling] Update maintainers - 2022-11-07 (#717)
- Move rosidl_generator_c/cpp tests to a separate package (#701)
- Move rosidl_cmake Python module to a new package rosidl_pycommon (#696) Deprecate the Python module in rosidl_cmake and move the implementation to the new package rosidl_pycommon.
- Add namespaced ALIAS target to easily consume generated libraries via add_subdirectory (#605)
- Contributors: Audrow Nash, Brian, Emerson Knapp, Jacob Perron, Silvio Traversaro

rosidl_generator_dds_idl

- [rolling] Update maintainers - 2022-11-07 (#60)
- Replace rosidl_cmake imports with rosidl_pycommon (#59)
- Contributors: Audrow Nash, Jacob Perron

rosidl_generator_py

- Hides the assertions that checks the data types of the message fields. (#194)
- Service introspection (#178)
- [rolling] Update maintainers - 2022-11-07 (#189)
- Remove stray numpy import (#185)
- man_farmer Fix NaN values bound numpy windows version (#182)
• Allow NaN values to pass floating point bounds check. (#167)
• Replace rosidl_cmake imports with rosidl_pycommon (#177)
• Change decode error mode to replace (#176)
• Merge pull request #173 from ros2/quarkytale/fix_import_order
• fix flake
• sorting after conversion
• Revert “Use modern cmake targets to avoid absolute paths to appear in binary archives (#160)” (#166)
• Use modern cmake targets to avoid absolute paths to appear in binary archives (#160)
• michel as author
• adding maintainer
• Contributors: Audrow Nash, Ben Wolsieffer, Brian, Cristóbal Arroyo, Dharini Dutia, Eloy Briceno, Ivan Santiago Paunovic, Jacob Perron, Tomoya Fujita, quarkytale, Øystein Sture

rosidl_generator_tests

• Type Description Codegen and Typesupport (rep2011) (#727)
• Type hash in interface codegen (rep2011) (#722)
• [service introspection] generate service_event messages (#700) * add service event message
• [rolling] Update maintainers - 2022-11-07 (#717)
• Move rosidl_generator_c/cpp tests to a separate package (#701)
• Contributors: Audrow Nash, Brian, Emerson Knapp, Jacob Perron

rosidl_generator_type_description

• Type Description Codegen and Typesupport (rep2011) (#727)
• Expose type hash on typesupports (rep2011) (#729)
• Type hash in interface codegen (rep2011) (#722)
• Contributors: Emerson Knapp

rosidl_parser

• [service introspection] generate service_event messages (#700)
• [rolling] Update maintainers - 2022-11-07 (#717)
• Always include whitespace in string literals (#688)
• Contributors: Audrow Nash, Brian, Shane Loretz
rosidl_pycommon

- Type Description Codegen and Typesupport (rep2011) (#727)
- Type hash in interface codegen (rep2011) (#722)
- [rolling] Update maintainers - 2022-11-07 (#717)
- Move rosidl_cmake Python module to a new package rosidl_pycommon (#696) Deprecate the Python module in rosidl_cmake and move the implementation to the new package rosidl_pycommon.
- Contributors: Audrow Nash, Emerson Knapp, Jacob Perron

rosidl_runtime_c

- Dynamic Subscription (BONUS: Allocators): rosidl (#737)
- Runtime Interface Reflection: rosidl (#728)
- Type Description Codegen and Typesupport (rep2011) (#727)
- Copied type_description_interfaces structs (rep2011) (#732)
- Expose type hash on typesupports (rep2011) (#729)
- Type hash in interface codegen (rep2011) (#722)
- [service introspection] generate service_event messages (#700)
- [rolling] Update maintainers - 2022-11-07 (#717)
- Contributors: Audrow Nash, Brian, Emerson Knapp, methylDragon

rosidl_runtime_cpp

- Type Description Codegen and Typesupport (rep2011) (#727)
- Copied type_description_interfaces structs (rep2011) (#732)
- Fix a few more clang analysis problems. (#731)
- Return reference from BoundedVector::emplace_back (#730)
- [service introspection] generate service_event messages (#700)
- [rolling] Update maintainers - 2022-11-07 (#717)
- fix conversion to ‘std::streamsize’ {aka ‘long int’} from ‘size_t’ {aka ‘long unsigned int’} may change the sign of the result (#715)
- Contributors: Alexander Hans, Audrow Nash, Brian, Chris Lalancette, Emerson Knapp, ralwing
rosidl_runtime_py

- Replace the use __slots__ for the appropriate API (#23)
- fix(typing): get_interface_packages returns a dict (#22)
- [rolling] Update maintainers - 2022-11-07 (#21)
- Expand timestamps for std_msgs.msg.Header and builtin_interfaces.msg.Time if ‘auto’ and ‘now’ are passed as values (#19)
- Document a missing parameter in message_to_yaml. (#18)
- Mirror rolling to master
- Contributors: Audrow Nash, Chris Lalancette, Eloy Briceno, Esteve Fernandez,

rosidl_typesupport_c

- Type Description Nested Support (#141)
- Fix rosidl_typesupport_c/cpp exec dependencies. (#140)
- Type hashes in typesupport (rep2011) (#135)
- Mark benchmark _ as UNUSED. (#134)
- Service introspection (#127)
- Update rosidl_typesupport to C++17. (#131)
- [rolling] Update maintainers - 2022-11-07 (#130)
- Replace rosidl_cmake imports with rosidl_pycommon (#126)
- [service introspection] Use stddef.h instead of cstddef (#125)
- Contributors: Audrow Nash, Brian, Chris Lalancette, Emerson Knapp, Jacob Perron

rosidl_typesupport_cpp

- Type Description Nested Support (#141)
- Fix rosidl_typesupport_c/cpp exec dependencies. (#140)
- Type hashes in typesupport (rep2011) (#135)
- Mark benchmark _ as UNUSED. (#134)
- Service introspection (#127)
- Update rosidl_typesupport to C++17. (#131)
- [rolling] Update maintainers - 2022-11-07 (#130)
- Replace rosidl_cmake imports with rosidl_pycommon (#126)
- Contributors: Audrow Nash, Brian, Chris Lalancette, Emerson Knapp, Jacob Perron
rosidl_typesupport_fastrtps_c

- Type Description Nested Support (#101)
- Type hashes on typesupport (rep2011) (#98)
- Expose type hash to typesupport structs (rep2011) (#95)
- Mark benchmark _ as UNUSED. (#96)
- Service introspection (#92)
- Update rosidl_typesupport_fastrtps to C++17. (#94)
- [rolling] Update maintainers - 2022-11-07 (#93)
- Replace rosidl_cmake imports with rosidl_pycommon (#91)
- Contributors: Audrow Nash, Brian, Chris Lalancette, Emerson Knapp, Jacob Perron

rosidl_typesupport_fastrtps_cpp

- Type Description Nested Support (#101)
- Type hashes on typesupport (rep2011) (#98)
- Depend on ament_cmake_ros to default SHARED to ON (#99)
- Expose type hash to typesupport structs (rep2011) (#95)
- Mark benchmark _ as UNUSED. (#96)
- Service introspection (#92)
- Update rosidl_typesupport_fastrtps to C++17. (#94)
- [rolling] Update maintainers - 2022-11-07 (#93)
- Replace rosidl_cmake imports with rosidl_pycommon (#91)
- Contributors: Audrow Nash, Brian, Chris Lalancette, Emerson Knapp, Jacob Perron, Tyler Weaver

rosidl_typesupport_interface

- [service introspection] generate service_event messages (#700)
- [rolling] Update maintainers - 2022-11-07 (#717)
- Contributors: Audrow Nash, Brian

rosidl_typesupport_introspection_c

- Type Description Codegen and Typesupport (rep2011) (#727)
- Expose type hash on typesupports (rep2011) (#729)
- Type hash in interface codegen (rep2011) (#722)
- [service introspection] generate service_event messages (#700)
- [rolling] Update maintainers - 2022-11-07 (#717)
• Move rosidl_cmake Python module to a new package rosidl_pycommon (#696) Deprecate the Python module in rosidl_cmake and move the implementation to the new package rosidl_pycommon.
• Fix build export dependencies in C introspection package (#695)
• Add namespaced ALIAS target to easily consume generated libraries via add_subdirectory (#605)
• Contributors: Audrow Nash, Brian, Emerson Knapp, Jacob Perron, Silvio Traversaro

rosidl_typesupport_introspection_cpp

• Type Description Codegen and Typesupport (rep2011) (#727)
• Expose type hash on typesupports (rep2011) (#729)
• Type hash in interface codegen (rep2011) (#722)
• Make sure to add the event message to typesupport introspection cpp. (#724)
• [service introspection] generate service_event messages (#700)
• [rolling] Update maintainers - 2022-11-07 (#717)
• Move rosidl_cmake Python module to a new package rosidl_pycommon (#696) Deprecate the Python module in rosidl_cmake and move the implementation to the new package rosidl_pycommon.
• Add namespaced ALIAS target to easily consume generated libraries via add_subdirectory (#605)
• Contributors: Audrow Nash, Brian, Chris Lalancette, Emerson Knapp, Jacob Perron, Silvio Traversaro

rosidl_typesupport_introspection_tests

• Fix a few more clang analysis problems. (#731) In particular, make sure to mark the fact that we are C++17 (as the emplace_back signature changed), and also add in a few more (void) for benchmark tests.
• [service introspection] generate service_event messages (#700)
• [rolling] Update maintainers - 2022-11-07 (#717)
• Contributors: Audrow Nash, Brian, Chris Lalancette

rosidl_typesupport_tests

• typesupport_tests needs to be updated to C++17 (#137)
• Fix Typesupport Introspection tests (#133)
• Make rosidl_typesupport_tests depend on rosidl_generator_cpp. (#132)
• Service introspection (#127)
• Contributors: Brian, Chris Lalancette, Cristóbal Arroyo, Lucas Wendland
Vulcanexus Documentation, Release 1.0.0

rpyutils

- [rolling] Update maintainers - 2022-11-07 (#10)
- Mirror rolling to master
- updating maintainer
- Contributors: Audrow Nash, Dharini Dutia

rqt

- fix build of rqt with setuptools>=v61.0.0 (#271)
- [rolling] Update maintainers - 2022-11-07 (#283)
- Fix up the package description. (#250)
- Contributors: Audrow Nash, Chris Lalancette, Daniel Reuter, Dharini Dutia, quarkytale

rqt_action

- [rolling] Update maintainers - 2022-11-07 (#14)
- Small cleanups to the rqt_action plugin (#13)
- Mirror rolling to ros2
- Contributors: Audrow Nash, Chris Lalancette

rqt_bag

- Use default storage id (#140)
- Use rosbag2_py API instead of direct bag parsing
- [rolling] Update maintainers - 2022-11-07 (#132)
- For get_entry_after, bump by 1 nanosecond otherwise always get the same message equal to the timestamp
- Use rosbag2_py.reader for all message queries, remove sqlite3 direct usage
- Cleanup for review
- Improved logging
- Use a rosbag2_py.Reader to get bag metadata
- Disable reading from bag while recording - use direct caching to index for timeline
- Increase publishing checkbox size (#122)
- Fix toggle thumbnails button (#117)
- ensure data types match what PyQt expects (#118)
- Visualize topics being published and highlight topic being selected (#116)
- Be able to scroll up and down, not only zoom-in and out the timeline (#114)
- [Fixes] Fix crash when no qos metadata, make scroll bar appear if needed, add gitignore (#113)
- Fix the types being passed into QFont and QColor. (#109)
• Fix tuples for bisect calls (#67) (#76)
• fix long topic names (#114)
• fix zoom behavior (#76)
• Contributors: Audrow Nash, Chris Lalancette, Emerson Knapp, Ivan Santiago Paunovic, Kenji Brameld, Yadunund

**rqt_bag_plugins**

• Changes the use of `__slots__` for the field and field type getter (#138)
• [rolling] Update maintainers - 2022-11-07 (#132)
• Contributors: Audrow Nash, Eloy Briceno

**rqt_console**

• [rolling] Update maintainers - 2022-11-07 (#39)
• added new maintainer
• Contributors: Arne Hitzmann, Audrow Nash

**rqt_graph**

• Refresh rosgraph when params checkbox is clicked (#87)
• [rolling] Update maintainers - 2022-11-07 (#83)
• Minor cleanup (#80)
• Mirror rolling to galactic-devel
• graph load/save into DOT file corrections for py3 (#78)
• Remove repeated prefixes from buttons
• Contributors: Audrow Nash, Chris Lalancette, David V. Lu!!, Yadunund, mergify[bot]

**rqt_gui**

• [rolling] Update maintainers - 2022-11-07 (#283)
• Display basic help information when no plugins are loaded (#268)
• Contributors: Audrow Nash, Dharini Dutia, Michael Jeronimo, quarkytale
**rqt_gui_cpp**

- Update rqt to C++17. (#285)
- [rolling] Update maintainers - 2022-11-07 (#283)
- Contributors: Audrow Nash, Chris Lalancette, Dharini Dutia, quarkytale

**rqt_gui_py**

- Fix an exception raised when terminating with Ctrl+c (#292)
- [rolling] Update maintainers - 2022-11-07 (#283)
- Contributors: Audrow Nash, Chen Lihui, Dharini Dutia, quarkytale

**rqt_msg**

- [rolling] Update maintainers - 2022-11-07 (#17)
- Contributors: Audrow Nash

**rqt_plot**

- Fix regression from #87 (#91)
- Changes the use of __slots__ for the field and field type getter (#87)
- [rolling] Update maintainers - 2022-11-07 (#83)
- Fix fixed-size Array visualization (#81)
- Contributors: Audrow Nash, Eloy Briceno, Jacob Perron, Michael Jeronimo, Yadunund

**rqt_publisher**

- Changes the use of __slots__ for the field and field type getter
- [rolling] Update maintainers - 2022-11-07 (#36)
- Minor cleanups in rqt_publisher for ROS 2 (#35)
- Delete sync to foxy-devel workflow
- Merge pull request #33 from NBadyal/improve-evaluation-of-types
- Use regex matching to strip errors from input
- Change slot_type verification strategy
- Mirror rolling to foxy-devel
- Contributors: Audrow Nash, Chris Lalancette, Geoffrey Biggs, Michael Jeronimo, Nicholas Badyal, Voldivh
rqt_py_common

- Changes the use of __slots__ for the field and field type getter (#289)
- [rolling] Update maintainers - 2022-11-07 (#283)
- Contributors: Audrow Nash, Dharini Dutia, Eloy Briceno, quarkytale

rqt_py_console

- [rolling] Update maintainers - 2022-11-07 (#13)
- Contributors: Audrow Nash, Jacob Perron

rqt_reconfigure

- reorder imports to fix flake8 warning (#129)
- Fixed validator locale when float value is not bound in a range. (#121)
- get parameter type from descriptor
- [rolling] Update maintainers - 2022-11-07 (#122)
- Cleanup mislabeled BSD license (#66)
- Add support for array types (#108)
- Fix float slider step size (#117)
- update maintainer
- Fixed package to run with ros2 run (#81)
- fix updating range limits (#108)
- Improvement; “GUI hangs for awhile or completely, when any one of nodes doesn’t return any value” (#81)
- Contributors: Aris Synodinos, Audrow Nash, Christian Rauch, Dharini Dutia, Florian Vahl, Jacob Perron, Shrijit Singh, Tully Foote, quarkytale

rqt_service_caller

- Contributors: Audrow Nash, Jacob Perron

rqt_shell

- [rolling] Update maintainers - 2022-11-07 (#17)
- Contributors: Audrow Nash, Jacob Perron
rqt_srv

- [rolling] Update maintainers - 2022-11-07 (#10)
- Contributors: Audrow Nash, Jacob Perron

rqt_topic

- [rolling] Update maintainers - 2022-11-07 (#43)
- Implement bandwidth monitoring (#40)
- Fix the display of array type elements. (#41)
- Fix removal of topics while they are being monitored. (#39)
- Contributors: Audrow Nash, Chris Lalancette, Jacob Perron

rti_connext_dds_cmake_module

- Use unified approach for checking the existence of environment variables (#117)
- Contributors: Christopher Wecht

rttest

- [rolling] Update maintainers - 2022-11-07 (#121)
- Addressing issues found in Humble testing (#116)
- Contributors: Audrow Nash, Michael Carroll

rviz2

- Make rviz1_to_rviz2.py accept configs with missing values (#945)
- Update rviz to C++17. (#939)
- [rolling] Update maintainers - 2022-11-07 (#923)
- Add rviz1_to_rviz2.py conversion script (#882)
- Contributors: Audrow Nash, Chris Lalancette, Shane Loretz

rviz_assimp_vendor

- If vendored assimp is present, always prefer that (#970)
- [rolling] Update maintainers - 2022-11-07 (#923)
- Fixes policy CMP0135 warning for CMake >= 3.24 (#898)
- Contributors: Audrow Nash, Cristóbal Arroyo, Scott K Logan
rviz_common

- Update Frame shortcut (#958) * Update Frame shortcut
- Update rviz to C++17. (#939)
- [rolling] Update maintainers - 2022-11-07 (#923)
- Remove YAML_CPP_DLL define (#831)
- Document getTransform() time behavior (#893)
- Ogre 1.12.10 upgrade (#878)
- Add RVIZ_COMMON_PUBLIC macro (#865)
- Add time jump handler (#752) (#791)
- Make sure not to dereference a null Renderable pointer. (#850)
- Contributors: Akash, Audrow Nash, Chris Lalancette, David V. Lu!!, Kenji Brameld, Marcel Zeilinger, Shane Loretz, juchajam

rviz_default_plugins

- Fix ODR errors with gmock (#967)
- Update Frame shortcut (#958)
- point_marker: fix bug where the number of rendered points accumulates over time (#949)
- Update rviz to C++17. (#939)
- Fix tolerance calculation precision (#934)
- Fix MeshResourceMarker for mesh with color-based embedded material (#928)
- [rolling] Update maintainers - 2022-11-07 (#923)
- Add Map Display binary option (#846)
- Delete frame_locked_markers when reusing marker (#907)
- Consider region of interest in CameraDisplay (#864)
- std::copy fix - OccupancyGridUpdate - Data is not being processed correctly (#895)
- Set error status when duplicate markers are in the same MarkerArray (#891)
- Make Axes display use latest transform (#892)
- Show link names in inertia error message (#874)
- Ogre 1.12.10 upgrade (#878)
- Use make_shared to construct PointCloud2 (#869)
- Fix include order (#858)
- Contributors: AndreasR30, Audrow Nash, Chris Lalancette, David V. Lu!!, Eric, Hunter L. Allen, Jacob Perron, Kenji Brameld, Patrick Roncaglio, Shane Loretz, Timon Engelke, Xavier BROQUERE, Xenofon Karamanos, methylDragon
rviz_ogre_vendor

- Fix build failures on macOS + Apple Silicon (#944)
- [rolling] Update maintainers - 2022-11-07 (#923)
- Remove broken rviz_ogre_vendor::RenderSystem_GL target (#920)
- Fixes policy CMP0135 warning for CMake >= 3.24 (#898)
- Ogre 1.12.10 upgrade (#878)
- Make resource file paths relative (#862)
- Use CMAKE_STAGING_PREFIX for staging OGRE installation (#861)
- Contributors: Audrow Nash, Cristóbal Arroyo, Kenji Brameld, Scott K Logan, Shane Loretz, Yadu

rviz_rendering

- [rolling] Update maintainers - 2022-11-07 (#923)
- add test to ensure binary STL files from SOLIDWORKS get imported without a warning (#917)
- Ogre 1.12.10 upgrade (#878)
- Stop using glsl150 resources for now. (#851)
- Contributors: Audrow Nash, Chris Lalancette, Kenji Brameld

rviz_rendering_tests

- [rolling] Update maintainers - 2022-11-07 (#923)
- add test to ensure binary STL files from SOLIDWORKS get imported without a warning (#917)
- Contributors: Audrow Nash, Kenji Brameld

rviz_visual_testing_framework

- Update rviz to C++17. (#939)
- [rolling] Update maintainers - 2022-11-07 (#923)
- Ogre 1.12.10 upgrade (#878)
- Contributors: Audrow Nash, Chris Lalancette, Kenji Brameld

sensor_msgs

- update YUV format codes and documentation (#214)
- sensor_msgs/Range lacks variance field (#181)
- Update common_interfaces to C++17. (#215)
- [rolling] Update maintainers - 2022-11-07 (#210)
- Replaced non-ASCII dash symbol with ASCII dash (#208)
• Add NV21 and NV24 to colour formats (#205)
• Update BatteryState.msg (#206)
• use regex for matching cv types (#202)
• Fix outdated file path for image_encodings (#200)
• Use uint32_t for pointcloud2 resize method (#195)
• Retain width and height after resize for master (#193)
• Contributors: Audrow Nash, Borong Yuan, Chris Lalancette, Christian Rauch, El Jawad Alaa, Geoffrey Biggs, Ivan Zatevakhin, Kenji Brameld, Tianyu Li

sensor_msgs_py

• Add missing dep for sensor_msgs_py (#217)
• [rolling] Update maintainers - 2022-11-07 (#210)
• Add support for non standard point step sizes (#199)
• Remove reference to old implementation (#198)
• Contributors: Audrow Nash, Florian Vahl, Yadu

service_msgs

• Update common_interfaces to C++17. (#215) (#151)
• Add service_msgs package (#143)
• Contributors: Brian, Chris Lalancette

shape_msgs

• Update common_interfaces to C++17. (#215)
• [rolling] Update maintainers - 2022-11-07 (#210)
• Fix SolidPrimitive.msg to contain a single Polygon (#189)
• Contributors: Audrow Nash, Chris Lalancette, M. Fatih Cırıt

shared_queues_vendor

• Add Michael Orlov as maintainer in rosbag2 packages (#1215)
• Fixes policy CMP0135 warning for CMake >= 3.24 (#1084)
• Contributors: Cristóbal Arroyo, Michael Orlov
Vulcanexus Documentation, Release 1.0.0

**spdlog_vendor**

- Update to spdlog 1.9.2 (#33)
- [rolling] Update maintainers - 2022-11-07 (#31)
- Update to spdlog 1.9.1 (#27)
  - Fixes policy CMP0135 warning for CMake >= 3.24 (#30)
  - build shared lib only if BUILD_SHARED_LIBS is set (#29)
- Mirror rolling to master
- xml tag order
- updating maintainer
- Contributors: Audrow Nash, Chris Lalancette, Cristóbal Arroyo, Dharini Dutia, Scott K Logan, hannes09

**sqlite3_vendor**

- Update to sqlite3 3.37.2 (#1274) This matches version distributed in Ubuntu Jammy.
- Add Michael Orlov as maintainer in rosbag2 packages (#1215)
- Fixes policy CMP0135 warning for CMake >= 3.24 (#1084)
- Contributors: Cristóbal Arroyo, Michael Orlov, Scott K Logan

**sros2**

- Fix SSH commands in SROS2_Linux.md (#286)
- Make type of get_package_share_directory apparent for sphinx (#284)
- Contributors: Boris Boutillier, Yadu

**statistics_msgs**

- Update common_interfaces to C++17. (#215) (#151)
- [rolling] Update maintainers - 2022-11-07 (#150)
- Contributors: Audrow Nash, Chris Lalancette

**std_msgs**

- Update common_interfaces to C++17. (#215)
- [rolling] Update maintainers - 2022-11-07 (#210)
- Contributors: Audrow Nash, Chris Lalancette
std_srvs

- Update common Interfaces to C++17. (#215)
- [rolling] Update maintainers - 2022-11-07 (#210)
- Contributors: Audrow Nash, Chris Lalancette

stereo_msgs

- Update common Interfaces to C++17. (#215)
- [rolling] Update maintainers - 2022-11-07 (#210)
- Contributors: Audrow Nash, Chris Lalancette

tango_icons_vendor

- [rolling] Update maintainers - 2022-11-07 (#10)
- Mirror rolling to master
- Contributors: Audrow Nash

test_cli

- Update the system tests to C++17. (#510)
- [rolling] Update maintainers - 2022-11-07 (#509)
- Contributors: Audrow Nash, Chris Lalancette

test_cli_remapping

- Update the system tests to C++17. (#510)
- [rolling] Update maintainers - 2022-11-07 (#509)
- Contributors: Audrow Nash, Chris Lalancette

test_communication

- Update the system tests to C++17. (#510)
- [rolling] Update maintainers - 2022-11-07 (#509)
- Revert “Replace deprecated spin_until_future_complete with spin_until_complete (#499)” (#504)
- Replace deprecated spin_until_future_complete with spin_until_complete (#499)
- Contributors: Audrow Nash, Chris Lalancette, Hubert Liberacki, William Woodall
test_interface_files

- [rolling] Update maintainers - 2022-11-07 (#21)
- Mirror rolling to master
- Contributors: Audrow Nash

test_launch_ros

- Enable document generation using rosdoc2 (#359)
- Fix normalize_parameters_dict for multiple nodes in the same namespace (#347)
- Implement None check for ComposableNodeContainer (#341)
- Add LifecycleTransition action (#317)
- Ensure load_composable_nodes respects condition (#339)
- [rolling] Update maintainers - 2022-11-07 (#331)
- RosTimer -> ROSTimer and PushRosNamespace -> PushROSNamespace, to follow PEP8 (#326)
- add SetROSLogDir action (#325)
- Support default values in parameter substitution (#313)
- Run condition for composable nodes (#311)
- Load composable nodes in sequence (#315)
- Contributors: Aditya Pande, Alexey Merzlyakov, Audrow Nash, Christoph Hellmann Santos, Kenji Miyake, Shane Loretz, William Woodall, Yadu, methylDragon

test_launch_testing

- [rolling] Update maintainers - 2022-11-07 (#671)
- Contributors: Audrow Nash

test_msgs

- Update common_interfaces to C++17. (#215) (#151)
- [rolling] Update maintainers - 2022-11-07 (#150)
- Depend on rosidl_core_generators for packages required by actions (#144)
- Make the functions in the header static inline (#140)
- Contributors: Audrow Nash, Chris Lalancette, Jacob Perron
**test_osrf_testing_tools_cpp**

- Changing C++ Compile Version (#76)
- Update maintainers (#74)
- Contributors: Audrow Nash, Lucas Wendland

**test_quality_of_service**

- Fix ODR errors with gtest (#514)
- Avoid flaky test (#513)
- Update the system tests to C++17. (#510)
- [rolling] Update maintainers - 2022-11-07 (#509)
- Pass rclcpp::QoS to create_service (#507)
- Pass rclcpp::QoS to create_client (#506)
- Remove Werror from test_quality_of_service. (#503)
- Revert “Replace deprecated spin_until_future_complete with spin_until_complete (#499)” (#504)
- Replace deprecated spin_until_future_complete with spin_until_complete (#499)
- Add tests for ‘best available’ QoS policies (#501)
- Contributors: Audrow Nash, Chen Lihui, Chris Lalancette, Hubert Liberacki, Jacob Perron, Shane Loretz, William Woodall, methylDragon

**test_rclcpp**

- Update the system tests to C++17. (#510)
- [rolling] Update maintainers - 2022-11-07 (#509)
- Pass rclcpp::QoS to create_service (#507)
- Pass rclcpp::QoS to create_client (#506)
- Revert “Replace deprecated spin_until_future_complete with spin_until_complete (#499)” (#504)
- Replace deprecated spin_until_future_complete with spin_until_complete (#499)
- Contributors: Audrow Nash, Chris Lalancette, Hubert Liberacki, Shane Loretz, William Woodall

**test_rmw_implementation**

- Add tests for rmw matched event (#216)
- Update rmw_implementation to C++17. (#214)
- [rolling] Update maintainers - 2022-11-07 (#212)
- Add rmw_get_gid_for_client & tests (#206)
- Contributors: Audrow Nash, Barry Xu, Brian, Chris Lalancette
test_ros2trace

- Move ros2trace tests to new test_ros2trace package (#63)
- Contributors: Christophe Bedard

test_security

- [rolling] Update maintainers - 2022-11-07 (#509)
- Contributors: Audrow Nash

test_tf2

- Update the demos to C++17. (#578)
- Update maintainers (#560)
- Contributors: Audrow Nash, Chris Lalancette

test_tracetools

- Disable tracing on Android (#72)
- Add intra-process tracepoints (#30)
- Allow requiring minimum lttng package version for is_lttng_installed (#59)
- Disable tracing on macOS (#53)
- Include tracepoints by default on Linux (#31)
- Fix memory leak in tracetools::get_symbol() (#43)
- Update tracing to C++17. (#33)
- Contributors: Chris Lalancette, Christophe Bedard, Przemysław Dąbrowski, ymski

test_tracetools_launch

- Error out if trace already exists unless ‘append’ option is used (#58)
- Make subbuffer size configurable with Trace action (#51)
- Allow requiring minimum lttng package version for is_lttng_installed (#59)
- Enable document generation using rosdoc2 for ament_python pkgs (#50)
- Contributors: Christophe Bedard, Christopher Wecht, Yadu
tf2

- Fix error code returned in BufferCore::walkToTopParent (#602)
- Depend on ament_cmake_ros to default SHARED to ON (#591)
- Fix a potential crash in TimeCache::findClosest (#592)
- Extend TimeCache API to provide rich ExtrapolationException infos (#586)
- Update geometry2 to C++17 (#584)
- Include required header Scalar.h (#559)
- Update maintainers (#560)
- Contributors: Audrow Nash, Chris Lalancette, Patrick Roncagliolo, Shane Loretz, Tyler Weaver

tf2_bullet

- Update the demos to C++17. (#578)
- Update maintainers (#560)
- Contributors: Audrow Nash, Chris Lalancette

tf2_eigen

- Update the demos to C++17. (#578)
- Update maintainers (#560)
- Contributors: Audrow Nash, Chris Lalancette

tf2_eigen_kdl

- Update geometry2 to C++17 (#584)
- Update maintainers (#560)
- Use orocos_kdl_vendor and orocos-kdl target (#534)
- Contributors: Audrow Nash, Chris Lalancette, Scott K Logan

tf2_geometry_msgs

- Add do_transform_polygon_stamped (#582)
- Update the demos to C++17. (#578)
- Update maintainers (#560)
- Add torque due to force offset (#538)
- Use orocos_kdl_vendor and orocos-kdl target (#534)
- Contributors: Audrow Nash, Chris Lalancette, Paul Gesel, Scott K Logan, Tony Najjar
tf2_kdl

- Update the demos to C++17. (#578)
- Update maintainers (#560)
- Use orocos_kdl_vendor and orocos-kdl target (#534)
- Contributors: Audrow Nash, Chris Lalancette, Scott K Logan

tf2_msgs

- Update geometry2 to C++17 (#584)
- Update maintainers (#560)
- Remove action_msgs dependency (#547)
- Contributors: Audrow Nash, Chris Lalancette, Jacob Perron

tf2_py

- Update geometry2 to C++17 (#584)
- Update maintainers (#560)
- Contributors: Audrow Nash, Chris Lalancette

tf2_ros

- Destroy callback group before node (#595)
- Enable TransformListener node-based constructor in Intra-process enabled components (#572)
- Fix use-after-free bug in BufferServer::cancelCB (#579)
- Update the demos to C++17. (#578)
- add constructor to static tf broadcaster accepting node interfaces (#576)
- Update maintainers (#560)
- Switching from sstream to c string formatting to fix ros arg issue (#557)
- allow construction of tf broadcaster from node object (not a pointer) (#555)
- Allow to construct TransformBroadcaster and TransformListener from node interfaces (#552)
- Suppress spam from calling canTransform (#529)
- Contributors: Alberto Soragna, Alexander Hans, Audrow Nash, Chris Lalancette, Gonzo, Michael Carroll, Patrick Roncagliolo
tf2_ros_py

- Update sys.path with working directory (#594)
- Enable document generation using rosdoc2 for ament_python pkgs (#587)
- Update maintainers (#560)
- Use pytest rather than unittest to enable repeat (#558)
- Contributors: Audrow Nash, Michael Carroll, Yadu

tf2_sensor_msgs

- Update the demos to C++17. (#578)
- Update maintainers (#560)
- feat: export tf2 sensor msgs target (#536)
- tf2_sensor_msgs find the right Python executable. (#525)
- Add missing ament_cmake_pytest package needed because of newly-enabled test (#520)
- Port point cloud transformation to numpy (#507)
- Contributors: Audrow Nash, Chris Lalancette, Daisuke Nishimatsu, Florian Vahl, Jorge Perez, Michael Jeronimo

tf2_tools

- Enable document generation using rosdoc2 for ament_python pkgs (#587)
- Update maintainers (#560)
- Contributors: Audrow Nash, Yadu

tlsf

- [rolling] Update maintainers - 2022-11-07 (#13)
- Update maintainers (#12)
- Contributors: Audrow Nash, methylDragon

tlsf_cpp

- Update realtime support to C++17. (#122)
- [rolling] Update maintainers - 2022-11-07 (#121)
- Addressing issues found in Humble testing (#116)
- Contributors: Audrow Nash, Chris Lalancette, Michael Carroll
topic_monitor

- update launch file name format to match documentation (#588)
- [rolling] Update maintainers - 2022-11-07 (#589)
- Contributors: Audrow Nash, Patrick Wspanialy

topic_statistics_demo

- Update the demos to C++17. (#594)
- [rolling] Update maintainers - 2022-11-07 (#589)
- Contributors: Audrow Nash, Chris Lalancette

tracetools

- Disable tracing on Android (#72)
- Add intra-process tracepoints (#30)
- Improve tracetools rosdoo2/doxygen output (#57)
- Update README and other documentation (#55)
- Disable tracing on macOS (#53)
- Include tracepoints by default on Linux (#31)
- Explicitly link against dl for dladdr() (#48)
- Fix memory leak in tracetools::get_symbol() (#43)
- Add TRACEPOINT_ENABLED() and DO_TRACEPOINT() macros (#46)
- Update tracing to C++17. (#33)
- Add new rclepp_subscription_init tracepoint to support new intra-process comms
- Contributors: Chris Lalancette, Christophe Bedard, Przemysław Dąbrowski, ymski

tracetools_launch

- Error out if trace already exists unless ‘append’ option is used (#58)
- Improve ‘ros2 trace’ command error handling & add end-to-end tests (#54)
- Make subbuffer size configurable with Trace action (#51)
- Enable document generation using rosdoo2 for ament_python pkgs (#50)
- Remove deprecated context_names parameter (#38)
- Contributors: Christophe Bedard, Christopher Wecht, Yadu
tracetools_trace

- Error out if trace already exists unless ‘append’ option is used (#58)
- Improve ‘ros2 trace’ command error handling & add end-to-end tests (#54)
- Make subbuffer size configurable with Trace action (#51)
- Add intra-process tracepoints (#30)
- Allow requiring minimum lttng package version for is_lttng_installed (#59)
- Include tracepoints by default on Linux (#31)
- Enable document generation using rosdoc2 for ament_python pkgs (#50)
- Remove deprecated context_names parameter (#38)
- Contributors: Christophe Bedard, Christopher Wecht, Yadu, ymski

trajectory_msgs

- Update common_interfaces to C++17. (#215)
- [rolling] Update maintainers - 2022-11-07 (#210)
- Contributors: Audrow Nash, Chris Lalancette

turtlesim

- Remove the range constraints from the holonomic parameter. (#150) (#151)
- Add icon (#148) (#149)
- Update turtlesim to C++17. (#146)
- Add parameter to enable holonomic motion (#131)
- Add humble turtle (#140)
- Contributors: Audrow Nash, Chris Lalancette, Daisuke Sato, mergify[bot]

type_description_interfaces

- Add GetTypeDescription.srv (rep2011) (#153)
- new package and interfaces for describing other types (#146)
- Contributors: Emerson Knapp, William Woodall
unique_identifier_msgs

- [rolling] Update maintainers - 2022-11-07 (#26)
- Depend on rosidl_core instead of rosidl_defaults (#24)
- Mirror rolling to master
- Update maintainers (#22)
- Contributors: Audrow Nash, Jacob Perron, methylDragon

urdf

- [urdf] package.xml: add missing exec_depend to urdf_parser_plugin (#34)
- Provide copy and move constructors for model (#33)
- Add linter tests and fix errors (#30)
- fix #30
- Contributors: Audrow Nash, Daniel Reuter, Tobias Neumann

urdf_parser_plugin

- Contributors: Audrow Nash

visualization_msgs

- Update common_interfaces to C++17. (#215)
- [rolling] Update maintainers - 2022-11-07 (#210)
- Contributors: Audrow Nash, Chris Lalancette

yaml_cpp_vendor

- [rolling] Update maintainers - 2022-11-07 (#40)
- Export YAML_CPP_DLL define on Windows (#30) (#38)
- Sets CMP0135 policy behavior to NEW (#36)
- Fixes policy CMP0135 warning for CMake >= 3.24 (#35)
- build shared lib only if BUILD_SHARED_LIBS is set (#34)
- Mirror rolling to master
- Contributors: Audrow Nash, Cristóbal Arroyo, Jacob Perron, hannes09
zstd_vendor

- Add Michael Orlov as maintainer in rosbag2 packages (#1215)
- Bump zstd to 1.4.8 in zstd_vendor package (#1132)
- Fix/zstd vendor does not find system zstd (#1111)
- Contributors: DasRoteSkelett, Michael Orlov

Table of Contents

- Supported Platforms
- Installation
- New features in this ROS 2 release
  - API documentation generation for Python packages
  - Service introspection
  - Pre and post set parameter callback support
  - Improved discovery options
  - Matched events
  - External configuration services of loggers
  - Type Description Distribution
  - Dynamic Types and Dynamic Messages
  - launch
  - launch_ros
  - launch_testing
  - rclcpp
  - rclpy
  - ros2param
  - ros2topic
- Changes since the Humble release
  - Change to the default console logging file flushing behavior
  - ament_cmake_auto
  - ament_cmake_nose
  - ament_lint
  - camera_info_manager
  - launch
  - launch_ros
  - launch_xml
  - launch_yaml
Iron Irwini is the ninth release of ROS 2. What follows is highlights of the important changes and features in Iron Irwini since the last release. For a list of all of the changes since Humble, see the long form changelog.

**Supported Platforms**

Iron Irwini is primarily supported on the following platforms:

**Tier 1 platforms:**
- Ubuntu 22.04 (Jammy): amd64 and arm64
- Windows 10 (Visual Studio 2019): amd64

**Tier 2 platforms:**
- RHEL 9: amd64

**Tier 3 platforms:**
- macOS: amd64
- Debian Bullseye: amd64

For more information about RMW implementations, compiler / interpreter versions, and system dependency versions see REP 2000.
Installation

Install Iron Irwini

New features in this ROS 2 release

API documentation generation for Python packages

ROS 2 has had automatic API documentation for C++ packages for several releases, e.g. https://docs.ros.org/en/rolling/p/rclcpp/generated/index.html. Iron adds automatic API documentation for Python packages as well, e.g. https://docs.ros.org/en/rolling/p/rclpy/rclpy.html.


Service introspection

It is now possible to enable service introspection on a per-service basis. When enabled, this allows users to see the metadata associated with the client requesting a service, the server accepting the request, the server sending the response, and the client accepting the response. Optionally, the contents of the client/server requests/responses can also be introspected. All of the information is published on a hidden topic generated from the name of the service. So if the service is called /myservice, then the information will be published on /myservice/_service_event.

Note that this functionality is disabled by default; to enable it, users must call configure_introspection after creating a server client or server. There are examples showing how to do this in https://github.com/ros2/demos/tree/iron/demo_nodes_cpp/src/services (C++) and https://github.com/ros2/demos/blob/iron/demo_nodes_py/demo_nodes_py/services/introspection.py (Python).


Pre and post set parameter callback support

For many releases now, users could register a callback to be called when parameters on a node were changed by an external entity (like ros2 param set). This callback could examine the changed parameter types and values, and reject the whole lot if one of them didn’t meet certain criteria. However, it could not modify the parameter list, nor should it have modified state (since there might be other callbacks after the set one that would reject the parameters).

This release adds in a pre and post callback. The callbacks are called in this order:

- The “pre” set parameter callback, which can modify the list of parameters based on arbitrary criteria.
- The “set” parameter callback, which cannot modify the list and should only accept or reject the parameters based on their type and value (this is the existing callback).
- The “post” set parameter callback, which can make state changes based on parameters and is only called if the previous two callbacks are successful.

There are examples of this in action in https://github.com/ros2/demos/blob/iron/demo_nodes_cpp/src/parameters/set_parameters_callback.cpp (C++) and https://github.com/ros2/demos/blob/iron/demo_nodes_py/demo_nodes_py/parameters/set_parameters_callback.py (Python).

Improved discovery options

Previous ROS 2 versions offered limited discovery options. The default behavior for DDS based RMW implementations was to discover any node reachable via multicast. It could be limited to the same machine by setting the environment variable `ROS_LOCALHOST_ONLY`, but any additional configuration required configuring the middleware directly, usually via middleware specific XML files and environment variables. ROS Iron retains the same default discovery behavior, but deprecates `ROS_LOCALHOST_ONLY` in favor of more granular options.

- **`ROS_AUTOMATIC_DISCOVERY_RANGE`** controls how far ROS nodes will try to discover each other. Valid options are:
  - `SUBNET` - The default, and for DDS-based middlewares it will discover any node reachable via multicast.
  - `LOCALHOST` - Will only try to discover other nodes on the same machine.
  - `OFF` - Will not attempt to discover any other nodes automatically, even on the same machine.
  - `SYSTEM_DEFAULT` - Will not change any discovery settings. This is useful when you already have custom settings for your middleware and don’t want ROS to change them.

- **`ROS_STATIC_PEERS`** - A semicolon (;) separated list of addresses that ROS should try to discover nodes on. This allows the user to connect to nodes on specific machines (as long as their discovery range is not set to `OFF`).

For example, you might have several robots with `ROS_AUTOMATIC_DISCOVERY_RANGE` set to `LOCALHOST` so they don’t communicate with each other. When you want to connect RViz to one of them, you add it’s address to `ROS_STATIC_PEERS` in your terminal. Now you can use ROS 2 CLI and visualization tools to interact with the robot.

See [https://github.com/ros2/ros2/issues/1359](https://github.com/ros2/ros2/issues/1359) for more information about this feature.

Matched events

In addition to QoS events, matched events can be generated when any publisher and subscription establishes or drops the connection between them. Users can provide each publisher and subscription with callback functions that are triggered by matched events and handle them in a way they see fit, similar to how messages received on a topic are handled.

- **publisher**: this event happens when it finds a subscription which matches the topic and has compatible QoS or a connected subscription is disconnected.

- **subscription**: this event happens when it finds a publisher which matches the topic and has compatible QoS or a connected publisher is disconnected.

See the tracking issue at [https://github.com/ros2/rmw/issues/330](https://github.com/ros2/rmw/issues/330) for more information.

- **C++ Demo of Matched Events**: [https://github.com/ros2/demos/blob/iron/demo_nodes_cpp/src/events/matched_event_detect.cpp](https://github.com/ros2/demos/blob/iron/demo_nodes_cpp/src/events/matched_event_detect.cpp)

- **Python Demo of Matched Events**: [https://github.com/ros2/demos/blob/iron/demo_nodes_py/demo_nodes_py/events/matched_event_detect.py](https://github.com/ros2/demos/blob/iron/demo_nodes_py/demo_nodes_py/events/matched_event_detect.py)
**External configuration services of loggers**

It is now possible to configure node logger levels remotely via a service. When the `enable_logger_service` option is enabled during node creation, the `set_logger_levels` and `get_logger_levels` services will be available.

Be advised that the `enable_logger_service` option is disabled by default, so the user needs to enable this option on node creation.

See [https://github.com/ros2/ros2/issues/1355](https://github.com/ros2/ros2/issues/1355) for more information.

**Type Description Distribution**

It is now possible to communicate information about the types of ROS 2 messages, so that systems with potentially-different types of the same name may discover their compatibility more transparently. This umbrella of capabilities, which is defined by a subset of REP-2011: Evolving Message Types, has had many parts land in Iron.

First, the introduction of the new package `type_description_interfaces` provides a common way to communicate the descriptions of ROS 2 communication interface types (msg, srv, action).

Next, a method to hash type descriptions has been decided on, the ROS Interface Hashing Standard (RIHS) - starting with the first version RIHS01. RIHS hashes are automatically calculated for all compiled ROS types at build time, and baked into the generated code so that they can be inspected. These hashes are also communicated automatically during discovery, and included in `rmw_topic_endpoint_info_t` for graph introspection queries such as `get_publishers_info_by_topic`.

The full `TypeDescription` data structure, as well as the raw source text (such as `.msg` file) that were used to generate it are now baked in by default to the message libraries, so they can be used by `typesupport` or end users. While we expect this data to provide value to most users, some users trying to minimize bytes in their install space can disable the feature when building ROS 2 Core by defining the CMake variable `ROSIIDL_GENERATOR_C_DISABLE_TYPE_DESCRIPTION_CODEGEN`.

Finally, the new service `type_description_interfaces/GetTypeName.srv` has been defined to allow nodes, on encountering an unknown RIHS type hash, to request the full definition from the node advertising that type. Work is in progress to provide this feature natively in ROS 2 Nodes, as an optional switch on node construction. This feature has not yet shipped, but is expected to be backported into Iron sometime mid-2023. Meanwhile, user nodes could implement this service independently, using the stable service interface.

See REP 2011 for the design proposal. See Type Description Distribution for tracking development on the feature set.

**Dynamic Types and Dynamic Messages**

Alongside the type description distribution feature mentioned above, is the ability to construct and access dynamically created types at runtime (i.e., dynamic types). This feature is available in Iron for Fast DDS and rcl, with new `rmw` interfaces for supporting the taking of messages as dynamic messages (i.e., messages built from or following the structure of the dynamic type).

First, utilities were introduced into `rosidl` to aid in the construction and manipulation of type descriptions.

Next, the `rosidl_dynamic_typesupport` package was written and provides a middleware-agnostic interface to construct dynamic types and dynamic messages at runtime. Types can be constructed at runtime either programmatically, or by parsing a `type_description_interfaces/TypeDescription` message.

**Note:** The `rosidl_dynamic_typesupport` library requires serialization support libraries to implement the middleware-specific dynamic type behavior. A serialization support library for Fast DDS was implemented in
Finally, to support the use of dynamic types and dynamic messages, new methods were added to rmw and rcl that support:

- The ability to obtain middleware-specific serialization support
- The ability to construct message type support at runtime that use dynamic types
- The ability to take dynamic messages using dynamic type

Work is in progress to enable the use of dynamic types to create subscriptions in the client libraries (see the rclcpp issue below), though it is uncertain when the feature will land or be backported. This will allow users to subscribe to topics whose type descriptions are only known at runtime. In the meantime, users may write their own subscriptions that subscribe to dynamic types by using the new rmw and rcl features introduced as part of this feature set.

See REP 2011 for the design proposal. See Dynamic Subscription for tracking development on the feature set, with rclcpp needing the bulk of the work.

**launch**

**PythonExpression now supports importing modules**

It is now possible to have a launch PythonExpression import modules before performing the evaluation. This can be useful for pulling in additional functionality to be used when evaluating an expression.

See https://github.com/ros2/launch/pull/655 for more information.

**ReadyToTest can be called from an event handler**

It is now possible to register an event handler that uses ReadyToTest in its output. This can be useful for doing things like downloading an asset before allowing a test to run.

See https://github.com/ros2/launch/pull/665 for more information.

**Addition of AnySubstitution and AllSubstitution**

It is now possible to specify a substitution to happen when any of the input arguments are true (AnySubstitution), or when all of the input arguments are true (AllSubstitution).

See https://github.com/ros2/launch/pull/649 for more details.

**Addition of a new substitution to get the launch logging directory**

It is now possible to use a substitution called LaunchLogDir to get the current logging directory for launch.

See https://github.com/ros2/launch/pull/652 for more details.
launch_ros

Add a LifecycleTransition action

It is now possible to send a transition signal to a lifecycle node via the new LifeCycleTransition action. See https://github.com/ros2/launch_ros/pull/317 for more information.

Add a SetROSLogDir action

It is now possible to configure the directory used for logging via the SetROSLogDir action. See https://github.com/ros2/launch_ros/pull/325 for more information.

Ability to specify a condition to a ComposableNode

It is now possible to specify a condition that must be satisfied in order for a ComposableNode to be inserted into its container. See https://github.com/ros2/launch_ros/pull/311 for more information.

launch_testing

Timeout for process startup is now configurable

Prior to this release, the ReadyToTest action would wait exactly 15 seconds for processes to start up. If the processes took longer than that, they would fail. There is now a new decorator called ready_to_test_action_timeout that allows the user to configure the amount of time to wait for the processes to start. See https://github.com/ros2/launch/pull/625 for more information.

rclcpp

Addition of a new paradigm for handling Node and LifecycleNode

The Node and LifecycleNode classes are related in that they both provide the same base set of methods (though LifecycleNode provides additional methods as well). Due to various implementation considerations, they are not derived from a common base class.

This has led to some trouble for downstream code that wants to accept either a Node or a LifecycleNode. One solution is to have two method signatures, one that accepts a Node and one that accepts a LifecycleNode. The other, recommended solution is to have a method that accepts the “node interfaces” pointers that can be accessed from both classes, e.g.

```c
void do_thing(rclcpp::node_interfaces::NodeGraphInterface graph)
{
    fprintf(stderr, "Doing a thing\n");
}

void do_thing(rclcpp::Node::SharedPtr node)
{
...
```

(continues on next page)
This works, but can get a bit unwieldy when many node interfaces are needed. To make this a bit better, there is now a new `NodeInterfaces` class that can be constructed to contain the interfaces, and then be used by other code.

There are examples on how to use this in https://github.com/ros2/rclcpp/pull/2041.

**Introduction of a new executor type: the Events Executor**

The `EventsExecutor` from iRobot has been merged into the main `rclcpp` codebase. This alternative executor implementation uses event-driven callbacks from the middleware implementations to fire callbacks at the `rclcpp` layer. In addition to the push-based model, the `EventsExecutor` also moves timer management into a separate thread, which can allow for more accurate results and lower overhead, especially with many timers.

The `EventsExecutor` has a substantial set of documentation and use-in-practice that make it a strong candidate for inclusion in the `rclcpp` codebase. For information about the initial implementation proposal as well as performance benchmarks, see https://discourse.ros.org/t/ros2-middleware-change-proposal/15863. For more information about the design, see the design PR: https://github.com/ros2/design/pull/305.

Since the API is the same, trying the `EventsExecutor` is as straightforward as replacing your current Executor implementation (eg. `SingleThreadedExecutor`):

```cpp
#include <rclcpp/experimental/executors/events_executor/events_executor.hpp>
using rclcpp::experimental::executors::EventsExecutor;

EventsExecutor executor;
executor.add_node(node);
executor.spin();
```

*Note* The `EventsExecutor` and `TimersManager` are currently in the `experimental` namespace. While it has been used as a standalone implementation for some time https://github.com/irobot-ros/events-executor, it was decided to use the `experimental` namespace for at least one release to give latitude in changing the API within the release. Use caution as it will not be subject to the same API/ABI guarantees that the non-experimental code has.

**rclpy**

**Ability to wait for another node to join the graph**

It is now possible to wait for another node to join the network graph with code like the following:

```python
node.wait_for_node('/fully_qualified_node_name')
```

See https://github.com/ros2/rclpy/pull/930 for more information.
Implementation of AsyncParameterClient

rclpy now has an AsyncParameterClient class, bringing it to feature parity with rclcpp. This class is used to perform parameter actions on a remote node without blocking the calling node.

See https://github.com/ros2/rclpy/pull/959 for more information and examples.

Subscription callbacks can now optionally get the message info

It is now possible to register for a subscription callback with a function signature that takes both the message, and the message info, like:

```python
def msg_info_cb(msg, msg_info):
    print('Message info:', msg_info)

node.create_subscription(msg_type=std_msgs.msg.String, topic='/chatter', qos_profile=10,
                         callback=msg_info_cb)
```

The message info structure contains various pieces of information like the sequence number of the message, the source and received timestamps, and the GID of the publisher.

See https://github.com/ros2/rclpy/pull/922 for more information.

Optional argument that hides assertions for messages class

All message classes now include a new optional argument that allows the hiding of assertions for each field type from the message. By default, assertions are hidden, which provides a performance improvement during runtime. In order to enable the assertions for development/debugging purposes, you are given two choices:

1. Define the environment variable `ROS_PYTHON_CHECK_FIELDS` to '1' (this would affect all the messages in your project):

   ```python
   import os
   from std_msgs.msg import String
   
   os.environ['ROS_PYTHON_CHECK_FIELDS'] = '1'
   new_message=String()
   
   2. Select the specific behavior for a single message by explicitly defining the new argument in the constructor:

   ```python
   from std_msgs.msg import String
   
   new_message=String(check_fields=True)
   
   See https://github.com/ros2/rosidl_python/pull/194 for more information.
ros2param

Option to timeout when waiting for a node with ros2 param

It is now possible to have the various ros2 param commands timeout by passing --timeout to the command.
See https://github.com/ros2/ros2cli/pull/802 for more information.

Deprecated options were removed

--output-dir and --print options with dump command have been removed.
See https://github.com/ros2/ros2cli/pull/824 for more information.

ros2topic

now as keyword for builtin_interfaces.msg.Time and auto for std_msgs.msg.Header

ros2 topic pub now allows to set a builtin_interfaces.msg.Time message to the current time via the now keyword. Similarly, a std_msgs.msg.Header message will be automatically generated when passed the keyword auto. This behavior matches that of ROS 1's rostopic (http://wiki.ros.org/ROS/YAMLCommandLine#Headers.2Ftimestamps)
Related PR: ros2/ros2cli#749

ros2 topic pub can be configured to wait a maximum amount of time

The command ros2 topic pub -w 1 will wait for at least that number of subscribers before publishing a message.
This release adds a --max-wait-time option so that the command will only wait a maximum amount of time before quitting if no subscribers are seen.
See https://github.com/ros2/ros2cli/pull/800 for more information.

ros2 topic echo can be configured to wait a maximum amount of time

The command ros2 topic echo now accepts a --timeout option, which controls the maximum amount of time that the command will wait for a publication to happen.
See https://github.com/ros2/ros2cli/pull/792 for more information.

Deprecated option was removed

--lost-messages option with echo command has been removed.
See https://github.com/ros2/ros2cli/pull/824 for more information.
Changes since the Humble release

Change to the default console logging file flushing behavior

This specifically applies to the default spdlog based logging backend in ROS 2, implemented in rcl_logging_spdlog. Log file flushing was changed to flush every time an “error” log message is used, e.g. each RCLCPP_ERROR() call, and also periodically every five seconds.

Previously, spdlog was used without configuring anything other than creating the sink for logging to a file.

We tested the change and did not find that the CPU overhead was significant, even on machines with slow disks (e.g. sd cards). However, if this change is causing you problems, you can get the old behavior by setting the RCL_LOGGING_SPDLOG_EXPERIMENTAL_OLD_FLUSHING_BEHAVIOR=1 environment variable.

Later we would like to have support for a full configuration file (see: https://github.com/ros2/rcl_logging/issues/92), giving you more flexibility in how the logging is done, but that is work that is only planned right now.

Therefore, this environment variable should be considered experimental and subject to removal without deprecation in the future, when we add config file support for the rcl_logging_spdlog logging backend.

See this pull request for more details about the change: https://github.com/ros2/rcl_logging/pull/95

ament_cmake_auto

Include dependencies are now marked as SYSTEM

When using ament_auto_add_executable or ament_auto_add_library, dependencies are now automatically added as SYSTEM. This means that warnings in the header files of the dependencies will not be reported.

See https://github.com/ament/ament_cmake/pull/385 for more details.

ament_cmake_nose

Package has been deprecated and removed

The Python nose package has long been deprecated. Since none of the open-source packages currently released into Humble or Rolling currently depend on it, this release deprecates and removes the ament wrapper around it.

See https://github.com/ament/ament_cmake/pull/415 for more information.

ament_lint

Files can be excluded from linter checks

Certain files can now be excluded from linter checks by setting the AMENT_LINT_AUTO_FILE_EXCLUDE CMake variable before calling ament_lint_auto_find_test_dependencies.

See https://github.com/ament/ament_lint/pull/386 for more information.
camera_info_manager

**Lifecycle node support**
camera_info_manager now supports lifecycle nodes in addition to regular ROS 2 nodes. See https://github.com/ros-perception/image_common/pull/190 for more information.

**launch**

**LaunchConfigurationEquals and LaunchConfigurationNotEquals are deprecated**
The LaunchConfigurationEquals and LaunchConfigurationNotEquals conditions are deprecated, and will be removed in a future release. Instead, the more universal Equals and NotEquals substitutions should be used instead. See https://github.com/ros2/launch/pull/649 for more details.

**launch_ros**

**Renamed classes which used Ros in the name to use ROS in line with PEP8**
Classes that were changed:

- `launch_ros.actions.RosTimer` -> `launch_ros.actions.ROSTimer`
- `launch_ros.actions.PushRosNamespace` -> `launch.actions.PushROSNamespace`

The old class names are still there, but will be deprecated. See https://github.com/ros2/launch_ros/pull/326 for more information.

**launch_xml**

**Expose emulate_tty to XML frontend**
It has been possible for several releases to have the launch Python code use pseudo-terminals to emulate a TTY (and hence do things like print colors). That functionality is now available in the XML frontend by passing the `emulate_tty` argument to an executable command.

See https://github.com/ros2/launch/pull/669 for more information.

**Expose sigterm_timeout and sigkill_timeout to XML frontend**
It has been possible for several releases to configure the maximum timeout value for the SIGTERM and SIGKILL signals in the launch Python code. That functionality is now available in the XML frontend by passing the `sigterm_timeout` or `sigkill_timeout` argument to an executable command.

See https://github.com/ros2/launch/pull/667 for more information.
launch_yaml

**Expose emulate_tty to YAML frontend**

It has been possible for several releases to have the `launch` Python code use pseudo-terminals to emulate a TTY (and hence do things like print colors). That functionality is now available in the YAML frontend by passing the `emulate_tty` argument to an executable command.

See [https://github.com/ros2/launch/pull/669](https://github.com/ros2/launch/pull/669) for more information.

**Expose sigterm_timeout and sigkill_timeout to YAML frontend**

It has been possible for several releases to configure the maximum timeout value for the SIGTERM and SIGKILL signals in the `launch` Python code. That functionality is now available in the YAML frontend by passing the `sigterm_timeout` or `sigkill_timeout` argument to an executable command.

See [https://github.com/ros2/launch/pull/667](https://github.com/ros2/launch/pull/667) for more information.

message_filters

**New approximate time policy**

Add in a simpler approximate time policy called `ApproximateEpsilonTime`. This time policy works like `ExactTime`, but allows timestamps being within a epsilon tolerance. See [https://github.com/ros2/message_filters/pull/84](https://github.com/ros2/message_filters/pull/84) for more information.

**New upsampling time policy**

Adds in a new time policy called `LatestTime`. It can synchronize up to 9 messages by their rates with upsampling via zero-order-hold. See [https://github.com/ros2/message_filters/pull/73](https://github.com/ros2/message_filters/pull/73) for more information.

rcl_yaml_param_parser

**Support for YAML `!!str` syntax in parameter files**

It is now possible to force the ROS parameter file parser to interpret a field as a string using the YAML `!!str` syntax. See [https://github.com/ros2/rcl/pull/999](https://github.com/ros2/rcl/pull/999) for more information.

rclcpp

**Default number of threads for multi-threaded executor has been changed**

If the user doesn’t specify otherwise, the default number of threads for the multi-threaded executor will be set to the number of CPUs on the machine. If the underlying OS doesn’t support getting this information, it will be set to 2.

See [https://github.com/ros2/rclcpp/pull/2032](https://github.com/ros2/rclcpp/pull/2032) for more information.
A warning is now printed when QoS of KEEP_LAST is specified with a depth of 0

Specifying a QoS of KEEP_LAST with a depth of 0 is a nonsensical arrangement, since the entity wouldn't be able to send or receive any data. rclcpp will now print a warning if this combination is specified, but will still continue on and let the underlying middleware choose a sane value (generally a depth of 1).

See https://github.com/ros2/rclcpp/pull/2048 for more information.

Deprecated RCLCPP_SCOPE_EXIT macro was removed

In Humble, the macro RCLCPP_SCOPE_EXIT was deprecated in favor of RCPPUTILS_SCOPE_EXIT. In Iron, the RCLCPP_SCOPE_EXIT macro has been completely removed.

rclpy

Default number of threads for multi-threaded executor has been changed

If the user doesn't specify otherwise, the default number of threads for the multi-threaded executor will be set to the number of CPUs on the machine. If the underlying OS doesn't support getting this information, it will be set to 2.

See https://github.com/ros2/rclpy/pull/1031 for more information.

A warning is now printed when QoS of KEEP_LAST is specified with a depth of 0

Specifying a QoS of KEEP_LAST with a depth of 0 is a nonsensical arrangement, since the entity wouldn't be able to send or receive any data. rclpy will now print a warning if this combination is specified, but will still continue on and let the underlying middleware choose a sane value (generally a depth of 1).

See https://github.com/ros2/rclpy/pull/1048 for more information.

Time and Duration no longer raise exception when compared to another type

It is now possible to compare rclpy.time.Time and rclpy.duration.Duration to other types without getting exceptions. If the types are not comparable, the comparison returns False. Note that this is a behavior change from previous releases.

```
print(None in [rclpy.time.Time(), rclpy.duration.Duration()])  # Prints "False" instead
```

See https://github.com/ros2/rclpy/pull/1007 for more information.
rcutils

**Improve the performance of message logging**

The code used to output a log message when RCUTILS_LOG_* or RCLCPP_* was optimized to reduce overhead. These log messages should now be more efficient, though they should still not be called at high rates. See [https://github.com/ros2/rcutils/pull/381](https://github.com/ros2/rcutils/pull/381), [https://github.com/ros2/rcutils/pull/372](https://github.com/ros2/rcutils/pull/372), [https://github.com/ros2/rcutils/pull/369](https://github.com/ros2/rcutils/pull/369), and [https://github.com/ros2/rcutils/pull/367](https://github.com/ros2/rcutils/pull/367) for more information.

**Deprecated rcutils/get_env.h header was removed**

In Humble, the header rcutils/get_env.h was deprecated in favor of rcutils/env.h. In Iron, the rcutils/get_env.h header been completely removed.

rmw

**Change the GID storage to 16 bytes**

The GID in the RMW layer is meant to be a globally unique identifier for writers in the ROS graph. Previously, this was erroneously set to 24 bytes based on a bug in an old RMW implementation. But the rmw package should define this, and all of the implementations should conform to that. Thus, this release defines it as 16 bytes (the DDS standard), and changes all implementations to use that definition.

See [https://github.com/ros2/rmw/pull/345](https://github.com/ros2/rmw/pull/345) and the (closed, but relevant) [https://github.com/ros2/rmw/pull/328](https://github.com/ros2/rmw/pull/328) for more information.

rmw_dds_common

**Change the GID storage to 16 bytes**

Along with the change in the rmw layer, change the message that sends out GID information to 16 bytes.

See [https://github.com/ros2/rmw_dds_common/pull/68](https://github.com/ros2/rmw_dds_common/pull/68) for more information.

ros2topic

**ros2 topic hz/bw/pub now respect use_sim_time**

When running under simulation, the ROS 2 ecosystem generally gets its time from a /clock topic published by the simulator (rather than using the system clock). ROS 2 nodes are typically informed of this change by setting the use_sim_time parameter on the node. The node created by the ros2 topic commands hz, bw, and pub now respect that parameter and will use simulation time as appropriate.

See [https://github.com/ros2/ros2cli/pull/754](https://github.com/ros2/ros2cli/pull/754) for more information.
rosbag2

Change default bag file type to mcap

Prior to this release, by default rosbag2 would record data into sqlite3 databases. During testing, it was found that in many cases this was not performant enough and lacked certain features desirable for offline processing.

To meet these needs, a new bag format (influenced by the original ROS 1 bag file format) called mcap was developed. This bag file format has many of the missing features from the sqlite3 file format, and should also be more performant.

This release switches to using mcap as the default file format for writing new bags. The old sqlite3 file format is still available and can be selected by the user for writing if desired. This release also allows playing back data from either the sqlite3 file format or the mcap file format.

See https://github.com/ros2/rosbag2/pull/1160 for more information.

Store message definitions in bag files with SQLite3 plugin

Now we support saving message definitions to the sqlite3 database file in the same format as we are saving it to the mcap file. This opens an opportunity for third-party tools to have the ability to deserialize rosbag2 files without having the correct version of all the original .msg files on the machine that is decoding the bag file recorded with sqlite3 plugin.


New playback and recording controls

Several pull requests have been added to enhance the user’s control over playback of bags. Pull request 960 adds the ability to play bag for a specified number of seconds. And pull request 1005 allows to play bag until specified timestamp. Another pull request 1007 adds the ability to stop playback remotely via service call. Stop will unpause player if it was in pause mode, stop playback and force exit from play() method if it was in progress.

Managing recording via service calls

There are new options to control the recording process from remote nodes. The pull request 1131 adds the ability to pause and resume recording via service calls. Another pull request 1115 adds the ability to split bags during recording by sending service call.

Filtering topics via regular expression during playback

Users sometimes need to replay only a subset of topics from recorded bags and the following two pull request adds such capability. Pull request 1034 adds a new option --topics-regex that allows filtering topics via regular expressions. The --topics-regex option accepts multiple regular expressions separated by space. And pull request 1046 adds the ability to exclude some certain topics from being replayed by providing regular expression in a new --exclude (and -x) option.
Allow plugins to register their own CLI verb arguments

Pull request 1209 adds the ability for rosbag2 plugins to register an optional Python entrypoint providing plugin-specific CLI argument values. As a result the command line option `--storage-preset-profile` for `ros2 bag record` verb will have different valid options depending on the underlying storage plugin.

Other changes

The pull request 1038 adds the ability to record any key/value pair in ‘custom’ field in metadata.yaml file. It is useful when users need to save some hardware specific id or coordinates where the recording was captured. And pull request 1180 adds an option to change the underlying node name for the recorder via providing the new command line `--node-name` option. This option might be used for creating remote distributed recording with multiple rosbag2 recorder instances. It provides the ability to send service calls for managing the recording process to the dedicated rosbag2 recorder instances.

rosidl_python

Modification of content of `__slots__` attribute

So far, the attribute `__slots__` from the python message classes, have been used as the member that contains the field names of the message. In Iron, this attribute no longer contains only the field names from the message structure, but the field names for all the class members. Therefore, users shouldn’t rely on this attribute to retrieve the field names information, instead, users should retrieve it using the method `get_field_and_field_types()`.

See https://github.com/ros2/rosidl_python/pull/194 for more information.

rviz

Map display can now be shown as binary

The RViz map display can now display the map as binary, with a settable threshold. This is useful in some cases to inspect maps or in combination with planners that have a settable threshold.

See https://github.com/ros2/rviz/pull/846 for more information.

Camera display plugin respects the ROI in the CameraInfo message

The CameraDisplay plugin now honors the region-of-interest (ROI) settings in the CameraInfo message, if it is provided. This accounts for the fact that an image was cropped by the camera driver to reduce the bandwidth.

See https://github.com/ros2/rviz/pull/864 for more information.
Binary STL files from SOLIDWORKS work without error

A change was made to the STL loader such that it accepts binary STL files from SOLIDWORKS that have the word “solid” in them. This technically violates the STL specification, but is common enough that a special case is added to handle these files.

See https://github.com/ros2/rviz/pull/917 for more information.

tracertools

Tracing instrumentation is now included by default on Linux

The ROS 2 core has had tracing instrumentation for a while now. However, it was compiled out by default. To get the instrumentation, the LTTng tracer had to be manually installed before rebuilding ROS 2 from source. In Iron, the tracing instrumentation and tracepoints are included by default; the LTTng tracer is therefore now a ROS 2 dependency.

Note that this only applies to Linux.

See https://github.com/ros2/ros2_tracing/pull/31 and https://github.com/ros2/ros2/issues/1177 for more information. See this how-to guide to remove the instrumentation (or add the instrumentation with Humble and older).

New tracepoints for rclcpp intra-process are added

New tracepoints have been added to support rclcpp intra-process communication. This allows the evaluation of the time between the message publishing and the callback start in intra-process communication.


Known Issues

- rmw_connextdds does not work with Windows Binary release packages. RTI is not longer distributing RTI ConnextDDS 6.0.1 which was used by the packaging jobs to create the binaries for Windows. Instead they now distribute RTI ConnextDDS 6.1.0 which is ABI incompatible with the generated binaries. The solution is to rely on source builds of ROS 2 and rmw_connextdds on Windows.
- sros2 on Windows requires users to downgrade the cryptography python module to cryptography==38.0.4 as discussed here.
- ros1_bridge does not work with ROS Noetic packages from upstream Ubuntu. The suggested workaround is to build ROS Noetic from sources, then build the ros1_bridge using that.

Release Timeline

- **November, 2022 - Platform decisions** REP 2000 is updated with the target platforms and major dependency versions.
- **By January, 2023 - Rolling platform shift** Build farm is updated with the new platform versions and dependency versions for Iron Irwini (if necessary).
- **Mon. April 10, 2023 - Alpha + RMW freeze** Preliminary testing and stabilization of ROS Base\(^1\) packages, and API and feature freeze for RMW provider packages.

\(^1\) The ros_base variant is described in REP 2001 (ros-base).
Mon. April 17, 2023 - Freeze  API and feature freeze for ROS Base\(^1\) packages in Rolling Ridley. Only bug fix releases should be made after this point. New packages can be released independently.

Mon. April 24, 2023 - Branch  Branch from Rolling Ridley. rosdistro is reopened for Rolling PRs for ROS Base\(^2\) packages. Iron development shifts from ros-rolling-* packages to ros-iron-* packages.

Mon. May 1, 2023 - Beta  Updated releases of ROS Desktop\(^2\) packages available. Call for general testing.

Mon. May 15, 2023 - Release Candidate  Release Candidate packages are built. Updated releases of ROS Desktop\(^2\) packages available.

Thu. May 18, 2023 - Distro Freeze  Freeze rosdistro. No PRs for Iron on the rosdistro repo will be merged (reopens after the release announcement).

Tue. May 23, 2023 - General Availability  Release announcement. rosdistro is reopened for Iron PRs.

Development progress

For progress on the development and release of Iron Irwini, see the tracking GitHub issue.

For the broad process followed by Iron Irwini, see the process description page.

Humble Hawksbill (humble)

Humble Hawksbill changelog

This page is a list of the complete changes in all ROS 2 core packages since the previous release.

Table of Contents

- action_msgs
- action_tutorials_cpp
- action_tutorials_interfaces
- action_tutorials_py
- actionlib_msgs
- ament_clang_format
- ament_clang_tidy
- ament_cmake
- ament_cmake_auto
- ament_cmake_clang_format
- ament_cmake_clang_tidy
- ament_cmake_copyright
- ament_cmake_core
- ament_cmake_cppcheck

\(^1\) The desktop variant is described in REP 2001 (desktop-variants).
• ament_cmake_cpplint
• ament_cmake_export_definitions
• ament_cmake_export_dependencies
• ament_cmake_export_include_directories
• ament_cmake_export_interfaces
• ament_cmake_export_libraries
• ament_cmake_export_link_flags
• ament_cmake_export_targets
• ament_cmake_flake8
• ament_cmake_gen_version_h
• ament_cmake_gmock
• ament_cmake_google_benchmark
• ament_cmake_gtest
• ament_cmake_include_directories
• ament_cmake_libraries
• ament_cmake_lint_cmake
• ament_cmake_mypy
• ament_cmake_nose
• ament_cmake_pclint
• ament_cmake_pep257
• ament_cmake_pycodestyle
• ament_cmake_pyflakes
• ament_cmake_pytest
• ament_cmake_python
• ament_cmake_ros
• ament_cmake_target_dependencies
• ament_cmake_test
• ament_cmake_uncrustify
• ament_cmake_version
• ament_cmake_xmllint
• ament(c)opyright
• ament_cppcheck
• ament_cpplint
• ament_flake8
• ament_index_cpp
<table>
<thead>
<tr>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>ament_index_python</td>
</tr>
<tr>
<td>ament_lint</td>
</tr>
<tr>
<td>ament_lint_auto</td>
</tr>
<tr>
<td>ament_lint_cmake</td>
</tr>
<tr>
<td>ament_lint_common</td>
</tr>
<tr>
<td>ament_mypy</td>
</tr>
<tr>
<td>ament_package</td>
</tr>
<tr>
<td>ament_pclint</td>
</tr>
<tr>
<td>ament_pep257</td>
</tr>
<tr>
<td>ament_pycodestyle</td>
</tr>
<tr>
<td>ament_pyflakes</td>
</tr>
<tr>
<td>ament_uncrustify</td>
</tr>
<tr>
<td>ament_xmllint</td>
</tr>
<tr>
<td>builtin_interfaces</td>
</tr>
<tr>
<td>camera_calibration_parsers</td>
</tr>
<tr>
<td>camera_info_manager</td>
</tr>
<tr>
<td>class_loader</td>
</tr>
<tr>
<td>common_interfaces</td>
</tr>
<tr>
<td>composition</td>
</tr>
<tr>
<td>composition_interfaces</td>
</tr>
<tr>
<td>demo_nodes_cpp</td>
</tr>
<tr>
<td>demo_nodes_cpp_native</td>
</tr>
<tr>
<td>demo_nodes_py</td>
</tr>
<tr>
<td>diagnostic_msgs</td>
</tr>
<tr>
<td>domain_coordinator</td>
</tr>
<tr>
<td>dummy_map_server</td>
</tr>
<tr>
<td>dummy_robot Bringup</td>
</tr>
<tr>
<td>dummy_sensors</td>
</tr>
<tr>
<td>example_interfaces</td>
</tr>
<tr>
<td>examples_rclcpp_async_client</td>
</tr>
<tr>
<td>examples_rclcpp_cbg_executor</td>
</tr>
<tr>
<td>examples_rclcpp_minimal_action_client</td>
</tr>
<tr>
<td>examples_rclcpp_minimal_action_server</td>
</tr>
<tr>
<td>examples_rclcpp_minimal_client</td>
</tr>
<tr>
<td>examples_rclcpp_minimal_composition</td>
</tr>
</tbody>
</table>
• examples_rclcpp_minimal_publisher
• examples_rclcpp_minimal_service
• examples_rclcpp_minimal_subscriber
• examples_rclcpp_minimal_timer
• examples_rclcpp_multithreaded_executor
• examples_rclcpp_wait_set
• examples_rclpy_executors
• examples_rclpy_guard_conditions
• examples_rclpy_minimal_action_client
• examples_rclpy_minimal_action_server
• examples_rclpy_minimal_client
• examples_rclpy_minimal_publisher
• examples_rclpy_minimal_service
• examples_rclpy_minimal_subscriber
• examples_rclpy_pointcloud_publisher
• examples_tf2_py
• fastrtps_cmake_module
• geometry_msgs
• google_benchmark_vendor
• image_tools
• image_transport
• interactive_markers
• intra_process_demo
• kdl_parser
• laser_geometry
• launch
• launch_pytest
• launch_ros
• launch_testing
• launch_testing_ament_cmake
• launch_testing_examples
• launch_testing_ros
• launch_xml
• launch_yaml
• libcurl_vendor
• libstatistics_collector
• libyaml_vendor
• lifecycle
• lifecycle_msgs
• lifecycle_py
• logging_demo
• message_filters
• mimick_vendor
• nav_msgs
• pendulum_control
• pendulum_msgs
• pluginlib
• pybind11_vendor
• python_cmake_module
• qt_gui_cpp
• quality_of_service_demo_cpp
• quality_of_service_demo_py
• rcl
• rcl_action
• rcl_interfaces
• rcl_lifecycle
• rcl_logging_interface
• rcl_logging_noop
• rcl_logging_spdlog
• rcl_yaml_param_parser
• rclepp
• rclepp_action
• rclepp_components
• rclepp_lifecycle
• rclpy
• rcpputils
• rcutils
• resource_retriever
• rmw
• rmw_connextdds
- `rmw_connextddds_common`
- `rmw_connextddsmicro`
- `rmw_cyclonedds_cpp`
- `rmw_dds_common`
- `rmw_fastrtps_cpp`
- `rmw_fastrtps_dynamic_cpp`
- `rmw_fastrtps_shared_cpp`
- `rmw_implementation`
- `rmw_implementation_cmake`
- `robot_state_publisher`
- `ros2action`
- `ros2bag`
- `ros2cli`
- `ros2cli_test_interfaces`
- `ros2component`
- `ros2doctor`
- `ros2interface`
- `ros2launch`
- `ros2lifecycle`
- `ros2lifecycle_test_fixtures`
- `ros2multicast`
- `ros2node`
- `ros2param`
- `ros2pkg`
- `ros2run`
- `ros2service`
- `ros2test`
- `ros2topic`
- `ros2trace`
- `rosbag2`
- `rosbag2_compression`
- `rosbag2_compression_zstd`
- `rosbag2_cpp`
- `rosbag2_interfaces`
- `rosbag2_performance_benchmarking`
- rosbag2_py
- rosbag2_storage
- rosbag2_storage_default_plugins
- rosbag2_test_common
- rosbag2_tests
- rosbag2_transport
- rosgraph_msgs
- rosidl_adapter
- rosidl_CLI
- rosidl_cmake
- rosidl_default_generators
- rosidl_default_runtime
- rosidl_generator_c
- rosidl_generator_cpp
- rosidl_generator_dds_idl
- rosidl_generator_py
- rosidl_parser
- rosidl_runtime_c
- rosidl_runtime_cpp
- rosidl_runtime_py
- rosidl_typesupport_c
- rosidl_typesupport_cpp
- rosidl_typesupport_fastrtps_c
- rosidl_typesupport_fastrtps_cpp
- rosidl_typesupport_interface
- rosidl_typesupport_introspection_c
- rosidl_typesupport_introspection_cpp
- rosidl_typesupport_introspection_tests
- rpyutils
- rqt_gui
- rqt_gui_cpp
- rqt_gui_py
- rqt_py_common
- rti_connext.dds_cmake_module
- rtttest
- rviz2
  - rviz_assimp_vendor
  - rviz_common
  - rviz_default_plugins
  - rviz_ogre_vendor
  - rviz_rendering
  - rviz_rendering_tests
  - rviz_visual_testing_framework
- sensor_msgs
- sensor_msgs_py
- shape_msgs
- shared_queues_vendor
- sqlite3_vendor
- sros2
- statistics_msgs
- std_msgs
- std_srvs
- stereo_msgs
- test_cli
- test_cli_remappping
- test_communication
- test_interface_files
- test_launch_ros
- test_launch_testing
- test_msgs
- test_quality_of_service
- test_rclcpp
- test_rmw_implementation
- test_security
- test_tf2
- test_tracetools
- test_tracetools_launch
- tf2
- tf2_bullet
- tf2_eigen
• tf2_eigen_kdl
• tf2_geometry_msgs
• tf2_kdl
• tf2_msgs
• tf2_py
• tf2_ros
• tf2_ros_py
• tf2_sensor_msgs
• tf2_tools
• tlsf
• tlsf_cpp
• topic_monitor
• topic_statistics_demo
• tracetools
• tracetools_launch
• tracetools_test
• tracetools_trace
• trajectory_msgs
• turtlesim
• urdf
• urdf_parser_plugin
• visualization_msgs
• yaml_cpp_vendor
• zstd_vendor

action_msgs

• Update maintainers to Chris Lalancette (#130)
• Contributors: Audrow Nash
action_tutorials_cpp

• Update maintainers to Audrow Nash and Michael Jeronimo (#543)
• Contributors: Audrow Nash

action_tutorials_interfaces

• Update maintainers to Audrow Nash and Michael Jeronimo (#543)
• Contributors: Audrow Nash

action_tutorials_py

• Update maintainers to Audrow Nash and Michael Jeronimo (#543)
• Contributors: Audrow Nash

actionlib_msgs

• Interface packages should fully <depend> on the interface packages that they depend on (#173)
• Update maintainers to Geoffrey Biggs and Tully Foote (#163)
• Contributors: Audrow Nash, Grey

ament_clang_format

• Update forthcoming version in changelogs
• Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
• Contributors: Audrow Nash

ament_clang_tidy

• Update forthcoming version in changelogs
• Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
• remove google style from clang-tidy default settings, removing need for default config file (#337)
• Improvements to ament_lint_clang_tidy. (#316)
• Contributors: Audrow Nash, Steven! Ragnarök, William Woodall
ament_cmake

- Update forthcoming version in changelog
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#362)
- Add ament_cmake_gen_version_h package (#198)
- Use FindPython3 instead of FindPythonInterp (#355)
- Update maintainers (#336)
- Contributors: Audrow Nash, Chris Lalancette, Shane Loretz, serge-nikulin

ament_cmake_auto

- Update forthcoming version in changelog
- Fix typo in ament_auto_find_test_dependencies (#363)
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#362)
- Add ament_auto_add_gtest (#344)
- Use FindPython3 instead of FindPythonInterp (#355)
- Update maintainers (#336)
- Contributors: Audrow Nash, Chris Lalancette, Daisuke Nishimatsu, Joshua Whitley, Shane Loretz

ament_cmake_clang_format

- Update forthcoming version in changelogs
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
- Contributors: Audrow Nash

ament_cmake_clang_tidy

- Update forthcoming version in changelogs
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
- Improvements to ament_lint_clang_tidy. (#316)
- Contributors: Audrow Nash, Steven! Ragnarök

ament_cmake_copyright

- Increase the ament_cmake_copyright default timeout. (#355)
- Update forthcoming version in changelogs
- [ament_cmake_copyright] Add file exclusion support (#328) *
- [ament_cmake_copyright] Add file exclusion support In the ament_copyright CMake function, the optional list argument EXCLUDE can now be used as an exclusion specifier. *
- [ament_cmake_copyright] Fix function header typo Remove reference to cppcheck in the EXCLUDE arg description.
• Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
• Contributors: Abrar Rahman Protyasha, Audrow Nash, Chris Lalancette

ament_cmake_core

• Update forthcoming version in changelog
• Resolve various ament_lint linter violations (#360) We can’t add ament_lint linters in ament_cmake in the traditional way without creating a circular dependency between the repositories. Even though we can’t automatically enforce linting, it’s still a good idea to try to keep conformance where possible.
• Update maintainers to Michael Jeronimo and Michel Hidalgo (#362)
• Use FindPython3 instead of FindPythonInterp (#355)
• Support commands with executable targets (#352)
• doc/resource_index: Indent list subitems correctly (#342)
• Update maintainers (#336)
• Contributors: Audrow Nash, Chris Lalancette, Michal Sojka, Scott K Logan, Shane Loretz

ament_cmake_cppcheck

• Update forthcoming version in changelogs
• [ament_cmake_cppcheck] Fix file exclusion behavior (#329) The EXCLUDE argument of the ament_cppcheck CMake function is a list, i.e. a multi-value keyword. As such, it needs to be placed out of the one-value keywords from the cmake_parse_arguments function call.
• Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
• Add cppcheck libraries option (#323) * adding ament_cppcheck libraries option * pass libraries option via CMake
  Co-authored-by: William Wedler <william.wedler@resquared.com>
• Contributors: Abrar Rahman Protyasha, Audrow Nash, Will

ament_cmake_cpplint

• Update forthcoming version in changelogs
• Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
• Contributors: Audrow Nash

ament_cmake_export_definitions

• Update forthcoming version in changelog
• Update maintainers to Michael Jeronimo and Michel Hidalgo (#362)
• Use FindPython3 instead of FindPythonInterp (#355)
• Update maintainers (#336)
• Contributors: Audrow Nash, Chris Lalancette, Shane Loretz
ament_cmake_export_dependencies

- Update forthcoming version in changelog
- Resolve various ament_lint linter violations (#360) We can’t add ament_lint linters in ament_cmake in the traditional way without creating a circular dependency between the repositories. Even though we can’t automatically enforce linting, it’s still a good idea to try to keep conformance where possible.
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#362)
- Use FindPython3 instead of FindPythonInterp (#355)
- Update maintainers (#336)
- Contributors: Audrow Nash, Chris Lalancette, Scott K Logan, Shane Loretz

ament_cmake_export_include_directories

- Update forthcoming version in changelog
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#362)
- Use FindPython3 instead of FindPythonInterp (#355)
- Update maintainers (#336)
- Contributors: Audrow Nash, Chris Lalancette, Shane Loretz

ament_cmake_export_interfaces

- Update forthcoming version in changelog
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#362)
- Use FindPython3 instead of FindPythonInterp (#355)
- Update maintainers (#336)
- Contributors: Audrow Nash, Chris Lalancette, Shane Loretz

ament_cmake_export_libraries

- Update forthcoming version in changelog
- Resolve various ament_lint linter violations (#360) We can’t add ament_lint linters in ament_cmake in the traditional way without creating a circular dependency between the repositories. Even though we can’t automatically enforce linting, it’s still a good idea to try to keep conformance where possible.
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#362)
- Use FindPython3 instead of FindPythonInterp (#355)
- Add note regarding interface libraries (#339)
- Update maintainers (#336)
- Contributors: Audrow Nash, Bjar Ne, Chris Lalancette, Scott K Logan, Shane Loretz
ament_cmake_export_link_flags

- Update forthcoming version in changelog
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#362)
- Use FindPython3 instead of FindPythonInterp (#355)
- Update maintainers (#336)
- Contributors: Audrow Nash, Chris Lalancette, Shane Loretz

ament_cmake_export_targets

- Update forthcoming version in changelog
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#362)
- Use FindPython3 instead of FindPythonInterp (#355)
- Update maintainers (#336)
- Contributors: Audrow Nash, Chris Lalancette, Shane Loretz

ament_cmake_flake8

- Update forthcoming version in changelogs
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
- Add custom config file support for flake8 (#331)
- Contributors: Audrow Nash, Kenji Miyake

ament_cmake_gen_version_h

- Add ament_generate_version_header and deprecate ament_cmake_gen_version_h (#377)
- Update forthcoming version in changelog
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#362)
- Add ament_cmake_gen_version_h package (#198)
- Contributors: Audrow Nash, Shane Loretz, serge-nikulin

ament_cmake_gmock

- Update forthcoming version in changelog
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#362)
- Use FindPython3 instead of FindPythonInterp (#355)
- Update maintainers (#336)
- Contributors: Audrow Nash, Chris Lalancette, Shane Loretz
ament_cmake_google_benchmark

- Update forthcoming version in changelog
- Resolve various ament_lint linter violations (#360) We can’t add ament_lint linters in ament_cmake in the traditional way without creating a circular dependency between the repositories. Even though we can’t automatically enforce linting, it’s still a good idea to try to keep conformance where possible.
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#362)
- Use FindPython3 instead of FindPythonInterp (#355)
- Update maintainers (#336)
- Contributors: Audrow Nash, Chris Lalancette, Scott K Logan, Shane Loretz

ament_cmake_gtest

- Update forthcoming version in changelog
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#362)
- Use FindPython3 instead of FindPythonInterp (#355)
- Update maintainers (#336)
- Contributors: Audrow Nash, Chris Lalancette, Shane Loretz

ament_cmake_include_directories

- Update forthcoming version in changelog
- Make ament_include_directories_order a function to allow paths with backslashes on windows. (#371) * Repalce backslashes with forward slashes on Windows * Typo * Replace slashes in ARGN * Don’t quote * Check ARGN has values before trying to string(REPLACE them) * Make ament_include_directories_order a function
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#362)
- Use FindPython3 instead of FindPythonInterp (#355)
- Update maintainers (#336)
- Contributors: Audrow Nash, Chris Lalancette, Shane Loretz

ament_cmake_libraries

- Update forthcoming version in changelog
- Resolve various ament_lint linter violations (#360) We can’t add ament_lint linters in ament_cmake in the traditional way without creating a circular dependency between the repositories. Even though we can’t automatically enforce linting, it’s still a good idea to try to keep conformance where possible.
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#362)
- Use FindPython3 instead of FindPythonInterp (#355)
- Update maintainers (#336)
- Contributors: Audrow Nash, Chris Lalancette, Scott K Logan, Shane Loretz
ament_cmake_lint_cmake

- Update forthcoming version in changelogs
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
- Contributors: Audrow Nash

ament_cmake_mypy

- Improve documentation by clarifying the purpose of different tools (#357)
- Update forthcoming version in changelogs
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
- Contributors: Audrow Nash, Bi0T1N

ament_cmake_nose

- Update forthcoming version in changelog
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#362)
- Use FindPython3 instead of FindPythonInterp (#355)
- Support commands with executable targets (#352)
- Update maintainers (#336)
- Contributors: Audrow Nash, Chris Lalancette, Shane Loretz

ament_cmake_pclint

- Improve documentation by clarifying the purpose of different tools (#357)
- Update forthcoming version in changelogs
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
- Contributors: Audrow Nash, Bi0T1N

ament_cmake_pep257

- Improve documentation by clarifying the purpose of different tools (#357)
- Update forthcoming version in changelogs
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
- Contributors: Audrow Nash, Bi0T1N
ament_cmake_pycodestyle

• Improve documentation by clarifying the purpose of different tools (#357)
• Update forthcoming version in changelogs
• Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
• Contributors: Audrow Nash, Bi0T1N

ament_cmake_pyflakes

• Update forthcoming version in changelogs
• Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
• Contributors: Audrow Nash

ament_cmake_pytest

• Update forthcoming version in changelog
• Update maintainers to Michael Jeronimo and Michel Hidalgo (#362)
• Fix misleading comment (#361)
• Use FindPython3 instead of FindPythonInterp (#355)
• Support commands with executable targets (#352)
• Mention other platforms in 'pytest/pytest-cov not found' warning (#337)
• Update maintainers (#336)
• Contributors: Audrow Nash, Chris Lalancette, Christophe Bedard, Shane Loretz

ament_cmake_python

• Use sysconfig directly to determine python lib dir (#378)
• Update forthcoming version in changelog
• Resolve various ament_lint linter violations (#360) We can’t add ament_lint linters in ament_cmake in the traditional way without creating a circular dependency between the repositories. Even though we can’t automatically enforce linting, it’s still a good idea to try to keep conformance where possible.
• Update maintainers to Michael Jeronimo and Michel Hidalgo (#362)
• Make ament_cmake_python symlink for symlink installs only (#357)
• Use FindPython3 instead of FindPythonInterp (#355)
• Make ament_python_install_package() match setuptools’ egg names. (#338)
• Drop ament_cmake_python outdated tests. (#340)
• Update maintainers (#336)
• Make ament_python_install_package() install console_scripts (#328)
• Contributors: Audrow Nash, Chris Lalancette, Michel Hidalgo, Scott K Logan, Shane Loretz
ament_cmake_ros

- Refactor domain_coordinator API to use a context manager (#12)
- Contributors: Timo Röhling

ament_cmake_target_dependencies

- Update forthcoming version in changelog
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#362)
- Use FindPython3 instead of FindPythonInterp (#355)
- Fix bug packages with multiple configurations (#318)
- Update maintainers (#336)
- Contributors: Audrow Nash, Chris Lalancette, Shane Loretz

ament_cmake_test

- Update forthcoming version in changelog
- Resolve various ament_lint linter violations (#360) We can’t add ament_lint linters in ament_cmake in the traditional way without creating a circular dependency between the repositories. Even though we can’t automatically enforce linting, it’s still a good idea to try to keep conformance where possible.
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#362)
- Use FindPython3 instead of FindPythonInterp (#355)
- Update maintainers (#336)
- Contributors: Audrow Nash, Chris Lalancette, Scott K Logan, Shane Loretz

ament_cmake_uncrustify

- Update forthcoming version in changelogs
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
- [ament_cmake_uncrustify] Add file exclude support (#330) In the ament_uncrustify CMake function, the optional list argument EXCLUDE can now be used as an exclusion specifier.
- Contributors: Abrar Rahman Protyasha, Audrow Nash

ament_cmake_version

- Update forthcoming version in changelog
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#362)
- Use FindPython3 instead of FindPythonInterp (#355)
- Update maintainers (#336)
- Contributors: Audrow Nash, Chris Lalancette, Shane Loretz
ament_cmake_xmllint

- Update forthcoming version in changelogs
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
- Contributors: Audrow Nash

ament_copyright

- Fix importlib_metadata warning on Python 3.10. (#365)
- Update forthcoming version in changelogs
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
- [ament_copyright] Fix file exclusion behavior (#327) * [ament_copyright] Fix file exclusion behavior This commit fixes the faulty file exclusion behavior reported in https://github.com/ament/ament_lint/issues/326. Specifically, the exclusion list is matched against traversed files in the crawler module. Changes inspired by https://github.com/ament/ament_lint/pull/299/. * Update excluded file path in copyright tests Since file names are not indiscriminately matched throughout the search tree anymore, the excluded files listed in the copyright tests need to be updated relative to the root of the package. * Add test cases to check exclusion behavior Specifically, these tests check for: - Incorrect exclusion of single filenames. - Correct exclusion of relatively/absolutely addressed filenames. - Correct exclusion of wildcarded paths. * Add unit tests for crawler module These unit tests must both search and exclusion behaviors are correctly demonstrated by the ament_copyright.crawler module.
- Add SPDX identifiers to the licenses. (#315)
- Contributors: Abrar Rahman Protyasha, Audrow Nash, Chris Lalancette

ament_cppcheck

- Disable cppcheck 2.x. (#345)
- Update forthcoming version in changelogs
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
- Add cppcheck libraries option (#323) * adding ament_cppcheck libraries option * pass libraries option via CMake Co-authored-by: William Wedler <william.wedler@resquared.com>
- Contributors: Audrow Nash, Chris Lalancette, Will

ament_cpplint

- ignore NOLINT comments with categories that come from clang-tidy (#339)
- Update forthcoming version in changelogs
- Reapply patches Reapply parts of 232428752251de61e84ef013bcd643e35eb9038d that are still relevant.
- Update cpplint version Point to the fork https://github.com/cpplint/cpplint Contains updates for modern C++ standards (e.g. C++14 and C++17).
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
• [ament_copyright] Fix file exclusion behavior (#327) * [ament_copyright] Fix file exclusion behavior This commit fixes the faulty file exclusion behavior reported in https://github.com/ament/ament_lint/issues/326. Specifically, the exclusion list is matched against traversed files in the crawler module. Changes inspired by https://github.com/ament/ament_lint/pull/299/. * Update excluded file path in copyright tests Since file names are not indiscriminately matched throughout the search tree anymore, the excluded files listed in the copyright tests need to be updated relative to the root of the package. * Add test cases to check exclusion behavior Specifically, these tests check for: - Incorrect exclusion of single filenames. - Correct exclusion of relatively/absolutely addressed filenames. - Correct exclusion of wildcarded paths. * Add unit tests for crawler module These unit tests make sure both search and exclusion behaviors are correctly demonstrated by the ament_copyright.crawler module.

• Contributors: Abrar Rahman Protyasha, Audrow Nash, Dirk Thomas, Jacob Perron, William Woodall

ament_flake8

• Remove use of distutils.version.LooseVersion. (#346)
• Update forthcoming version in changelogs
• Ignore */_* dirs in ament_flake8 (#335) Other ament_* linters specifically ignore directories starting with a dot or underscore when crawling for files to lint. They also do so implicitly, so this change mimics that same pattern so that the behavior is consistent.
• Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
• Ignore flake8-blind-except B902 (#292)
• Contributors: Audrow Nash, Chris Lalancette, Scott K Logan

ament_index_cpp

• Install includes to include/ (#83)
• Remove ament_export_include_directories and ament_export_libraries (#81)
• Contributors: Shane Loretz

ament_index_python

• Print warning when get_package_share_directory() does not exist (Fix #74) (#77)
• Fail lookups on invalid resource names (#69)
• Add get_package_share_path method (#73)
• Contributors: David V. Lu, rob-clarke
ament_lint

- Update forthcoming version in changelogs
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
- Contributors: Audrow Nash

ament_lint_auto

- Update forthcoming version in changelogs
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
- Contributors: Audrow Nash

ament_lint_cmake

- Update forthcoming version in changelogs
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
- [ament_copyright] Fix file exclusion behavior (#327) * [ament_copyright] Fix file exclusion behavior This commit fixes the faulty file exclusion behavior reported in https://github.com/ament/ament_lint/issues/326. Specifically, the exclusion list is matched against traversed files in the crawler module. Changes inspired by https://github.com/ament/ament_lint/pull/299/. * Update excluded file path in copyright tests Since file names are not indiscriminately matched throughout the search tree anymore, the excluded files listed in the copyright tests need to be updated relative to the root of the package. * Add test cases to check exclusion behavior Specifically, these tests check for: - Incorrect exclusion of single filenames. - Correct exclusion of relatively/absolutely addressed filenames. - Correct exclusion of wildcarded paths. * Add unit tests for crawler module These unit tests make sure both search and exclusion behaviors are correctly demonstrated by the ament_copyright.crawler module.
- Contributors: Abrar Rahman Protyasha, Audrow Nash

ament_lint_common

- Update forthcoming version in changelogs
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
- Fix typo in ament_lint_common/package.xml (#336)
- Contributors: Audrow Nash, Kenji Miyake

ament_mypy

- Improve documentation by clarifying the purpose of different tools (#357)
- Update forthcoming version in changelogs
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
- Contributors: Audrow Nash, Bi0T1N
Vulcanexus Documentation, Release 1.0.0

ament_package

- Set forthcoming for previous version
- Add support for appending to environment variables (#130) This works largely the same as ‘prepend-non-duplicate’, but instead puts the candidate value at the end of the target variable.
- Update maintainers to Audrow Nash (#135)
- Make python executable variable ament_package specific (#134)
- Contributors: Audrow Nash, Scott K Logan, Shane Loretz

ament_pclint

- Improve documentation by clarifying the purpose of different tools (#357)
- Update forthcoming version in changelogs
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
- [ament_copyright] Fix file exclusion behavior (#327) This commit fixes the faulty file exclusion behavior reported in https://github.com/ament/ament_lint/issues/326. Specifically, the exclusion list is matched against traversed files in the crawler module. Changes inspired by https://github.com/ament/ament_lint/pull/299/. * Update excluded file path in copyright tests Since file names are not indiscriminately matched throughout the search tree anymore, the excluded files listed in the copyright tests need to be updated relative to the root of the package. * Add test cases to check exclusion behavior Specifically, these tests check for: - Incorrect exclusion of single filenames. - Correct exclusion of relatively/absolutely addressed filenames. - Correct exclusion of wildcarded paths. * Add unit tests for crawler module These unit tests make sure both search and exclusion behaviors are correctly demonstrated by the ament_copyright.crawler module.
- Contributors: Abrar Rahman Protyasha, Audrow Nash, Bi0T1N

ament_pep257

- Improve documentation by clarifying the purpose of different tools (#357)
- Remove use of distutils.version.LooseVersion. (#346)
- Update forthcoming version in changelogs
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
- Contributors: Audrow Nash, Bi0T1N, Chris Lalancette

ament_pycodestyle

- Improve documentation by clarifying the purpose of different tools (#357)
- Update forthcoming version in changelogs
- Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
- Contributors: Audrow Nash, Bi0T1N

Chapter 4. Contributing to the documentation
ament_pyflakes

• Update forthcoming version in changelogs
• Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
• Contributors: Audrow Nash

ament_uncrustify

• Update forthcoming version in changelogs
• [ament_uncrustify] Fix file exclusion behavior (#334) * [ament_uncrustify] Fix file exclusion behavior This PR fixes the file exclusion behavior reported in #326. Specifically, the exclusion list is matched against files/directories as the search path is traversed. Tries to maintain consistency with #327. * [ament_uncrustify] Add file exclusion tests * [ament_uncrustify] Remove erroneous pytest marker
• Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
• [ament_uncrustify] Add ament_lint tests (#338) * Add ament_lint tests on ament_uncrustify * Address linter warnings in ament_uncrustify
• Contributors: Abrar Rahman Protyasha, Audrow Nash

ament_xmllint

• Update forthcoming version in changelogs
• Update maintainers to Michael Jeronimo and Michel Hidalgo (#340)
• Contributors: Audrow Nash

builtin_interfaces

• Update maintainers to Chris Lalancette (#130)
• Contributors: Audrow Nash

camera_calibration_parsers

• Tests depend on rcpputils (#236)
• Remove YAML_CPP_DLL define (#231)
• Export a modern CMake target instead of variables and install includes to include/${PROJECT_NAME} (#218)
• Update maintainers (#173)
• Contributors: Akash, Alejandro Hernández Cordero, Shane Loretz
camera_info_manager

- Export a modern CMake target instead of variables and install includes to include/${PROJECT_NAME} (#218)
- Update maintainers (#173)
- Contributors: Alejandro Hernández Cordero, Shane Loretz

class_loader

- Install includes to include/ (#191)
- Fix include order for cpplint (#192)
- Update maintainers to Geoffrey Biggs and Michael Carroll (#190)
- Fix spelling mistake (#184)
- Contributors: Audrow Nash, David V. Lu!!, Jacob Perron, Shane Loretz

common_interfaces

- Update maintainers to Geoffrey Biggs and Tully Foote (#163)
- Contributors: Audrow Nash

composition

- Update maintainers to Audrow Nash and Michael Jeronimo (#543)
- Additional fixes for documentation in demos. (#538)
- Fixing deprecated subscriber callback warnings (#532)
- Contributors: Abrar Rahman Protyasha, Audrow Nash, Chris Lalancette

composition_interfaces

- Update maintainers to Chris Lalancette (#130)
- Contributors: Audrow Nash

demo_nodes_cpp

- Update maintainers to Audrow Nash and Michael Jeronimo (#543)
- Add how to fix the most vexing parse problem (#541) * use uniform initialization
- Fixing deprecated subscriber callback warnings (#532)
- Update talker_loaned_message.cpp (#518)
- Revert “Use sizeof(char) in place for sizeof(void) (#515)” (#516)
- change how serialized message works with subscription (#497)
- Use sizeof(char) in place for sizeof(void) (#515)
• Fix small print issue in allocator tutorial. (#509)
• Contributors: Abrar Rahman Protyasha, Audrow Nash, Chris Lalancette, Michel Hidalgo, Tomoya Fujita, William Woodall, Zongbao Feng

demo_nodes_cpp_native

• Update maintainers to Audrow Nash and Michael Jeronimo (#543)
• Fix typo in demo_nodes_cpp_native package description (#536)
• Contributors: Audrow Nash, Víctor Mayoral Vilches

demo_nodes_py

• Cleanups in demo_nodes_py. (#555)
• Update maintainers to Audrow Nash and Michael Jeronimo (#543)
• Fixed typo executor -> executors (#542)
• Update python nodes SIGINT handling (#539)
• Contributors: Audrow Nash, Chris Lalancette, Ivan Santiago Paunovic, ori155

diagnostic_msgs

• Interface packages should fully <depend> on the interface packages that they depend on (#173)
• Update maintainers to Geoffrey Biggs and Tully Foote (#163)
• Contributors: Audrow Nash, Grey

domain_coordinator

• Update maintainers to Michel Hidalgo (#13)
• Refactor domain_coordinator API to use a context manager (#12)
• Contributors: Audrow Nash, Timo Röhling

dummy_map_server

• Update maintainers to Audrow Nash and Michael Jeronimo (#543)
• Contributors: Audrow Nash
dummy_robotbringup

- Update maintainers to Audrow Nash and Michael Jeronimo (#543)
- Contributors: Audrow Nash

dummy_sensors

- Update maintainers to Audrow Nash and Michael Jeronimo (#543)
- Contributors: Audrow Nash

example_interfaces

- Update maintainers to Mabel Zhang (#15)
- Add changelog (#14)
- Contributors: Audrow Nash, Ivan Santiago Paunovic

examples_rclcpp_async_client

- Updated maintainers (#329)
- Add example of how to prune old requests in client API (#322)
- Contributors: Aditya Pande, Ivan Santiago Paunovic

examples_rclcpp_cbg_executor

- Improve scheduling configuration of examples_rclcpp_cbg_executor package (#331)
- Added jitter measurement to examples_rclcpp_cbg_executor. (#328)
- Fix deprecated subscriber callbacks (#323)
- Remove use of get_callback_groups(). (#320)
- Contributors: Abrar Rahman Protyasha, Chris Lalancette, Ralph Lange

examples_rclcpp_minimal_action_client

- Updated maintainers (#329)
- Contributors: Aditya Pande
examples_rclcpp_minimal_action_server

- Updated maintainers (#329)
- Contributors: Aditya Pande

examples_rclcpp_minimal_client

- Updated maintainers (#329)
  - Add example of how to prune old requests in client API (#322)
  - Contributors: Aditya Pande, Ivan Santiago Paunovic

examples_rclcpp_minimal_composition

- Updated maintainers (#329)
  - Contributors: Aditya Pande

examples_rclcpp_minimal_publisher

- Add an example about how to use wait_for_all_acked (#316)
- Updated maintainers (#329)
- Add try&catch statement to unique network flow publisher example (#313)
- Add type adaption example (#300)
- Contributors: Aditya Pande, Audrow Nash, Barry Xu, Tomoya Fujita

examples_rclcpp_minimal_service

- Updated maintainers (#329)
  - Contributors: Aditya Pande

examples_rclcpp_minimal_subscriber

- Use const& signature for read-only sub callbacks (#337)
- Updated maintainers (#329)
- Fix deprecated subscriber callbacks (#323)
- Add wait set examples (#315)
- Add type adaption example (#300)
- Contributors: Abrar Rahman Protyasha, Aditya Pande, Audrow Nash, carlossvg
examples_rclcpp_minimal_timer

- Updated maintainers (#329)
- Contributors: Aditya Pande

examples_rclcpp_multithreaded_executor

- Updated maintainers (#329)
- Fix deprecated subscriber callbacks (#323)
- Contributors: Abrar Rahman Protyasha, Aditya Pande

examples_rclcpp_wait_set

- Add wait set examples (#315)
- Contributors: carlossvg

examples_rclpy_executors

- Update maintainers to Aditya Pande and Shane Loretz (#332)
- Updated maintainers (#329)
- Update python nodes sigint/sigterm handling (#330)
- Contributors: Aditya Pande, Audrow Nash, Ivan Santiago Paunovic

examples_rclpy_guard_conditions

- Update maintainers to Aditya Pande and Shane Loretz (#332)
- Updated maintainers (#329)
- Contributors: Aditya Pande, Audrow Nash

examples_rclpy_minimal_action_client

- Update maintainers to Aditya Pande and Shane Loretz (#332)
- Updated maintainers (#329)
- Contributors: Aditya Pande, Audrow Nash
examples_rclpy_minimal_action_server

- Update maintainers to Aditya Pande and Shane Loretz (#332)
- Updated maintainers (#329)
- Contributors: Aditya Pande, Audrow Nash

examples_rclpy_minimal_client

- Update maintainers to Aditya Pande and Shane Loretz (#332)
- Updated maintainers (#329) * Updated maintainers * Removed author
- Contributors: Aditya Pande, Audrow Nash

examples_rclpy_minimal_publisher

- Update maintainers to Aditya Pande and Shane Loretz (#332)
- Updated maintainers (#329)
- Contributors: Aditya Pande, Audrow Nash

examples_rclpy_minimal_service

- Update maintainers to Aditya Pande and Shane Loretz (#332)
- Updated maintainers (#329)
- Contributors: Aditya Pande, Audrow Nash

examples_rclpy_minimal_subscriber

- Update maintainers to Aditya Pande and Shane Loretz (#332)
- Updated maintainers (#329)
- Contributors: Aditya Pande, Audrow Nash

examples_rclpy_pointcloud_publisher

- Update maintainers to Aditya Pande and Shane Loretz (#332)
- Contributors: Audrow Nash
examples_tf2_py

- Update maintainers to Alejandro Hernandez Cordero and Chris Lalancette (#481)
- Use underscores instead of dashes in setup.cfg. (#403)
- Contributors: Audrow Nash

fastrtps_cmake_module

- Update maintainers to Shane Loretz (#83)
- Contributors: Audrow Nash

geometry_msgs

- Interface packages should fully $\langle$depend$\rangle$ on the interface packages that they depend on (#173)
- Update maintainers to Geoffrey Biggs and Tully Foote (#163)
- Contributors: Audrow Nash, Grey

google_benchmark_vendor

- Add git buildtool dependency.
- Use git hash for google_benchmark_vendor (#20)
- Update to google benchmark version 1.6.1 (#19)
- Update maintainers to Audrow Nash (#18)
- Update google_benchmark to v1.5.3 (#16) 1. Change google_benchmark version from v1.5.2 to v1.5.3. Because v1.5.2 can not build with GCC 11 2. Removed shrink-tz-offset-size.patch because of this patch was merged in google-benchmark repo.
- Add changelog (#15)
- Shrink the size of the tz_offset variable. (#13)
- Update the patching to work on Windows without admin. (#11)
- Always preserve source permissions in vendor packages. (#12)
- Update package maintainers. (#10)
- Upgrade google benchmark from v1.5.1 to v1.5.2 to include QNX patch. (#9)
- Set the SOVERSION on benchmark libraries. (#8)
- Set minimum criteria for system package. (#3)
- Work around warnings building Google Benchmark w/Clang. (#2)
- Initial google_benchmark_vendor package. (#1)
- Initial commit.
image_tools

- Install includes to include/${PROJECT_NAME} (#548)
- Fix include order and relative paths for cpplint (#551)
- Reduce the number of OpenCV libraries image_tools links against. (#549)
- Adds copy constructor and assignment operator to ROSCvMatContainer (#546)
- Fixes for uncrustify 0.72 (#545)
- Update maintainers to Audrow Nash and Michael Jeronimo (#543)
- Additional fixes for documentation in demos. (#538)
- Fixing deprecated subscriber callback warnings (#532)
- ambiguity: unknown type name 'nullptr_t' (#528)
- Add type masquerading demos (#482)
- Add support for visualizing yuv422 (#499)
- Contributors: Abrar Rahman Protyasha, Audrow Nash, Chris Lalancette, Gonzo, Jacob Perron, Shane Loretz, William Woodall, joshua-qnx, xwnb

image_transport

- Image transport publisher crash fixes (#235)
- Simple IT plugins shutdown (#225)
- Remove PLUGINLIB_DISABLE_BOOST_FUNCTIONS definition. (#226)
- Fix include order for cpplint (#221) Relates to https://github.com/ament/ament_lint/pull/324
- Export a modern CMake target instead of variables and install includes to include/${PROJECT_NAME} (#218)
- Fix SimpleSubscriberPlugin (#195)
- Make sure to mark overridden methods as 'override'. (#192)
- Expose subscription options (#186)
- fix mistyping 'cammera_publisher.hpp -> camera_publisher.hpp' (#177)
- Update maintainers (#173)
- make CameraPublisher::getNumSubscribers() work (#163)
**interactive_markers**

- Do not publish if context is invalid during shutdown (#89)
- Install includes to include/ and misc CMake fixes (#85)
- Fix deprecation warning introduced after client API update (#83)
- Fix deprecated sub callback warnings (#84)
- Include tf2_geometry_msgs.hpp instead of the h file. (#82)
- Contributors: Abrar Rahman Protyasha, Chris Lalancette, Ivan Santiago Paunovic, Jacob Perron, Shane Loretz

**intra_process_demo**

- Add opencv_imgproc dependency for cv::putText (#554)
- Install includes to include/${PROJECT_NAME} (#548)
- Fix include order and relative paths for cpplint (#551)
- Update maintainers to Audrow Nash and Michael Jeronimo (#543)
- Additional fixes for documentation in demos. (#538)
- Fixing deprecated subscriber callback warnings (#532)
- Revert “Add type masquerading demos (#482)” (#520)
- Add type masquerading demos (#482)
- Contributors: Abrar Rahman Protyasha, Audrow Nash, Chris Lalancette, Jacob Perron, Shane Loretz, William Woodall

**kdl_parser**

- Depend on orocos-kdl vendor packages (#58)
- Install includes to include/ and misc CMake fixes (#61)
- Update to uncrustify 0.72 (#60)
- Contributors: Chris Lalancette, Jacob Perron, Shane Loretz

**laser_geometry**

- Install headers to include/${PROJECT_NAME} (#86)
- Explicit cast to double to prevent loss of precision
- Fix Duration casting issue leading to no undistortion
- Fix building on running on Windows Debug (#82)
- Update python code and tests for ros2 (#80)
- Contributors: Chris Lalancette, Jonathan Binney, Marco Lampacrescia, Shane Loretz
launch

- Sandbox environment in tests to fix repeated job failures (#609)
- Start Python faster in test_execute_processss_shutdown to avoid flakey failures (#608)
- Fix warnings from importlib_metdata on Python 3.10. (#606)
- Add boolean substitutions (#598)
- Support scoping environment variables (#601)
- Fix awaiting shutdown in launch context (#603)
- Fix parse respawn var (#569)
- Make the logged command pretty in ExecuteLocal (#594)
- 'output' is expanded as a substitution in XML/YAML files (#577)
- Skip warning test if warning already happened (#585)
- Use asyncio.wait with timeout rather than sleep (#576)
- Make test_parser compatible with Python older than 3.8 (#575)
- Propagate exceptions of completed actions to launch service main loop (#566)
- Warn when loading launch extensions fails (#572)
- Add in two fixes for Jammy (#571)
- Evaluate math symbols and functions in python expression (#557)
- Document TimerAction params (#558)
- Improve launch arguments introspection (#556)
- Update maintainers to Aditya Pande and Michel Hidalgo (#559)
- Updated maintainers (#555)
- First prototype of native pytest plugin for launch based tests (#528)
- Allow for raw path specification in IncludeLaunchDescription (#544)
- Adding Executable description class (#454)
- event handlers: Allow to match the target action with a callable and not only with an object instance (#540)
- Add AppendEnvironmentVariable action (#543)
- Document EnvironmentVariable substitution resolution context caveat (#541)
- Feature clear launch configs (#515)
- Add examples to ExecuteProcess docs (#525)
- Fix DeclareLaunchArgument xml parsing and constructor (#529)
- Fix pytest run on Windows (#526)
- Improving docs (#523)
- Add filtering mechanism for executable prefix application (#522)
- Make each parser extension provide a set of file extensions (#516)
- Add missing exec dependency on PyYAML (#493)
- Refactor TimerAction to allow RosTimer to extend (#512)
• Improve (Not)Equals condition type hinting (#510)
• Contributors: Aditya Pande, Audrow Nash, Cameron Miller, Chris Lalancette, Christophe Bedard, David V. Lu!, Derek Chopp, HMellor, Immanuel Martini, Ivan Santiago Paunovic, Jacob Perron, Kenji Miyake, Khush Jain, Kosuke Takeuchi, Rebecca Butler, Scott K Logan, Shane Loretz, roger-strain, tumtom

launch_pytest

• Update maintainers to Aditya Pande and Michel Hidalgo (#559)
• [launch_pytest] Modify how wait_for_output()/wait_for_stderr() work, add assert_*() alternatives (#553)
• Updated maintainers (#555)
  • [launch_pytest] Fix issue when colcon –retest-until-fail flag is used (#552)
• First prototype of native pytest plugin for launch based tests (#528)
• Contributors: Aditya Pande, Audrow Nash, Ivan Santiago Paunovic

launch_ros

• Fix importlib_metadata warning on Python 3.10. (#307)
• Use correct namespace when evaluating parameter files for composable nodes (#303)
• Handle empty strings when evaluating parameters (#300)
• Add parameter substitution (#297)
• fix bug in warning when an entry point fails to load (#243)
• More Helpful Error Messages (#275)
• Update maintainers in setup.py (#287)
• Set parameters from file for composable nodes (#281)
• Update package maintainers (#284)
• Update node name matcher (#282)
• Support both parameter file configurations for composable nodes (#259)
• Handle substitutions in RosTimer (#264)
• Add SetParametersFromFile action (#260)
• Properly support ros_args attribute through launch frontends (#253)
• Add ‘push_ros_namespace’ alias to ‘push-ros-namespace’ (#250)
• Add ros_arguments option to Node action (#249)
• Refactor RosTimer to extend TimerAction (#248)
• ROS Timer Action (#244)
• Support container in frontend (#235)
• Fix a small typo in a comment (#237)
• Better document parameter handling in Node (#234)
• Make ‘ros2 launch’ work again. (launch #201)
• Added LaunchLogger class (launch #145)
• Changed logger.warn (deprecated) to logger.warning. (launch #199)
• Added Plumb rclpy.init context to get_default_launch_description. (launch #193)
• Added normalize_parameters and evaluate_parameters. (launch #192)
• Added normalize_remap_rule and types. (launch #173)
• Renamed transitions to match changes in lifecycle_msgs (launch #153)
• Added support for passing parameters as a dictionary to a Node (launch #138)
• Made various fixes and added tests for remappings passed to Node actions (launch #137)
• Contributors: Aditya Pande, Audrow Nash, Chris Lalancette, Christophe Bedard, David V. Lu!!, Felix Divo, Jacob Perron, Kenji Miyake, Michel Hidalgo, Rebecca Butler, William Woodall

launch_testing

• Removed the deprecated ready_fn feature (#589)
• Added case for instances of ExecuteLocal in resolveProcess function (#587)
• Add compatibility with pytest 7 (#592)
• Renamed three files from example_processes (#573)
• Fix launch_testing README.md proc keyword to process. (#554) (#560)
• Declare frontend group dependency & use explicit dependencies in launch_testing (#520)
• Update maintainers to Aditya Pande and Michel Hidalgo (#559)
• Updated maintainers (#555)
• First prototype of native pytest plugin for launch based tests (#528)
• Adding Executable description class (#454)
• Add a “hello world” style example (#532)
• Contributors: Aditya Pande, Audrow Nash, Christophe Bedard, Ivan Santiago Paunovic, Jacob Perron, Khush Jain, Matt Lanting, Shane Loretz, William Woodall, roger-strain

launch_testing_ament_cmake

• [launch_testing_ament_cmake] Add test label (#584)
• Update maintainers to Aditya Pande and Michel Hidalgo (#559)
• Updated maintainers (#555)
• Contributors: Aditya Pande, Audrow Nash, Keisuke Shima
Vulcanexus Documentation, Release 1.0.0

launch_testing_examples

- Readded WaitForTopics utility (#333)
- Final batch of examples (#327)
- Update maintainers to Aditya Pande and Shane Loretz (#332)
- Updated maintainers (#329)
- Reverted WaitForTopics utility usage (#326)
- Moved examples (#324)
- Contributors: Aditya Pande, Audrow Nash

launch_testing_ros

- Add hz param to talker.py to fix wait_for_topic_launch_test (#309)
- Revert WaitForTopics (#288)
- Update maintainers in setup.py (#287)
- Move pytest entrypoints to own module (#278)
- Update package maintainers (#284)
- Check that future is done, and always call rclpy.shutdown (#273)
- Revert “launch testing : Wait for topics to publish (#274)” (#276)
- Add WaitForTopics utility for waiting on publishers (#274)
- Remove unused code, Future.result() already raises (#270)
- Add timeout to wait for service response in example (#271)
- Add examples (#263)
- Contributors: Aditya Pande, Audrow Nash, Jacob Perron, Jorge Perez, Michel Hidalgo, Shane Loretz

launch_xml

- Fix sphinx directive to cross-ref Launch method (#605)
- Add boolean substitutions (#598)
- Support scoping environment variables (#601)
- ‘output’ is expanded as a substitution in XML/YAML files (#577)
- Declare frontend group dependency & use explicit dependencies in launch_testing (#520)
- Update maintainers to Aditya Pande and Michel Hidalgo (#559)
- Updated maintainers (#555)
- Add AppendEnvironmentVariable action (#543)
- Feature clear launch configs (#515)
- Fix DeclareLaunchArgument xml parsing and constructor (#529)
- Add ‘launch’ to sets of launch file extensions (#518)
• Make each parser extension provide a set of file extensions (#516)
• Contributors: Abrar Rahman Protyasha, Aditya Pande, Audrow Nash, Christophe Bedard, Derek Chopp, Ivan Santiago Paunovic, Jacob Perron, Kenji Miyake, Khush Jain

**launch_yaml**

• Fix sphinx directive to cross-ref Launch method (#605)
• Add boolean substitutions (#598)
• Support scoping environment variables (#601)
• ‘output’ is expanded as a substitution in XML/YAML files (#577)
• Declare frontend group dependency & use explicit dependencies in launch_testing (#520)
• Update maintainers to Aditya Pande and Michel Hidalgo (#559)
• Updated maintainers (#555)
• Add AppendEnvironmentVariable action (#543)
• Feature clear launch configs (#515)
• Add ‘launch’ to sets of launch file extensions (#518)
• Make each parser extension provide a set of file extensions (#516)
• Contributors: Abrar Rahman Protyasha, Aditya Pande, Audrow Nash, Christophe Bedard, Derek Chopp, Jacob Perron, Kenji Miyake, Khush Jain

**libcurl_vendor**

• Update to curl 7.81. (#74)
• Update maintainers (#66)
• Contributors: Audrow Nash, Chris Lalancette

**libstatistics_collector**

• Bump pascalgn/automerge-action from 0.14.3 to 0.15.2
• Bump ros-tooling/setup-ros from 0.2.2 to 0.3.0
• Bump actions/upload-artifact from 2.3.1 to 3
• Bump actions/upload-artifact from 2.2.4 to 2.3.1
• Bump actions/checkout from 2 to 3
• Bump ros-tooling/setup-ros from 0.2.1 to 0.2.2 (#123)
• Install includes to include/${PROJECT_NAME} (#122)
• Bump codecov/codecov-action from 2.0.3 to 2.1.0
• Bump pascalgn/automerge-action from 0.14.2 to 0.14.3
• Bump codecov/codecov-action from 2.0.2 to 2.0.3
• Use rosidl_get_typesupport_target() (#116)
• Bump codecov/codecov-action from 2.0.1 to 2.0.2
• Bump codecov/codecov-action from 1.5.2 to 2.0.1
• Bump actions/upload-artifact from 1 to 2.2.4
• Bump codecov/codecov-action from 1.5.1 to 1.5.2
• Bump codecov/codecov-action from 1.3.1 to 1.5.1
• Bump ros-tooling/setup-ros from 0.2.0 to 0.2.1
• Bump pascalgn/automerge-action from 0.14.1 to 0.14.2
• Bump ros-tooling/setup-ros from 0.1 to 0.2.0
• Bump pascalgn/automerge-action from 0.13.1 to 0.14.1
• Fix autoapprove
• Package.json explicitly owned by emerson to minimize notifications
• Bump hmarr/auto-approve-action from v2.0.0 to v2.1.0
• Bump codecov/codecov-action from v1.2.1 to v1.3.1
• Contributors: Chris Lalancette, Emerson Knapp, Shane Loretz, dependabot[bot]

**libyaml_vendor**

• Add a buildtool dependency on git. (#48)
• Install headers to include$/PROJECT_NAME/ (#46)
• Merge pull request #43 from ros2/update-maintainers
• Update maintainers to Audrow Nash
• Contributors: Audrow Nash, Shane Loretz, Steven! Ragnarök

**lifecycle**

• Make lifecycle demo automatically exit when done (#558)
• Use default on_activate()/on_deactivate() implementenation of Node (#552)
• Update maintainers to Audrow Nash and Michael Jeronimo (#543)
• Fix use of future in lifecycle demo (#534)
• Fixing deprecated subscriber callback warnings (#532)
• Contributors: Abrar Rahman Protyasha, Audrow Nash, Christophe Bedard, Ivan Santiago Paunovic, Shane Loretz
lifecycle_msgs

• Update maintainers to Chris Lalancette (#130)
• Contributors: Audrow Nash

lifecycle_py

• Create changelog for lifecycle_py
• Add rclpy lifecycle demo (#547)
• Contributors: Audrow Nash, Ivan Santiago Paunovic

logging_demo

• Update maintainers to Audrow Nash and Michael Jeronimo (#543)
• Additional fixes for documentation in demos. (#538)
• Use rosidl_get_typesupport_target() (#529)
• Contributors: Audrow Nash, Chris Lalancette, Shane Loretz

message_filters

• Use RCL_ROS_TIME for message_traits::TimeStamp (#72)
• Install includes to include/${PROJECT_NAME} (#71)
• Update maintainers (#67)
• Suppress rclcpp deprecation warnings in unit tests (#62)
• Add missing overrides to subscriber.h (#60)
• Add lifecycle node support (#59)
• Correct package.xml and CMakeLists.txt (#58)
• Expose Subscription Options - V2 (#56)
• Contributors: Abrar Rahman Protyasha, Audrow Nash, Hunter L. Allen, Kenji Brameld, Michel Hidalgo, Rebecca Butler, Shane Loretz

mimick_vendor

• support pi zero (#24)
• Update maintainers to Geoffrey Biggs (#23)
• Update to latest commit for Apple M1 support (#20)
• Contributors: Audrow Nash, Brett Downing, Christophe Bedard
nav_msgs

- Interface packages should fully <depend> on the interface packages that they depend on (#173)
- Update maintainers to Geoffrey Biggs and Tully Foote (#163)
- Contributors: Audrow Nash, Grey

pendulum_control

- Fix include order and relative paths for cpplint (#551)
- Remove the malloc_hook from the pendulum_demo. (#544)
- Update maintainers to Audrow Nash and Michael Jeronimo (#543)
- Additional fixes for documentation in demos. (#538)
- Fix documentation for pendulum_control. (#537)
- Contributors: Audrow Nash, Chris Lalancette, Jacob Perron

pendulum_msgs

- Update maintainers to Audrow Nash and Michael Jeronimo (#543)
- Contributors: Audrow Nash

pluginlib

- Install includes to include/${PROJECT_NAME} and remove ament_target_dependencies calls (#226)
- Require <memory> (#225)
- Move LibraryLoadExceptions down a level for more accurate error messages (#221)
- Update maintainers to Chris Lalancette (#223)
- extend termination condition to avoid infinite loop if package.xml is not found (#220)
- Remove deprecated headers. (#217)
- Contributors: Alberto Soragna, Audrow Nash, Chris Lalancette, David V. Lu!!, Shane Loretz

pybind11_vendor

- Use sha256 hash instead of tag (#12)
- Install headers to include/${PROJECT_NAME} (#11)
- Update pybind11 to 2.7.1. (#10) This is the version that is shipped in Ubuntu 22.04.
- Contributors: Chris Lalancette, Shane Loretz
python_cmake_module

- require Python 3.6 as we use format strings in various places (#10)
- Document all variables set by this module (#5)
- Add changelog (#4)
- Contributors: Ivan Santiago Paunovic, Shane Loretz, William Woodall

qt_gui_cpp

- Install headers to include$[PROJECT_NAME] (#259)
- Export targets instead of old-style CMake variables (#257)
- FindPython3 explicitly instead of FindPythonInterp implicitly (#254)
- Contributors: Shane Loretz

quality_of_service_demo_cpp

- Install includes to include$/PROJECT_NAME (#548)
- Fix include order and relative paths for cpplint (#551)
- Fixes for uncrustify 0.72 (#545)
- Update maintainers to Audrow Nash and Michael Jeronimo (#543)
- Additional fixes for documentation in demos. (#538)
- Fixing deprecated subscriber callback warnings (#532)
- Initialize message correctly (#522)
- Contributors: Abrar Rahman Protyasha, Audrow Nash, Chris Lalancette, Ivan Santiago Paunovic, Jacob Perron, Shane Loretz

quality_of_service_demo_py

- Update maintainers to Audrow Nash and Michael Jeronimo (#543)
- Update python nodes SIGINT handling (#539)
- Contributors: Audrow Nash, Ivan Santiago Paunovic

rcl

- add content-filtered-topic interfaces (#894)
- Add additional null check for timer argument (#973)
- Allow forward slashes within a parameter name rule in argument parsing (#860)
- Suppress false positive from clang-tidy (#951)
- Fix missing terminating 0 in rcl_context_impl_t.argv (#969)
- test_publisher_wait_all_ack depends on repputils (#968)
• Micro-optimizations in rcl (#965)
• If timer canceled, rcl_timer_get_time_until_next_call returns TIMER_CANCELED (#963)
• Add Events Executor (#839)
• Remove fastrtps customization on test_events (#960)
• Add client/service QoS getters (#941)
• Introduce ROS_DISABLE_LOAN_MSG to disable can_loan_messages. (#949)
• Install includes it include/\$[PROJECT_NAME] (#959)
• Make rcl_difference_times args const (#955)
• Update inject_on_return test skipping logic (#953)
• Fix jump callbacks being called when zero time jump thresholds used (#948)
• Only change the default logger level if default_logger_level is set (#943)
• Add Library for wait_for_entity_helpers to deduplicate compilation (#942)
• Increase Windows timeout 15 -> 25 ms (#940)
• Test should check specified number of entities. (#935)
• Fix up documentation build for rcl when using rosdock2 (#932)
• Include rmw_event_t instead of forward declaring it (#933)
• Add rcl_publisher_wait_for_all_acked support. (#913)
• Add tracing instrumentation for rcl_take. (#930)
• Fix #include in C++ typesupport example in rcl_subscription_init docblock. (#927)
• Update includes after rcutils/get_env.h deprecation. (#917)
• Use proper rcl_logging return value type and compare to constant. (#916)
• Contributors: Barry Xu, Chen Lihui, Chris Lalancette, Christophe Bedard, Haowei Wen, Ivan Santiago Paunovic, Jafar Abdi, Michel Hidalgo, Miguel Company, NoyZuberi, Scott K Logan, Shane Loretz, Tomoya Fujita, William Woodall, iRobot ROS, mauropasse

rcl_action

• Add Events Executor (#839)
• Install includes it include/\$[PROJECT_NAME] (#959)
• Fix up documentation build for rcl_action when using rosdock2 (#937)
• Fix expired goals capacity of action server (#931)
• Wait for action server in rcl_action comm tests. (#919)
• Contributors: Michel Hidalgo, Shane Loretz, iRobot ROS, spiralray
rcl_interfaces

- Update maintainers to Chris Lalancette (#130)
- Contributors: Audrow Nash

rcl_lifecycle

- Install includes it include/${PROJECT_NAME} (#959)
- [rcl_lifecycle] Do not share transition event message between nodes (#956)
- Update maintainers to Ivan Paunovic and William Woodall (#952)
- Fix up documentation build for rcl_lifecycle when using rosdoc2 (#938)
- Rename variable to fix name shadowing warning (#929)
- Contributors: Alberto Soragna, Audrow Nash, Ivan Santiago Paunovic, Michel Hidalgo, Shane Loretz

rcl_logging_interface

- Install includes to include/${PROJECT_NAME} (#85)
- Fix include order for cpplint (#84) Relates to https://github.com/ament/ament_lint/pull/324
- Update maintainers to Chris Lalancette (#83)
- Fix renamed rcpputils header (#81)
- Add Doxyfile to rcl_logging_interface package (#80)
- Update includes after rcutils/get_env.h deprecation (#75)
- Contributors: Abrar Rahman Protyasha, Audrow Nash, Christophe Bedard, Jacob Perron, Michel Hidalgo, Shane Loretz

rcl_logging_noop

- Update maintainers to Chris Lalancette (#83)
- Contributors: Audrow Nash

rcl_logging_spdlog

- Fix include order for cpplint (#84) Relates to https://github.com/ament/ament_lint/pull/324
- Update maintainers to Chris Lalancette (#83)
- Fix renamed rcpputils header (#81)
- Update includes after rcutils/get_env.h deprecation (#75)
- Contributors: Abrar Rahman Protyasha, Audrow Nash, Christophe Bedard, Jacob Perron
rcl_yaml_param_parser

- Install includes it include/${PROJECT_NAME} (#959)
- Update maintainers to Ivan Paunovic and William Woodall (#952)
- Tweak rcl_yaml_param_parser documentation (#939)
- Contributors: Audrow Nash, Michel Hidalgo, Shane Loretz

rclcpp

- remove DEFINE_CONTENT_FILTER cmake option (#1914)
- remove things that were deprecated during galactic (#1913)
- add take_data_by_entity_id API to waitable (#1892)
- add content-filtered-topic interfaces (#1561)
- [NodeParameters] Set name in param info pre-check (#1908)
- Add test-dep ament_cmake_google_benchmark (#1904)
- Add publish by loaned message in GenericPublisher (#1856)
- Add missing ament dependency on rcl_interfaces (#1903)
- Update data callback tests to account for all published samples (#1900)
- Increase timeout for acknowledgments to account for slower Connext settings (#1901)
- clang-tidy: explicit constructors (#1782)
- Add client/service QoS getters (#1784)
- Fix a bunch more rosdoc2 issues in rclcpp. (#1897)
- time_until_trigger returns max time if timer is cancelled (#1893)
- Micro-optimizations in rclcpp (#1896)
- spin_all with a zero timeout. (#1878)
- Add RMW listener APIs (#1579)
- Remove fastrtps customization on tests (#1887)
- Install headers to include/${PROJECT_NAME} (#1888)
- Use ament_generate_version_header (#1886)
- use universal reference to support rvalue. (#1883)
- fix one subscription can wait_for_message twice (#1870)
- Add return value version of get_parameter_or (#1813)
- Cleanup time source object lifetimes (#1867)
- add is_spinning() method to executor base class
- Cleanup the TypeAdapt tests (#1858)
- Cleanup includes (#1857)
- Fix include order and relative paths for cpplint (#1859)
• Rename stringstream in macros to a more unique name (#1862)
• Add non transform capabilities for intra-process (#1849)
• Fix rclcpp documentation build (#1779)
• Use UninitializedStaticallyTypedParameterException (#1689)
• Add wait_for_all_acked support (#1662)
• Add tests for function templates of declare_parameter (#1747)
• Fixes for uncrustify 0.72 (#1844)
• use private member to keep the all reference underneath. (#1845)
• Make node base sharable (#1832)
• Add Clock::sleep_for() (#1828)
• Synchronize rcl and std::chrono steady clocks in Clock::sleep_until (#1830)
• Use rclcpp::guard_condition (#1612)
• Call CMake function to generate version header (#1805)
• Use parantheses around logging macro parameter (#1820)
• Remove author by request (#1818)
• Update maintainers (#1817)
• min_forward & min_backward thresholds must not be disabled (#1815)
• Re-add Clock::sleep_until (#1814)
• Fix lifetime of context so it remains alive while its dependent node handles are still in use (#1754)
• Add the interface for pre-shutdown callback (#1714)
• Take message ownership from moved LoanedMessage (#1808)
• Suppress clang dead-store warnings in the benchmarks. (#1802)
• Wait for publisher and subscription to match (#1777)
• Fix unused QoS profile for clock subscription and make ClockQoS the default (#1801)
• Fix dangerous std::bind capture in TimeSource implementation. (#1768)
• Fix dangerous std::bind capture in ParameterEventHandler implementation. (#1770)
• Handle sigterm, in the same way sigint is being handled. (#1771)
• rclcpp::Node copy constructor: make copy of node_waitables_ member. (#1799)
• Extend NodeGraph to match what rcl provides. (#1484)
• Context::sleep_for(): replace recursion with do-while to avoid potential stack-overflow. (#1765)
• extend_sub_namespace(): Verify string::empty() before calling string::front(). (#1764)
• Deprecate the void shared_ptr<MsgType> subscription callback signatures. (#1713)
• Remove can_be_nullptr assignment check for QNX case. (#1752)
• Update client API to be able to remove pending requests. (#1734)
• Fix: Allow to add a node while spinning in the StaticSingleThreadedExecutor. (#1690)
• Add tracing instrumentation for executor and message taking. (#1738)
• Fix: Reset timer trigger time before execute in StaticSingleThreadedExecutor. (#1739)
• Use FindPython3 and make python3 dependency explicit. (#1745)
• Use rosidl_get_typesupport_target(). (#1729)
• Fix returning invalid namespace if sub_namespace is empty. (#1658)
• Add free function to wait for a subscription message. (#1705)
• Use rcpputils/scope_exit.hpp and remove rclcpp/scope_exit.hpp. (#1727)
• Remove unsafe get_callback_groups API. Callers should change to using for_each_callback_group(), or store the callback groups they need internally.
• Add in callback_groups_for_each. The main reason to add this method in is to make accesses to the callback_groups_ vector thread-safe. By having a callback_groups_for_each that accepts a std::function, we can just have the callers give us the callback they are interested in, and we can take care of the locking. The rest of this fairly large PR is cleaning up all of the places that use get_callback_groups() to instead use callback_groups_for_each().
• Use a different mechanism to avoid timers being scheduled multiple times by the MultiThreadedExecutor (#1692)
• Fix windows CI (#1726) Fix bug in AnyServiceCallback introduced in #1709.
• Support to defer to send a response in services. (#1709) Signed-off-by: Ivan Santiago Paunovic <ivan-pauno@ekumenlabs.com>
• Fix documentation bug. (#1719) Signed-off-by: William Woodall <william@osrfoundation.org>
• Removed left over is_initialized() implementation (#1711) Leftover from https://github.com/ros2/rclcpp/pull/1622
• Fixed declare parameter methods for int and float vectors (#1696)
• Cleaned up implementation of the intra-process manager (#1695)
• Added the node name to an executor runtime_error (#1686)
• Fixed a typo “Attack” -> “Attach” (#1687)
• Removed use of std::allocator<>::rebind (#1678) rebind is deprecated in c++17 and removed in c++20
• Allow declare uninitialized parameters (#1673)
• Fix syntax issue with gcc (#1674)
• [service] Don’t use a weak_ptr to avoid leaking (#1668)
• Fix doc typo (#1663)
• [rclcpp] Type Adaptation feature (#1557)
• Do not attempt to use void allocators for memory allocation. (#1657)
• Keep custom allocator in publisher and subscription options alive. (#1647)
• Fix get_publishers_subscriptions_info_by_topic test in test_node.cpp (#1648)
• Use OnShutdown callback handle instead of OnShutdown callback (#1639)
• use dynamic_pointer_cast to detect allocator mismatch in intra process manager (#1643)
• Contributors: Abrar Rahman Protyasha, Ahmed Sobhy, Alberto Soragna, Andrea Sorbini, Audrow Nash, Barry Xu, Bi0T1N, Chen Lihui, Chris Lalancette, Christophe Bedard, Doug Smith, Emerson Knapp, Gaël Écorchard, Geoffrey Biggs, Gonzo, Grey, Ivan Santiago Paunovic, Jacob Perron, Jorge Perez, Karsten Knese, Kenji Miyake, M. Hofstätter, M. Mostafa Farzan, Mauro Passerino, Michel Hidalgo, Miguel Company, Nikolai Morin, Petter
rclcpp_action

- remove things that were deprecated during galactic (1913)
- add take_data_by_entity_id API to waitable (1892)
- Fix rosdoc2 issues (1897)
- Add RMW listener APIs (1579)
- Install headers to include/PROJECT_NAME (1888)
- Fix include order and relative paths for cpplint (1859)
- Fixes for uncrustify 0.72 (1844)
- Use rclcpp::guard_condition (1612)
- Remove author by request (1818)
- Update maintainers (1817)
- Suppress clang dead-store warnings in the benchmarks. (1802)
- Deprecate the void shared_ptr<MessageT> subscription callback signatures (1713)
- Use rcpputils/scope_exit.hpp and remove rclcpp/scope_exit.hpp. (1727)
- Fixed occasionally missing goal result caused by race condition (1677)
- Bump the benchmark timeout for benchmark_action_client (1671)
- Returns CancelResponse::REJECT while goal handle failed to transit to CANCELING state (1641)
- Fix action server deadlock issue that caused by other mutexes locked in CancelCallback (1635)
- Contributors: Abrar Rahman Protyasha, Alberto Soragna, Chris Lalancette, Christophe Bedard, Jacob Perron, Kaven Yau, Shane Loretz, Tomoya Fujita, William Woodall, iRobot ROS, mauropasse

rclcpp_components

- Select executor in node registration (1898)
- Fix rosdoc2 issues in rclcpp (1897)
- Fix bugprone-exception-escape in node_main.cpp.in (1895)
- small improvements to node_main.cpp.in
- Install headers to include/PROJECT_NAME (1888)
- Use spin() in component_manager_isolated.hpp (1881)
- add use_global_arguments for node options of component nodes (1776)
- Add rclcpp_components::component (1855)
- Add parameter to configure number of thread (1708)
- remove RCLCPP_COMPONENTS_PUBLIC in class ComponentManagerIsolated (1843)
- create component_container_isolated (1781)
• Remove author by request (#1818)
• Update maintainers (#1817)
• Suppress clang dead-store warnings in the benchmarks. (#1802)
• Update client API to be able to remove pending requests. (#1734)
• Deprecate method names that use CamelCase in rclcpp_components. (#1716)
• Added a hook to generate node options in ComponentManager (#1702)
• Contributors: Alberto Soragna, Chris Lalancette, Daisuke Nishimatsu, Hirokazu Ishida, Ivan Santiago Paunovic, Jacob Perron, Rebecca Butler, Shane Loretz, gezp

**rclcpp_lifecycle**

• remove things that were deprecated during galactic (#1913)
• Fix rosdoc2 issues (#1897)
• Install headers to include/${PROJECT_NAME} (#1888)
• LifecycleNode::on_deactivate deactivate all managed entities. (#1885)
• Automatically transition lifecycle entities when node transitions (#1863)
• Remove author by request (#1818)
• Update maintainers (#1817)
• Suppress clang dead-store warnings in the benchmarks. (#1802)
• Update forward declarations of rcl_lifecycle types (#1788)
• Deprecate the void shared_ptr<MessageT> subscription callback signatures (#1713)
• Update client API to be able to remove pending requests. (#1734)
• Change log level for lifecycle_publisher. (#1715)
• Fix: RCLCPP_PUBLIC -> RCLCPP_LIFECYCLE_PUBLIC (#1732)
• Use rcpputils/scope_exit.hpp and remove rclcpp/scope_exit.hpp (#1727)
• Remove unsafe get_callback_groups API. Callers should change to using for_each_callback_group(), or store the callback groups they need internally.
• Add in callback_groups_for_each. The main reason to add this method in is to make accesses to the callback_groups_ vector thread-safe. By having a callback_groups_for_each that accepts a std::function, we can just have the callers give us the callback they are interested in, and we can take care of the locking. The rest of this fairly large PR is cleaning up all of the places that use get_callback_groups() to instead use callback_groups_for_each().
• Fix destruction order in lifecycle benchmark (#1675)
• [rclcpp] Type Adaptation feature (#1557)
• Contributors: Abrar Rahman Protyasha, Alberto Soragna, Audrow Nash, Chris Lalancette, Christophe Bedard, Ivan Santiago Paunovic, Jacob Perron, Michel Hidalgo, Shane Loretz, Tomoya Fujita, William Woodall
rclpy

- Make rclpy dependencies explicit (#906)
- Avoid exception in Node constructor when use override for ‘use_sim_time’ (#896)
- time_until_next_call returns max if timer is canceled. (#910)
- Properly implement action server/client handle cleanup. (#905)
- Make sure to take out contexts on Action[Client,Server]. (#904)
- Make sure to free the goal_status_array when done using it. (#902)
- Bugfix to Node.destroy_rate() result (#901)
- Remove fastrtps customization on tests (#895)
- fix typo (#890)
- Document that Future.result() may return None (#884)
- update doc release number (#885)
- Fix multi-threaded race condition in client.call_async (#871)
- Fix include order for cpplint (#877)
- Bugfix/duration to msg precision (#876)
- Update to pybind11 2.7.1 (#874)
- QoS history depth is only available with KEEP_LAST (#869)
- Implement managed nodes. (#865)
- Make rclpy.try_shutdown() behavior to follow rclpy.shutdown() more closely. (#868)
- Update TopicEndpointTypeEnum.__str__() method to include history kind and history depth. (#849)
- Add Clock.sleep_for() using Clock.sleep_until(). (#864)
- Add Clock.sleep_until() (#858)
- Add __enter__ and __exit__ to JumpHandle. (#862)
- Don’t override rclpy._rclpy_pybind11 docs. (#863)
- Improve JumpThreshold documentation and forbid zero durations. (#861)
- Fix time.py and clock.py circular import. (#860)
- Make context.on_shutdown() allow free functions. (#859)
- Fix automatically declared parameters descriptor type. (#853)
- Shutdown asynchronously when sigint is received. (#844)
- Update maintainers. (#845)
- Add entities to callback group before making them available to the executor to avoid a race condition. (#839)
- Avoid race condition in client.call(). (#838)
- Handle sigterm. (#830)
- Use pybind11 for signal handling, and delete now unused rclpy_common, pycapsule, and handle code. (#814)
- Fix memory leak in Service::take_request() and Client::take_response(). (#828)
- Add Publisher.wait_for_all_acked(). (#793)
• Only add one done callback to a future in Executor. (#816)
• Add convert function from ParameterValue to Python builtin. (#819)
• Call Context._logging_fini() in Context.try_shutdown(). (#800)
• Lift LoggingSeverity enum as common dependency to logging and rcutils_logger modules (#785)
• Set Context.__context to None in __init__(). (#812)
• Remove unused function make_mock_subscription. (#809)
• Removed common.c/h (#789)
• Allow declaring uninitialized parameters (#798)
• Reject cancel request if failed to transit to CANCEL.GOAL state (#791)
• Deleted handle as it should no longer be used (#786)
• Removed some functions in common.c and replaced them in utils.cpp (#787)
• Moved exception.cpp/hpp to the _rclpy_pybind11 module (#788)
• Print ‘Infinite’ for infinite durations in topic endpoint info (#722)
• Break log function execution ASAP if configured severity is too high (#776)
• Convert Node and Context to use C++ Classes (#771)
• Misc action server improvements (#774)
• Misc action goal handle improvements (#767)
• Convert Guardcondition to use C++ classes (#772)
• Removed unused structs rclpy_client_t and rclpy_service_t (#770)
• Convert WaitSet to use C++ Classes (#769)
• Convert ActionServer to use C++ Classes (#766)
• Convert ActionClient to use C++ classes (#759)
• Use py::class_ for rcl_action_goal_handle_t (#751)
• Convert Publisher and Subscription to use C++ Classes (#756)
• Rename QoS*Policy enum’s to *Policy (#379)
• Use params from node ‘/**’ from parameter YAML file (#370)
• Updated to use params from node ‘/**’ from parameter YAML file. (#399)
rcpputils

- Install includes to include/${PROJECT_NAME} (#160)
- Fix include order for cpplint (#158)
- [path] Declare the default assignment operator (#156)
- Fixes for uncrustify 0.72 (#154)
- Fix the BSD license headers to use the standard one. (#153)
- Update maintainers to Chris Lalancette (#152)
- Add checked convert_to_nanoseconds() function (#145)
- Add missing sections in docs/FEATURES.md TOC (#151)
- [env] Add set_env_var function (#150)
- Add missing cstddef include (#147)
- Add accumulator test to CMakeLists.txt (#144)
- rcpputils::fs: Fix doxygen parameter identifier (#142)
- Make thread safety macro C++ standards compliant (#141)
- Fix API documentation for clean rosd2 build (#139)
- Improve rcppmath Doxygen documentation (#138)
- Improve documentation of utilities in docs/FEATURES.md (#137)
- Include rcppmath utilities in docs/FEATURES.md (#136)
- Fix IllegalStateException reference in FEATURES (#135)
- migrate rolling mean from ros2_controllers to rcppmath (#133)
- Update includes after rcutils/get_env.h deprecation (#132)
- Contributors: Abrar Rahman Protyasha, Audrow Nash, Barry Xu, Chris Lalancette, Christophe Bedard, Jacob Perron, Karsten Knese, Octogonapus, Shane Loretz

rcutils

- Update launch test for change related to enviroment variables in launch (#354)
- Remove dst_size from strlen usage (#353)
- Install headers to include/${PROJECT_NAME} (#351)
- Use static_cast instead of C-style cast (#349)
- Fixing up documentation build when using rosd2 (#344)
- Stop double-defining structs. (#333)
- Use FindPython3 explicitly instead of FindPythonInterp implicitly (#345)
- Fix build on Android (#342)
- Deprecate get_env.h and move content to env.{h,c} (#340)
- Contributors: Chris Lalancette, Christophe Bedard, Ivan Santiago Paunovic, Jacob Perron, Jorge Perez, Shane Loretz, William Woodall
resource_retriever

- Install headers to include/${PROJECT_NAME} (#72)
- Fix include order for cpplint (#69)
- Update maintainers (#66)
- Remove the deprecated retriever.h header (#63)
- Contributors: Audrow Nash, Chris Lalancette, Jacob Perron, Shane Loretz

rmw

- Add content filtered topics support. (#302)
- Add sequence numbers to rmw_message_info_t. (#318)
- Add rmw_feature_supported(). (#318)
- Add EventsExecutor (#286)
- Document that rmw_wait() SHOULD use a monotonic clock (#316)
- Install headers to include/${PROJECT_NAME} (#317)
- Update rmw_server_is_available() API documentation. (#277)
- Add client/service QoS getters. (#314)
- Fix up documentation build for rmw when using rosdoc2 (#313)
- Fix up errors in doxygen documentation (#311)
- Fix copy-paste error in API doc for rmw_get_gid_for_publisher (#310)
- Add rmw_publisher_wait_for_all_acked support. (#296)
- Contributors: Barry Xu, Chen Lihui, Chris Lalancette, Christophe Bedard, Ivan Santiago Paunovic, Michel Hidalgo, Shane Loretz, iRobot ROS, mauropasse

rmw_connextdds

- Exclude missing sample info fields when building rmw_connextddsmicro (#79)
- Update launch_testing_ros output filter prefixes for Connext6 (#80)
- Add support for user-specified content filters (#68)
- Add rmw listener apis (#44)
- Add client/service QoS getters. (#67)
- Add rmw_publisher_wait_for_all_acked support. (#20)
- Contributors: Andrea Sorbini, Barry Xu, Chen Lihui, Ivan Santiago Paunovic, iRobot ROS, mauropasse
**rmw_connextdds_common**

- Exclude missing sample info fields when building `rmw_connextddsmicro` (#79)
- Properly initialize CDR stream before using it for filtering (#81)
- Add support for user-specified content filters (#68)
- Add stub for content filtered topic (#77)
- Add sequence numbers to message info structure (#74)
- Add `rmw` listener apis (#44)
- Fix cpplint errors (#69)
- Add client/service QoS getters. (#67)
- Update `rmw_context_impl_t` definition (#65)
- Use the new `rmw_dds_common::get_security_files` API (#61)
- Add `rmw_publisher_wait_for_all_acked` support. (#20)
- Support extended signature for `message_type_support_callbacks_t::max_serialized_size()` from `rosidl_typesupport_fastrtps_cpp`. (#14)
- Update includes after rcutils/get_env.h deprecation. (#55)
- Always modify `UserObjectQosPolicy` regardless of override policy. (#53)
- Improved conversion of time values between ROS and DDS formats. (#43)
- Allow sharing DomainParticipant with C++ applications. (#25)
- Add environment variable to control override of `DomainParticipantQos`. (#41)

**Contributors:** Andrea Sorbini, Barry Xu, Chen Lihui, Chris Lalancette, Christophe Bedard, Ivan Santiago Paunovic, Jacob Perron, Michel Hidalgo, Miguel Company, iRobot ROS, mauropasse

**rmw_connextddsmicro**

- Exclude missing sample info fields when building `rmw_connextddsmicro` (#79)
- Add support for user-specified content filters (#68)
- Add stub for content filtered topic (#77)
- Add sequence numbers to message info structure (#74)
- Add `rmw` listener apis (#44)
- Add client/service QoS getters. (#67)
- Add `rmw_publisher_wait_for_all_acked` support. (#20)

**Contributors:** Andrea Sorbini, Barry Xu, Chen Lihui, Ivan Santiago Paunovic, iRobot ROS, mauropasse
rmw_cyclonedds_cpp

- Fix `get_topic_name` and handling long service names
- Add serialization for `SDK_DATA`
- Additional checks for loan API
- Depend on just `rmw.dds_common::rmw.dds_common_library` (#385)
- Fix error message in `rmw_init_options_copy()`. (#380)
- Add content filter topic feature empty stub. (#289)
- Update to work with Cyclone 0.9.0 and Iceoryx 2.0 (#379)
- Fill message info sequence numbers as unsupported, add `rmw_feature_supported()` implementation. (#381)
- Fix a warning by making a pointer `nullptr`. (#375)
- Bump QDs to QL2 (#371)
- Add `EventsExecutor` (#256)
- Call `dissociate_reader` in `rmw_destroy_subscription`
- Wrap creation of new `serdata_rmw` within a try-catch block
- Fix memory leak in error scenario on the publish side with SHM
- Fix memory leaks on the take side with SHM
- Rename `_cyclonedds_has_shm` to follow the convention
- Add `iceoryx_binding_c` as dependency to `rmw_cyclonedds_cpp`
- Release `iox_chunk` to `iceoryx` in `serdata_free` if the `iox_chunk` is still available
- Update `iceoryx_subscriber` also when constructing the `serdata` from the `iox_chunk`
- Fix cpplint errors (#363)
- Updates for uncrustify 0.72 (#358)
- Export only `rmw::rmw` to downstream targets (#360)
- Export modern CMake targets (#357)
- Free with the same allocator in `rmw_destroy_node` (#355)
- Add client/service QoS getters. (#343)
- Updated version number and quality level. (#349)
- Update package maintainers. (#351)
- Fix undesired memory initialization in zero-copy data path. (#348)
- Fix QoS depth settings for clients/service being ignored. (#340)
- Link to Cyclone DDS in Quality Declaration. (#342)
- Update `rmw_context_impl_t` definition (#337)
- Add quality declaration for `rmw_cyclonedds_cpp` (#335)
- Fix use of deprecated `is_loan_available` (#336)
- Add `-latomic` for RISC-V (#332)
- Add pub/sub init, publish and take instrumentation using tracetools (#329)
• Pass the CRL down to CycloneDDS if it exists (#325)
• Use the new rmw_dds_common::get_security_files API (#323)
• Add rmw_publisher_wait_for_all_acked support. (#294)
• Fix zero copy issues. (#309)
• Handle allocation errors during message deserialization. (#313)
• Update includes after rcutils/get_env.h deprecation. (#312)
• Contributors: Barry Xu, Chen Lihui, Chris Lalancette, Christophe Bedard, Dietrich Krönke, Erik Boasson, Haowei Wen, Ivan Santiago Paunovic, Jacob Perron, Joe Speed, Michel Hidalgo, Shane Loretz, Sumanth Nirmal, eboasson, guillaume-pais-siemens, iRobot ROS, mauropasse

rmw_dds_common

• Depend on target generated by rosidl_typesupport_cpp (#58)
• Use rosidl_get_typesupport_target() and target_link_libraries(). (#57)
• Install headers to include/${PROJECT_NAME} (#56)
• Fix include order for cpplint (#55)
• Fix up rmw_dds_common documentation when using rosdock2 (#54)
• Add support for Certificate Revocation List files (#52)
• Silence clang warning (range-loop-construct) (#53)
• Add a common function for security files. (#51)
• Normalize rmw_time_t according to DDS spec (#48)
• Contributors: Andrea Sorbini, Chris Lalancette, Jacob Perron, Karsten Knese, Michel Hidalgo, Shane Loretz

rmw_fastrtps_cpp

• Add pub/sub init, publish and take instrumentation using tracetools (#591)
• Add content filter topic feature (#513)
• Add sequence numbers to message info structure (#587)
• Removed some heap interactions in rmw_serialize.cpp (#590)
• Add EventsExecutor (#468)
• Install headers to include/${PROJECT_NAME} (#578)
• Add client/service QoS getters. (#560)
• Correctly recalculate serialized size on bounded sequences. (#540)
• Fix type size alignment. (#550)
• Change links from index.ros.org -> docs.ros.org (#539)
• Add rmw_publisher_wait_for_all_acked support. (#519)

• Contributors: Barry Xu, Chen Lihui, Chris Lalancette, Christophe Bedard, Ivan Santiago Paunovic, Miguel Company, Shane Loretz, WideAwakeTN, iRobot ROS, mauropasse

rmw_fastrtps_dynamic_cpp

• Add content filter topic feature (#513)
• Add sequence numbers to message info structure (#587)
• Add EventsExecutor (#468)
• Install headers to include/${PROJECT_NAME} (#578)
• Add client/service QoS getters. (#560)
• Correctly recalculate serialized size on bounded sequences. (#540)
• Fix type size alignment. (#550)
• Add rmw_publisher_wait_for_all_acked support. (#519)


• Contributors: Barry Xu, Chen Lihui, Ivan Santiago Paunovic, Miguel Company, Shane Loretz, iRobot ROS, mauropasse

rmw_fastrtps_shared_cpp

• Address linter waning for windows. (#592)
• Add pub/sub init, publish and take instrumentation using tracetools (#591)
• Add content filter topic feature (#513)
• Add sequence numbers to message info structure (#587)
• Add EventsExecutor (#468)
• Complete events support (#583)
• Install headers to include/${PROJECT_NAME} (#578)
• Change default to synchronous (#571)
• Fix cpplint error (#574)
• Fixes for uncrustify 0.72 (#572)
• Add client/service QoS getters. (#560)
• Fix QoS depth settings for clients/service being ignored. (#564)
• Update rmw_context_impl_t definition. (#558)
• Update the LoanManager to do internal locking. (#552)
• Pass the CRL down to Fast-DDS if available. (#546)
• Use the new rmw_dds_common::get_security_files (#544)
• Support for SubscriptionOptions::ignore_local_publications (#536)
• Change links from index.ros.org -> docs.ros.org (#539)
• Add rmw_publisher_wait_for_all_acked support. (#519)
• Export rmw_dds_common as an rmw_fastrtps_shared_cpp dependency (#530)
• Update includes after rcutils/get_env.h deprecation (#529)
• Contributors: Audrow Nash, Barry Xu, Chen Lihui, Chris Lalancette, Christophe Bedard, Ivan Santiago Paunovic, Jacob Perron, Jose Antonio Moral, Michel Hidalgo, Miguel Company, Shane Loretz, Tomoya Fujita, iRobot ROS, mauropasse

rmw_implementation

• add content-filtered-topic interfaces (#181)
• Add rmw_feature_supported() (#204)
• Add EventsExecutor (#161)
• Fix relative path include syntax for cpplint (#203)
• Support and prefer exported targets from rmw implementations (#201)
• Add client/service QoS getters. (#196)
• Update maintainers to Audrow Nash and Michael Carroll. (#199)
• Fix renamed rcpputils header (#198)
• Fix rmw_implementation generated documentation (#197)
• Add rmw_publisher_wait_for_all_acked. (#188)
• Attempt to load any available RMW implementation. (#189)
• Update includes after rcutils/get_env.h deprecation (#190)
• Contributors: Abrar Rahman Protyasha, Audrow Nash, Barry Xu, Chen Lihui, Chris Lalancette, Christophe Bedard, Ivan Santiago Paunovic, Jacob Perron, Michel Hidalgo, Shane Loretz, iRobot ROS, mauropasse

rmw_implementation_cmake

• Use FastDDS as default DDS (#315)
• Contributors: Audrow Nash
robot_state_publisher

- Depend on orocos_kdl_vendor (#191)
- export dependencies, to use robot_state_publisher as a component (#193)
- Fix include order for cpplint (#186)
- Change how parameter updates are handled (#180)
- Install includes to instal/${PROJECT_NAME} (#184)
- Make the change_fixed_joint test more robust (#183)
- Add in a test to make sure fixed transforms change on update
- Small C++ nice-isms in the tests
- Switch to using target_include_directories for tests
- Publish new fixed transforms when URDF is updated
- Make joint_states subscription QoS configurable; default to SensorDataQoS (#179)
- Remove dependency on urdfdom_headers (#168)
- Fix deprecated subscriber callbacks (#173)
- Cleanup the documentation in the RobotStatePublisher class. (#172)
- Always publish fixed frames to /tf_static (#158)
- corrected publish_frequency default in README (#166)
- Add tf frame_prefix parameter (#159)
- Contributors: Abrar Rahman Protyasha, Anthony Deschamps, Chris Lalancette, Jacob Perron, Kenji Brameld, Nils Schulte, Russell Joyce, Shane Loretz, Steve Nogar

ros2action

- Add timeout to kill hanging tests (#701)
- Depend on launch packages instead of ros_testing to avoid circular dependency (#685)
- Update maintainers to Aditya Pande, Audrow Nash, and Michael Jeronimo (#673)
- Updated maintainers (#670)
- Add changelogs (#635)
- Contributors: Aditya Pande, Audrow Nash, Ivan Santiago Paunovic, Shane Loretz
### ros2bag

- Support to publish as loaned message (#981)
- Revert “Add the ability to record any key/value pair in the ‘custom’ field in metadata.yaml (#976)” (#984)
- Add the ability to record any key/value pair in the ‘custom’ field in metadata.yaml (#976)
- Support to publish as loaned message (#981)
- Revert “Add the ability to record any key/value pair in the ‘custom’ field in metadata.yaml (#976)” (#984)
- Add the ability to record any key/value pair in the ‘custom’ field in metadata.yaml (#976)
- Bump version number to avoid conflict
- Make sure published messages are acknowledged for play mode (#951)
- TopicFilter use regex_search instead of regex_match (#932)
- Add start-offset play option (#931)
- Expose bag_rewrite as ros2 bag convert (#921)
- Add “ignore leaf topics” option to recorder (#925)
- Auto-detect storage_id for Reader (if possible) (#918)
- Add pause/resume options to the bag recorder (#905)
- Add –start-paused option to ros2 bag play (#904)
- Update package maintainers (#899)
- Fix converter plugin choices for record (#897)
- Add missing spaces to error message (#875)
- Keyboard controls for pause/resume toggle and play-next: (#847)
- Add –snapshot-mode argument to the “record” verb (#851)
- Refactor plugin query mechanism and standardize trait management (#833)
- Update PlayOptions::delay to rclcpp::Duration to get nanosecond resolution (#843)
- Load compression and serialization choices via plugin query (#827)
- Add delay option (#789)
- Avoid passing exception KeyboardInterrupt to the upper layer (#788)

**Contributors:** Abrar Rahman Protyasha, Audrow Nash, Barry Xu, Cameron Miller, Chris Lalancette, Emerson Knapp, Ivan Santiago Paunovic, Jacob Perron, Jorge Perez, Kosuke Takeuchi, Michel Hidalgo, Sonia Jin, Tony Peng
**ros2cli**

- Fix importlib_metadata warning on Python 3.10. (#706)
- Add timeout to kill hanging tests (#701)
- Use try_shutdown() instead of shutdown() in DirectNode.__exit__() (#683)
- Update maintainers to Aditya Pande, Audrow Nash, and Michael Jeronimo (#673)
- Updated maintainers (#670)
- Reapply #659 (#661)
- Revert “Make the ros2cli output always line buffered (#659)” (#660)
- Make the ros2cli output always line buffered (#659)
- Add uuid to ros2 daemon node name. (#658)
- Transfer daemon socket ownership on spawn. (#652)
- Add changelogs (#635)
- Contributors: Aditya Pande, Audrow Nash, Chris Lalancette, Ivan Santiago Paunovic, Michel Hidalgo, Tomoya Fujita

**ros2cli_test_interfaces**

- Update maintainers to Aditya Pande, Audrow Nash, and Michael Jeronimo (#673)
- Updated maintainers (#670)
- Add changelogs (#635)
- Contributors: Aditya Pande, Audrow Nash, Ivan Santiago Paunovic

**ros2component**

- Add timeout to kill hanging tests (#701)
- Update maintainers to Aditya Pande, Audrow Nash, and Michael Jeronimo (#673)
- Updated maintainers (#670)
- Drop deprecated get_container_components_info() API. (#647)
- Add changelogs (#635)
- Contributors: Aditya Pande, Audrow Nash, Ivan Santiago Paunovic, Michel Hidalgo
**ros2doctor**

- Fix importlib_metadata warning on Python 3.10. (#706)
- Add timeout to kill hanging tests (#701)
- Switch ros2 doctor to using psutil for network checks. (#687)
- Depend on launch packages instead of ros_testing to avoid circular dependency (#685)
- Update maintainers to Aditya Pande, Audrow Nash, and Michael Jeronimo (#673)
- Updated maintainers (#670)
- Add changelogs (#635)
- Improve ros2 doctor on Windows. (#631)
- Add QoS compatibility check and report. (#621)
- Contributors: Aditya Pande, Alberto Soragna, Audrow Nash, Chris Lalancette, Ivan Santiago Paunovic, Shane Loretz

**ros2interface**

- Add timeout to kill hanging tests (#701)
- Depend on launch packages instead of ros_testing to avoid circular dependency (#685)
- Update maintainers to Aditya Pande, Audrow Nash, and Michael Jeronimo (#673)
- Updated maintainers (#670)
- Add changelogs (#635)
- Contributors: Aditya Pande, Audrow Nash, Ivan Santiago Paunovic, Shane Loretz

**ros2launch**

- Update maintainers in setup.py (#287)
- Use frontend group dependency & explicit dependencies in ros2launch (#256)
- Update package maintainers (#284)
- Add regex filter for selective launch-prefix application (#261)
- Resolves #37 - Added --launch-prefix argument for ‘ros2 launch’ command (#254)
- Use sets of file extensions provided by parser extensions (#252)
- Simplify logic to fix absolute paths (#230)
- add way to include other Python launch files (launch #122)
- Contributors: Audrow Nash, Cameron Miller, Christophe Bedard, Michel Hidalgo, rob-clarke
ros2lifecycle

- Add timeout to kill hanging tests (#701)
- Depend on launch packages instead of ros_testing to avoid circular dependency (#685)
- Update maintainers to Aditya Pande, Audrow Nash, and Michael Jeronimo (#673)
- Updated maintainers (#670)
- Add changelogs (#635)
- Contributors: Aditya Pande, Audrow Nash, Ivan Santiago Paunovic, Shane Loretz

ros2lifecycle_test_fixtures

- Update maintainers to Aditya Pande, Audrow Nash, and Michael Jeronimo (#673)
- Updated maintainers (#670)
- Add changelogs (#635)
- Contributors: Aditya Pande, Audrow Nash, Ivan Santiago Paunovic

ros2multicast

- Add timeout to kill hanging tests (#701)
- Update maintainers to Aditya Pande, Audrow Nash, and Michael Jeronimo (#673)
- Updated maintainers (#670)
- Add changelogs (#635)
- Contributors: Aditya Pande, Audrow Nash, Ivan Santiago Paunovic

ros2node

- Add timeout to kill hanging tests (#701)
- Depend on launch packages instead of ros_testing to avoid circular dependency (#685)
- Update maintainers to Aditya Pande, Audrow Nash, and Michael Jeronimo (#673)
- Updated maintainers (#670)
- Add changelogs (#635)
- Contributors: Aditya Pande, Audrow Nash, Ivan Santiago Paunovic, Shane Loretz
**ros2param**

- Add timeout to kill hanging tests (#701)
- Fix how ros2 param interprets command-line arguments. (#684)
- Update maintainers to Aditya Pande, Audrow Nash, and Michael Jeronimo (#673)
- Updated maintainers (#670)
- Removed redundant code (#666)
- Reapply #659 (#661)
- Fix flaky ros2 param list (#656)
- Skip None Result (#646)
- add ‘--write’ option to avoid an unintentional data loss. (#638)
- Add changelogs (#635)
- Contributors: Aditya Pande, Audrow Nash, Chris Lalancette, Ivan Santiago Paunovic, Jacob Perron, Jay Wang, Tomoya Fujita

**ros2pkg**

- Add timeout to kill hanging tests (#701)
- Use local git config instead of global (#693)
- Depend on launch packages instead of ros_testing to avoid circular dependency (#685)
- [ros2pkg] Skip copyright tests in template packages (#676)
- Update maintainers to Aditya Pande, Audrow Nash, and Michael Jeronimo (#673)
- Updated maintainers (#670)
- Generate LICENSE files on ros2 pkg create. (#650)
- Handle ValueError (#643)
- Pass package exports to template in pkg create api (#619) (#628)
- Add changelogs (#635)

**ros2run**

- Add timeout to kill hanging tests (#701)
- Update maintainers to Aditya Pande, Audrow Nash, and Michael Jeronimo (#673)
- Updated maintainers (#670)
- check subprocess.returncode to print error message. (#639)
- Add changelogs (#635)
- Contributors: Aditya Pande, Audrow Nash, Ivan Santiago Paunovic, Tomoya Fujita
ros2service

- Add timeout to kill hanging tests (#701)
- Also provide `--include-hidden-services` for `ros2 service list` verb (#551)
- Depend on launch packages instead of ros_testing to avoid circular dependency (#685)
- Update maintainers to Aditya Pande, Audrow Nash, and Michael Jeronimo (#673)
- Updated maintainers (#670)
- Add changelogs (#635)
- Contributors: Aditya Pande, Audrow Nash, Ivan Santiago Paunovic, Karsten Knese, Shane Loretz

ros2test

- Use new domain_coordinator API (#10)
- Contributors: Timo Röhling

ros2topic

- Add timeout to kill hanging tests (#701)
- Add yaml dump flow style. (#698)
- Support ros2topic echo once option. (#695)
- Fix special case for fastrtps incompatible QoS. (#694)
- Depend on launch packages instead of ros_testing to avoid circular dependency (#685)
- Add QoS history and depth information if available. (#678)
- Cleanup mislabeled BSD license (#447)
- Update maintainers to Aditya Pande, Audrow Nash, and Michael Jeronimo (#673)
- Updated maintainers (#670)
- Update lost messages test case (#669)
- Implementation of message filtering for ros2 (#654)
- Change default QoSProfile for pub (#653)
- Add option in ros2 topic pub to wait for N matching subscriptions, use N=1 by default when combined with `--times` (#642)
- ros2 topic pub starts publishing right away. (#626)
- Fix Topic Info Test with “Infinite” printing (#616)
- Add changelogs (#635)
- QoS autodetection (#613)
- Make Lost Messages option ON by default (#633)
- Contributors: Aditya Pande, Audrow Nash, Chris Lalancette, Emerson Knapp, Gonzo, Ivan Santiago Paunovic, Jorge Perez, Shane Loretz, Tomoya Fujita, Tully Foote, matthews-jca
ros2trace

- Fix ‘ros2 trace’ fini() error
- Don’t require kernel tracer and detect when it’s not installed
- Deprecate ‘context_names’ param and replace with ‘context_fields’
- Contributors: Christophe Bedard

rosbag2

- Bump version number to avoid conflict
- Update package maintainers (#899)
- Contributors: Chris Lalancette, Michel Hidalgo

rosbag2_compression

- Bump version number to avoid conflict
- Install headers to include/${PROJECT_NAME} (#958)
- Remove unnecessary public definition. (#950)
- Changes for uncrustify 0.72 (#937)
- Bugfix for broken bag split when using cache (#936)
- Update package maintainers (#899)
- Don’t preprocess a storage file more than once (#895)
- added seek interface (#836)
- Refactor plugin query mechanism and standardize trait management (#833)
- fix sequential reader rollover-to-next-file strategy: (#839)
- Load compression and serialization choices via plugin query (#827)
- Contributors: Cameron Miller, Chris Lalancette, Michael Orlov, Michel Hidalgo, Shane Loretz, sonia

rosbag2_compression_zstd

- Bump version number to avoid conflict
- Install headers to include/${PROJECT_NAME} (#958)
- Update package maintainers (#899)
- Contributors: Chris Lalancette, Michel Hidalgo, Shane Loretz
rosbag2_cpp

• Revert “Add the ability to record any key/value pair in the ‘custom’ field in metadata.yaml (#976)” (#984)
• Add the ability to record any key/value pair in the ‘custom’ field in metadata.yaml (#976)
• Revert “Add the ability to record any key/value pair in the ‘custom’ field in metadata.yaml (#976)” (#984)
• Add the ability to record any key/value pair in the ‘custom’ field in metadata.yaml (#976)
• Bump version number to avoid conflict
• Install headers to include/${PROJECT_NAME} (#958)
• Remove unnecessary public definition. (#950)
• Fix relative path syntax for cpplint (#947)
• Mark up the message_cache with TSA annotations (#946)
• Changes for uncrustify 0.72 (#937)
• Redesign in cache consumer and circular message cache to get rid from busy loop (#941)
• Bugfix for broken bag split when using cache (#936)
• Remove JumpHandler copy-implementation from PlayerClock/TimeControllerClock (#935)
• Auto-detect storage_id for Reader (if possible) (#918)
• Add –start-paused option to ros2 bag play (#904)
• Use the message_introspection header to get MessageMember. (#903)
• Update package maintainers (#899)
• Fix converter plugin choices for record (#897)
• Enable sanitizers only if code actually can run (#572)
• Need to pass introspection TS to converter plugin for it to be useful (#896)
• Don’t preprocess a storage file more than once (#895)
• Fix a bug on invalid pointer address when using “MESSAGE” compression... (#866)
• Metadata per file info (#870)
• Fix TSA warnings when building with clang thread analysis. (#877)
• Implement snapshot mechanism and corresponding ROS Service (#850)
• Circular Message Cache implementation for snapshot feature (#844)
• Fix discovery silently stops after unknown msg type is found. (#848)
• added seek interface (#836)
• Refactor plugin query mechanism and standardize trait management (#833)
• fix sequential reader rollover-to-next-file strategy: (#839)
• Load compression and serialization choices via plugin query (#827)
• Workaround for false positive findings by clang thread safety analysis in time controller jump callbacks API. (#799)
• Add callbacks for PlayerClock::jump(time_point) API with CI fix (#779)
• Revert “Add callbacks for PlayerClock::jump(time_point) API (#775)” (#778)
• Add callbacks for PlayerClock::jump(time_point) API (#775)
• Contributors: Audrow Nash, Barry Xu, Cameron Miller, Chris Lalancette, Emerson Knapp, Ivan Santiago Paunovic, Jacob Perron, Jorge Perez, Lei Liu, Michael Orlov, Michel Hidalgo, Shane Loretz, Tony Peng, Wojciech Jaworski, sonia

rosbag2_interfaces

• Bump version number to avoid conflict
• Add burst-mode to Player (#977)
• Update package maintainers (#899)
• Implement snapshot mechanism and corresponding ROS Service (#850) * Add snapshot service to recorder node
  * Simplify and clarify double buffering patterns
• Contributors: Cameron Miller, Chris Lalancette, Geoffrey Biggs, Michel Hidalgo

rosbag2_performance_benchmarking

• Bump version number to avoid conflict
• Install headers to include/${PROJECT_NAME} (#958)
• Enable YAML encoding/decoding for RecordOptions and StorageOptions (#916) * Enable YAML encoding/decoding for RecordOptions and StorageOptions
• Update package maintainers (#899)
• Updated node declare parameter to new syntax (#882)
• Updated benchmark package to use writer close() instead of old reset() (#881)
• Contributors: Adam Dąbrowski, Chris Lalancette, Emerson Knapp, Michel Hidalgo, Shane Loretz

rosbag2_py

• support to publish as loaned message (#981)
• Revert “Add the ability to record any key/value pair in the ‘custom’ field in metadata.yaml (#976)” (#984)
• Add the ability to record any key/value pair in the ‘custom’ field in metadata.yaml (#976)
• support to publish as loaned message (#981)
• Revert “Add the ability to record any key/value pair in the ‘custom’ field in metadata.yaml (#976)” (#984)
• Add the ability to record any key/value pair in the ‘custom’ field in metadata.yaml (#976)
• Bump version number to avoid conflict
• Make sure published messages are acknowledged for play mode (#951)
• Fix relative path syntax for cpplint (#947)
• Update to pybind11 2.7.1 (#945)
• Add start-offset play option (#931)
• Expose bag_rewrite as ros2 bag convert (#921)
Vulcanexus Documentation, Release 1.0.0

• Add “ignore leaf topics” option to recorder (#925)
• Add a ReaderWriterFactory utility to share logic for reuse (#923)
• Add pause/resume options to the bag recorder (#905)
• Add –start-paused option to ros2 bag play (#904)
• Update package maintainers (#899)
• Fix converter plugin choices for record (#897)
• Metadata per file info (#870)
• keyboard controls for pause/resume toggle and play-next: (#847)
• Add –snapshot-mode argument to the “record” verb (#851)
• Add stopRecording into rosbag2_py (#854)
• added seek interface (#836)
• Refactor plugin query mechanism and standardize trait management (#833)
• Update PlayOptions::delay to rclcpp::Duration to get nanosecond resolution (#843)
• Load compression and serialization choices via plugin query (#827)
• Add delay option (#789)
• Handle SIGTERM gracefully in recording (#792)
• Contributors: Abrar Rahman Protyasha, Afonso da Fonseca Braga, Audrow Nash, Barry Xu, Cameron Miller, Chris Lalancette, Emerson Knapp, Ivan Santiago Paunovic, Jacob Perron, Jorge Perez, Kosuke Takeuchi, Michel Hidalgo, Tony Peng, Wojciech Jaworski, sonia

rosbag2_storage

• Revert “Add the ability to record any key/value pair in the ‘custom’ field in metadata.yaml (#976)” (#984)
• Add the ability to record any key/value pair in the ‘custom’ field in metadata.yaml (#976)
• Revert “Add the ability to record any key/value pair in the ‘custom’ field in metadata.yaml (#976)” (#984)
• Add the ability to record any key/value pair in the ‘custom’ field in metadata.yaml (#976)
• Bump version number to avoid conflict
• Install headers to include/${PROJECT_NAME} (#958)
• Remove unnecessary public definition. (#950)
• Enable YAML encoding/decoding for RecordOptions and StorageOptions (#916)
• Update package maintainers (#899)
• Provide MetadataIO interface to convert metadata to a string in memory, alongside file IO versions (#894)
• Metadata per file info (#870)
• Implement snapshot mechanism and corresponding ROS Service (#850)
• added seek interface (#836)
• Refactor plugin query mechanism and standardize trait management (#833)
• Contributors: Audrow Nash, Cameron Miller, Chris Lalancette, Emerson Knapp, Jorge Perez, Michel Hidalgo, Shane Loretz, Tony Peng, Wojciech Jaworski, sonia
rosbag2_storage_default_plugins

- Bump version number to avoid conflict
- Install headers to include/${PROJECT_NAME} (#958)
- Emit a warning rather than crash when a message is too big for sqlite (#919)
- Enable YAML encoding/decoding for RecordOptions and StorageOptions (#916)
- Update package maintainers (#899)
- added seek interface (#836)
- Contributors: Chris Lalancette, Emerson Knapp, Michel Hidalgo, Shane Loretz, William Woodall, sonia

rosbag2_test_common

- Bump version number to avoid conflict
- Install headers to include/${PROJECT_NAME} (#958)
- Update package maintainers (#899)
- Make sure the subscription exists before publishing messages (#804)
- Handle SIGTERM gracefully in recording (#792)
- Add spin_and_wait_for_matched to PublicationManager and update test c… (#797)
- Avoid passing exception KeyboardInterrupt to the upper layer (#788)
- Contributors: Barry Xu, Chris Lalancette, Emerson Knapp, Michel Hidalgo, Shane Loretz

rosbag2_tests

- Revert “Add the ability to record any key/value pair in the ‘custom’ field in metadata.yaml (#976)” (#984)
- Add the ability to record any key/value pair in the ‘custom’ field in metadata.yaml (#976)
- Revert “Add the ability to record any key/value pair in the ‘custom’ field in metadata.yaml (#976)” (#984)
- Add the ability to record any key/value pair in the ‘custom’ field in metadata.yaml (#976)
- Bump version number to avoid conflict
- Add pause/resume options to the bag recorder (#905)
- Update package maintainers (#899)
- Fix a bug on invalid pointer address when using “MESSAGE” compressio… (#866)
- Metadata per file info (#870)
- Fix record test to reflect plugin query changes (#838)
- Make sure the subscription exists before publishing messages (#804)
- Handle SIGTERM gracefully in recording (#792)
- Add spin_and_wait_for_matched to PublicationManager and update test c… (#797)
- Remove rmw_fastrtps_cpp find_package in rosbag2_tests (#774)
• Contributors: Audrow Nash, Barry Xu, Cameron Miller, Chris Lalancette, Emerson Knapp, Ivan Santiago Paunovic, Jorge Perez, Michel Hidalgo, Tony Peng, Wojciech Jaworski

rosbag2_transport

• support to publish as loaned message (#981)
• support to publish as loaned message (#981)
• Bump version number to avoid conflict
• Add burst-mode to Player (#977)
• Install headers to include/${PROJECT_NAME} (#958)
• Make sure published messages are acknowledged for play mode (#951)
• Changes for uncrustify 0.72 (#937)
• TopicFilter use regex_search instead of regex_match (#932)
• Add start-offset play option (#931)
• Add parentheses suggested by Clang on OSX to fix build warning (#930)
• Bag rewriter (C++) (#920)
• Add “ignore leaf topics” option to recorder (#925)
• Rewrite TopicFilter for single-call reusability (#924)
• Add a ReaderWriterFactory utility to share logic for reuse (#923)
• Add pause/resume options to the bag recorder (#905)
• Add logging macros for rosbag2_transport (#917)
• Enable YAML encoding/decoding for RecordOptions and StorageOptions (#916)
• Expose the QoS object wrapper (#910)
• Add –start-paused option to ros2 bag play (#904)
• Update package maintainers (#899)
• Add a Seek service (#874)
• Add simple keyboard control for playback rate (#893)
• Fix a bug on invalid pointer address when using “MESSAGE” compressio… (#866)
• Fix typo (#880)
• Use Reader’s seek() method for seeking/jumping in Player (#873)
• keyboard controls for pause/resume toggle and play-next: (#847)
• Implement snapshot mechanism and corresponding ROS Service (#850)
• Circular Message Cache implementation for snapshot feature (#844)
• Add jump/seek API for Player class (#826)
• Restructure test_play_timing to one test per case, to see which times out (#863)
• Fix discovery silently stops after unknown msg type is found. (#848)
• Fixing deprecated subscriber callback warnings (#852)
• Bugfix for race condition in Player::peek_next_message_from_queue() (#849)
• added seek interface (#836)
• Update PlayOptions::delay to rclcpp::Duration to get nanosecond resolution (#843)
• Move notification about ready for playback inside play_messages_from_queue() (#832)
• Add wait for player to be ready for playback in Player::play_next() method (#814)
• Make sure the subscription exists before publishing messages (#804)
• Add delay option (#789)
• Copy recorder QoS profile to local variable so that temporary value isn’t cleared (#803)
• test_play_services: fail gracefully on future error (#798)
• Recording with --all and --exclude fix (#765)

rosgraph_msgs

• Update maintainers to Chris Lalancette (#130)
• Contributors: Audrow Nash

rosidl_adapter

• rename nested loop index (#643)
• Fix how comments in action interfaces are processed (#632)
• Pass comments in ros interface constants to the .idl generated files (#630)
• Update package maintainers (#624)
• Make rosidl packages use FindPython3 instead of FindPythonInterp (#612)
• Fix escaping in string literals (#595)
• Ignore multiple # characters and dedent comments (#594)
• Contributors: Ivan Santiago Paunovic, Michel Hidalgo, Shane Loretz, ibnHatab

rosidl_cli

• Fix importlib_metadata warning with Python 3.10. (#674)
• Update maintainers to Michel Hidalgo and Shane Loretz (#633)
• Update package maintainers (#624)
• Support passing keyword arguments to rosidl CLI extensions (#597)
• Add missing f for format string (#600)
• Contributors: Audrow Nash, Chris Lalancette, Michel Hidalgo, Shane Loretz
rosidl_cmake

• Make rosidl_get_typesupport_target return -NOTFOUND instead of FATAL_ERROR (#672)
• Add introspection typesupport tests for C/C++ messages (#651)
• Use target output name for exporting typesupport library (#625)
• Update package maintainers (#624)
• Revert “Bundle and ensure the exportation of rosidl generated targets” (#611)
• Add rosidl_get_typesupport_target and deprecate rosidl_target_interfaces (#606)
• Bundle and ensure the exportation of rosidl generated targets (#601)
• Contributors: Jonathan Selling, Michel Hidalgo, Shane Loretz

rosidl_default_generators

• Unroll group dependencies (#20)
• Contributors: Shane Loretz

rosidl_default_runtime

• Unroll group dependencies (#20)
• Contributors: Shane Loretz

rosidl_generator_c

• Fix error handling when copying C sequence messages (#671)
• Install generated headers to include/$PROJECT_NAME] (#670)
• Misc cleanup in the rosidl generator extensions (#662)
• Set the output size unconditionally when copying sequences (#669)
• Implement copy function for C messages (#650)
• Implement equality operator function for C messages. (#648)
• Generate documentation in generated C header files based on ROS interfaces comments (#593)
• Update package maintainers (#624)
• Make rosidl packages use FindPython3 instead of FindPythonInterp (#612)
• Revert “Bundle and ensure the exportation of rosidl generated targets” (#611)
• Bundle and ensure the exportation of rosidl generated targets (#601)
• Fix a cpplint allocator regression. (#590)
• Use RCUtils allocators in rosidl_generator_c (#584)
• Contributors: Chris Lalancette, Ivan Santiago Paunovic, Michel Hidalgo, Nikolai Morin, Pablo Garrido, Shane Loretz
rosidl_generator_cpp

- Install generated headers to include/${PROJECT_NAME} (#670)
- Misc cleanup in the rosidl generator extensions (#662)
- Add missing build_export_depend dependency (#665)
- Fix bug where rosidl_runtime_cpp wasn’t depended upon (#660)
- Fix include order for cpplint (#644)
- Set CXX standard to 17 (#635)
- Update package maintainers (#624)
- Make rosidl packages use FindPython3 instead of FindPythonInterp (#612)
- Support flow style YAML printing (#613)
- Revert “Bundle and ensure the exportation of rosidl generated targets” (#611)
- Relocate to_yaml() under message namespace (#609)
- Bundle and ensure the exportation of rosidl generated targets (#601)
- Contributors: Jacob Perron, Jorge Perez, Michel Hidalgo, Shane Loretz, Øystein Sture

rosidl_generator_dds_idl

- Add changelog (#56)
- Contributors: Ivan Santiago Paunovic

rosidl_generator_py

- Removes erroneous unmatched closing parenthesis (#125)
- require Python 3.6 as we use format strings in various places (#152)
- Fix rosidl_generator_py assuming incorrect library names (#149)
- Fix for msg file containing a property field that is not at the end (#151)
- Update package maintainers (#147)
- Use rosidl_get_typesupport_target() (#139)
- Support available typesupport specification in CLI extension (#133)
- Use python_d for test_cli_extension in Debug mode (#136)
- Add missing float<double bounds check (#128)
- Added optimization for copying arrays using buffer protocol (#129)
- Add smoke test for CLI extension (#132)
- Install generated Python interfaces in a Python package (#131)
- Contributors: Charles Cross, Chen Lihui, Michel Hidalgo, Seulbae Kim, Shane Loretz, William Woodall, ksuszka
Vulcanexus Documentation, Release 1.0.0

rosidl_parser

- Set maybe_placeholders to False for lark 1.+ compatibility (#664)
- Generate documentation in generated C header files based on ROS interfaces comments (#593)
- Pass comments in ros interface constants to the .idl generated files (#630)
- Update package maintainers (#624)
- Fix escaping in string literals (#595)
- Contributors: Ivan Santiago Paunovic, Michel Hidalgo, Shane Loretz

rosidl_runtime_c

- Fix error handling when copying C sequence messages (#671)
- Set the output size unconditionally when copying sequences (#669)
- De-duplicate Quality Level from README and QUALITY_DECLARATION (#661)
- Install headers to include/${PROJECT_NAME} (#658)
- Implement copy function for C messages (#650)
- Implement equality operator function for C messages. (#648)
- Set CXX standard to 17 (#635)
- Update package maintainers (#624)
- Use RCUtils allocators in rosidl_generator_c (#584)
- Contributors: Jose Luis Rivero, Michel Hidalgo, Nikolai Morin, Pablo Garrido, Shane Loretz, Øystein Sture

rosidl_runtime_cpp

- Add missing dependency on rosidl_runtime_c (#666)
- De-duplicate Quality Level from README and QUALITY_DECLARATION (#661)
- Install headers to include/${PROJECT_NAME} (#658)
- Set CXX standard to 17 (#635)
- Update package maintainers (#624)
- Contributors: Jose Luis Rivero, Michel Hidalgo, Shane Loretz, Øystein Sture

rosidl_runtime_py

- add yaml dump flow style. (#16)
- Update maintainers (#15) * Update maintainers to Shane Loretz * Update Shane’s email Co-authored-by: Shane Loretz <sloretz@openrobotics.org>
- Contributors: Audrow Nash, Tomoya Fujita

288 Chapter 4. Contributing to the documentation
**rosidl_typesupport_c**

- Use target_link_libraries(...) PRIVATE ... in single typesupport case (#124)
- rosidl CMake cleanup in rosidl_typesupport_c (#123)
- Install headers to include/${PROJECT_NAME} (#121)
- Use FindPython3 (#118)
- Revert “Bundle and ensure the exportation of rosidl generated targets” (#116)
- Support available typesupport specification in CLI extensions (#112)
- Bundle and ensure the exportation of rosidl generated targets (#113)
- Fix C and C++ typesupports CLI extensions (#111)
- Contributors: Michel Hidalgo, Shane Loretz

**rosidl_typesupport_cpp**

- Use target_link_libraries(...) PRIVATE ... in single typesupport case (#124)
- rosidl CMake cleanup in rosidl_typesupport_cpp (#123)
- Install headers to include/${PROJECT_NAME} (#121)
- Make sure to check typesupport handles against nullptr properly (#119)
- Use FindPython3 (#118)
- Revert “Bundle and ensure the exportation of rosidl generated targets” (#116)
- Support available typesupport specification in CLI extensions (#112)
- Bundle and ensure the exportation of rosidl generated targets (#113)
- Fix C and C++ typesupports CLI extensions (#111)
- Contributors: Chris Lalancette, Michel Hidalgo, Shane Loretz

**rosidl_typesupport_fastrtps_c**

- Install generated headers to include/${PROJECT_NAME} (#88)
- Misc fastrtps typesupport generator cleanup (#87)
- Install headers to include/${PROJECT_NAME} (#86)
- Fix include order for cpplint (#84)
- Update maintainers to Shane Loretz (#83)
- Use FindPython3 explicitly instead of PythonInterp implicitly (#78)
- Revert rosidl targets and dependencies exportation (#76) * Revert “Export rosidl_typesupport_fastrtps_c* dependencies (#75) * Revert “Bundle and ensure the exportation of rosidl generated targets (#73)”
- Correctly inform that a BoundedSequence is bounded (#71)
- Export rosidl_typesupport_fastrtps_c* dependencies (#75)
- Bundle and ensure the exportation of rosidl generated targets (#73)
• Fix Fast-RTPS C++ typesupport CLI extension (#72)
• Fastdds type support extensions (#67)
• Remove fastrtps dependency (#68)
• Contributors: Andrea Sorbini, Audrow Nash, Jacob Perron, Michel Hidalgo, Miguel Company, Shane Loretz

rosidl_typesupport_fastrtps_cpp

• Install generated headers to include/${PROJECT_NAME} (#88)
• Misc fastrtps typesupport generator cleanup (#87)
• Install headers to include/${PROJECT_NAME} (#86)
• Fix include order for cpplint (#84) * Fix include order for cpplint Relates to https://github.com/ament/ament_lint/pull/324 * Use double-quotes for other includes This is backwards compatible with older versions of cpplint.
• Update maintainers to Shane Loretz (#83)
• Re-introduce improvements to serialization of primitive bounded sequences for C++ type support (#81)
• Revert “Improve serialization of … (#79)” (#80)
• Improve serialization of primitive bounded sequences in C++ type support (#79)
• Use FindPython3 explicitly instead of PythonInterp implicitly (#78)
• Revert rosidl targets and dependencies exportation (#76) * Revert “Export rosidl_typesupport_fastrtps_c* dependencies (#75)” * Revert “Bundle and ensure the exportation of rosidl generated targets (#73)”
• Correctly inform that a BoundedSequence is bounded (#71)
• Export rosidl_typesupport_fastrtps_c* dependencies (#75)
• Bundle and ensure the exportation of rosidl generated targets (#73)
• Fix Fast-RTPS C++ typesupport CLI extension (#72)
• Fastdds type support extensions (#67)
• Remove fastrtps dependency (#68)
• Contributors: Andrea Sorbini, Audrow Nash, Jacob Perron, Jorge Perez, Michel Hidalgo, Miguel Company, Shane Loretz

rosidl_typesupport_interface

• De-duplicate Quality Level from README and QUALITY_DECLARATION (#661)
• Install headers to include/${PROJECT_NAME} (#658)
• Add ROSIDLTYPESUPPORTINTERFACE__LIBRARY_NAME() macro (#649)
• Set CXX standard to 17 (#635)
• Update package maintainers (#624)
• Contributors: Jose Luis Rivero, Michel Hidalgo, Shane Loretz, Øystein Sture
rosidl_typesupport_introspection_c

- Install generated headers to include/\{PROJECT_NAME\} (#670)
- Misc cleanup in the rosidl generator extensions (#662)
- De-duplicate Quality Level from README and QUALITY_DECLARATION (#661)
- Update Quality declaration to level 1 in README for instrospection pkgs (#659)
- Install headers to include/\{PROJECT_NAME\} (#658)
- Move rosidl_typesupport_introspection_c quality declaration to Q1 (#657)
- Move rosidl_typesupport_introspection_c quality declaration to Q1 (#656)
- Add documentation for generators and API (#646)
- Rework nested types’ items introspection in C and C++ (#652)
- Fix up the documentation for rosidl_typesupport_introspection_c (#628)
- Update package maintainers (#624)
- Quality Declaration for typesupport_introspection (#621)
- Make rosidl packages use FindPython3 instead of FindPythonInterp (#612)
- Revert “Bundle and ensure the exportation of rosidl generated targets” (#611)
- Bundle and ensure the exportation of rosidl generated targets (#601)
- Update function prefix (#596)
- Contributors: Chris Lalancette, Jose Luis Rivero, Michel Hidalgo, Pablo Garrido, Shane Loretz, eboasson

rosidl_typesupport_introspection_cpp

- Install generated headers to include/\{PROJECT_NAME\} (#670)
- Misc cleanup in the rosidl generator extensions (#662)
- De-duplicate Quality Level from README and QUALITY_DECLARATION (#661)
- Update Quality declaration to level 1 in README for instrospection pkgs (#659)
- Install headers to include/\{PROJECT_NAME\} (#658)
- Move rosidl_typesupport_introspection_cpp quality declaration to Q1 (#657)
- Add documentation for generators and API (#646)
- Rework nested types’ items introspection in C and C++ (#652)
- Set CXX standard to 17 (#635)
- Fix up the documentation for rosidl_typesupport_introspection_cpp (#627)
- Update package maintainers (#624)
- Quality Declaration for typesupport_introspection (#621)
- Make rosidl packages use FindPython3 instead of FindPythonInterp (#612)
- Revert “Bundle and ensure the exportation of rosidl generated targets” (#611)
- Bundle and ensure the exportation of rosidl generated targets (#601)
• Contributors: Chris Lalancette, Jose Luis Rivero, Michel Hidalgo, Shane Loretz, eboasson, Øystein Sture

rosidl_typesupport_introspection_tests

• Bump rosidl_typesupport_introspection_tests coverage (#655)
• Add introspection typesupport tests for C/C++ services (#653)
• Add introspection typesupport tests for C/C++ messages (#651)
• Contributors: Michel Hidalgo

rpyutils

• Make sure to call abspath when adding Windows DLL directories. (#8)
• Update troubleshooting links to docs.ros.org (#6)
• Contributors: Chris Lalancette, Christophe Bedard

rqt_gui

• Changed getiter to iter (#1) (#241)
• Update maintainers (#233) (#237)
• add missing dependencies: rospkg-modules, python_qt_binding, rospy (#227)
• bump CMake minimum version to avoid CMP0048 warning (#219)
• use catkin_install_python for Python script (#206)
• Contributors: Michael Jeronimo, sven-herrmann

rqt_gui_cpp

• Update maintainers (#233) (#237)
• bump CMake minimum version to avoid CMP0048 warning (#219)
• [Windows] fix rqt_gui_cpp install path (#190)
• [Windows] fix building (#189)
• Contributors: Michael Jeronimo

rqt_gui_py

• Update maintainers (#233) (#237)
• bump CMake minimum version to avoid CMP0048 warning (#219)
rqt_py_common

- Update maintainers (#233) (#237)
- bump CMake minimum version to avoid CMP0048 warning (#219)
- fix missing import bugs (#139)
- Contributors: Michael Jeronimo

rti_connext.dds_cmake_module

- Update rti-connext-dds dependency to 6.0.1. (#71)
- Contributors: Steven Ragnarök

rttest

- Install includes to include/${PROJECT_NAME} (#114)
- Fix include order for cpplint (#113)
- Fixes for uncrustify 0.72 (#111)
- Mark dependent targets as PRIVATE (#112)
- Export modern CMake targets (#110)
- Contributors: Chris Lalancette, Jacob Perron, Shane Loretz

rviz2

- Change links index.ros.org -> docs.ros.org. (#698)
- Contributors: Chris Lalancette

rviz_assimp_vendor

- Make sure to pass compiler and flags down to assimp (#844)
- Fix support for assimp 5.1.0 (#817)
- Contributors: Chris Lalancette, Silvio Traversaro

rviz_common

- Add implementation for cancel interface (#809)
- Install headers to include/${PROJECT_NAME} (#829)
- Remove definition of PLUGINLIB_DISABLE_BOOST. (#821)
- Fix support for assimp 5.1.0 (#817)
- Fix cpplint errors (#818)
- Set message type for ros topic display (#800)
• Fixes for uncrustify 0.72 (#807)
• Do not block visualization manager updates when opening the display panel dialog (#795)
• Switch to using Qt::MiddleButton for RViz. (#802)
• Removed traces in renderPanel (#777)
• move yaml_config_writer.hpp to public includes (#764)
• Update displays_panel.cpp (#745)
• Robot: Report mesh loading issues (#744)
• Exposed tool_manager header file. (#767)
• refactor: make const getter methods const (#756)
• Efficiently handle 3-bytes pixel formats (#743)
• Report sample lost events (#686)
• Update window close icon (#734)
• Fix missing “X” icon in panel close button (#731)
• Add rviz_rendering dependency to rviz_common (#727)
• Remove the word “Alpha” from the splash screen. (#696)
• Removed some memory leaks in rviz_rendering and rviz_rendering_tests (#710)
• Contributors: ANDOU Tetsuo, Alejandro Hernández Cordero, Chen Lihui, Chris Lalancette, Daisuke Nishimatsu, Gonzo, Ivan Santiago Paunovic, Jacob Perron, Joseph Schornak, Rebecca Butler, Shane Loretz, Silvio Traversaro, davidorchansky

rviz_default_plugins

• Add far plane distance property to camera (#849)
• Drop ignition-math6 from rviz_default_plugins link interface (#833)
• add implementation for cancel interface (#809)
• Install headers to include/${PROJECT_NAME} (#829)
• Remove definition of PLUGINLIB_DISABLE_BOOST. (#821)
• Remove TF filter from ImageTransportDisplay (#788)
• Add underscores to material names (#811)
• Export image_transport dependency (#813)
• Fixes for uncrustify 0.72 (#807)
• Switch to using Qt::MiddleButton for RViz. (#802)
• Add a tf_buffer_cache_time_ns to tf_wrapper (#792)
• Make libraries to avoid compiling files multiple times (#774)
• Computed inertia with ignition-math (#751)
• Fixed crash when changing rendering parameters for pointcloud2 while ‘Selectable’ box is unchecked (#768)
• Robot: Report mesh loading issues (#744)
• Handle NaN values for Wrench msgs (#746)
• Triangle lists support textures (#719)
• Report sample lost events (#686)
• Fix path message orientation error (#736)
• Set topic namespace in interactive markers display (#725)
• mass property visualization (#714)
• Export InteractiveMarker (#718)
• Yuv to rgb changes (#701)
• Extract message type in ImageTransportDisplay (#711)
• Duplicated code RobotJoint (#702)
• Don’t attempt to moc generate files that don’t have QOBJECT. (#690)
• Switch to including tf2_geometry_msgs.hpp (#689)
• Export Qt5 dependencies properly (#687)
• Add support for namespace-scoped DELETEALL action in Marker displays (#685)
• Use image_transport to subscribe to image messages (#523)
• Contributors: Akash, Alejandro Hernández Cordero, Audrow Nash, Chen Lihui, Chris Lalancette, Cory Crean, Gonzoz, Greg Balke, Ivan Santiago Paunovic, Jacob Perron, Martin Idel, Michel Hidalgo, Paul, Rebecca Butler, Scott K Logan, Shane Loretz, bailaC, brian soe, cturcotte-qnx, ketatam

rviz_ogre_vendor

• Fix interface link libraries in ogre vendor (#761)
• Fix the build for Ubuntu Jammy arm64. (#828)
• Strip RPATH from installed Ogre binaries (#688)
• Contributors: Chris Lalancette, Laszlo Turanyi, Michel Hidalgo

rviz_rendering

• Make getVerticesPerPoint method public and improve tests (#843)
• Disable class-memaccess warnings for Eigen (#838)
• Disable a warning when including Eigen. (#835)
• Install headers to include/${PROJECT_NAME} (#829)
• Fix support for assimp 5.1.0 (#817)
• Fix cpplint errors (#818)
• Fixes for uncrustify 0.72 (#807)
• Suppress assimp warnings in rviz_rendering build (#775)
• Fix for ogre failing when material already exists (#729)
• Removed some memory leaks in rviz_rendering and rviz_rendering_tests (#710)
• Export Qt5 dependencies properly (#687)
• Putting glsl 1.50 resources back in RenderSystem (#668)
• Contributors: Alejandro Hernández Cordero, Chris Lalancette, Jacob Perron, Jorge Perez, Michel Hidalgo, Piotr Jaroszek, Scott K Logan, Shane Loretz, Silvio Traversaro, Wolf Vollprecht

rviz_rendering_tests

• Removed some memory leaks in rviz_rendering and rviz_rendering_tests (#710)
• Contributors: Alejandro Hernández Cordero

rviz_visual_testing_framework

• Install headers to include/${PROJECT_NAME} (#829)
• Fixes for uncrustify 0.72 (#807)
• Update includes after rcutils/get_env.h deprecation (#677)
• Contributors: Chris Lalancette, Christophe Bedard, Shane Loretz

sensor_msgs

• Move the find_package statements for BUILD_TESTING (#186)
• Feedback on conditional sensor_msgs_library target (#1) (#183)
• [Fix] Fix image_encodings.hpp’s URL in README (#184)
• [Fix] Fix fill_image.hpp’s URL in README (#182)
• Add sensor_msgs_library target and install headers to include/${PROJECT_NAME} (#178)
• Interface packages should fully <depend> on the interface packages that they depend on (#173)
• Add YUV420 and YUV444 to image encodings (#172)
• Cleanup mislabeled BSD license (#83)
• Update maintainers to Geoffrey Biggs and Tully Foote (#163)
• Fix rosdoc2 warnings in sensor_msgs. (#162)
• Add equidistant distortion model (#160)
• Use rosidl_get_typesupport_target() (#156)
• Update CompressedImage documentation: add ‘tiff’ as a supported format (#154)
• Contributors: Audrow Nash, Chris Lalancette, Grey, Hemal Shah, Homalozoa X, Ivan Santiago Paunovic, Martin Günther, Michael Jeronimo, Pablo Garrido, Shane Loretz, Tully Foote
sensor_msgs_py

- Add in a compatibility layer for older versions of numpy. (#185)
- Port pointcloud creation to numpy. (#175)
- Update maintainers to Geoffrey Biggs and Tully Foote (#163)
- Contributors: Audrow Nash, Chris Lalancette, Florian Vahl

shape_msgs

- Interface packages should fully <depend> on the interface packages that they depend on (#173)
- Add prism type to the SolidPrimitive.msg (#166) (#167)
- Update maintainers to Geoffrey Biggs and Tully Foote (#163)
- Contributors: Audrow Nash, Grey, M. Fatih Cırıt

shared_queues_vendor

- Bump version number to avoid conflict
- Update package maintainers (#899)
- Contributors: Chris Lalancette, Michel Hidalgo

sqlite3_vendor

- Bump version number to avoid conflict
- Update package maintainers (#899)
- Contributors: Chris Lalancette, Michel Hidalgo

sros2

- Increase the shutdown timeout for test_generate_policy_no_nodes. (#278)
- Contributors: Chris Lalancette

statistics_msgs

- Update maintainers to Chris Lalancette (#130)
- Contributors: Audrow Nash
std_msgs

- Interface packages should fully `<depend>` on the interface packages that they depend on (#173)
- Update maintainers to Geoffrey Biggs and Tully Foote (#163)
- Contributors: Audrow Nash, Grey

std_srvs

- Update maintainers to Geoffrey Biggs and Tully Foote (#163)
- Contributors: Audrow Nash

stereo_msgs

- Interface packages should fully `<depend>` on the interface packages that they depend on (#173)
- Update maintainers to Geoffrey Biggs and Tully Foote (#163)
- Contributors: Audrow Nash, Grey

test_cli

- Updated maintainers (#489)
- Add changelogs (#473)
- Merge pull request #356 from ros2/issue/321_enhance_parameter_api
- Contributors: Aditya Pande, Ivan Santiago Paunovic

test_cli_remapping

- Update python nodes SIGINT handling (#490)
- Updated maintainers (#489)
- Add changelogs (#473)
- Contributors: Aditya Pande, Ivan Santiago Paunovic

test_communication

- Split test_subscriber into multiple compilation units. (#500)
- Add test_msgs dependency (#497)
- Update python nodes SIGINT handling (#490)
- Updated maintainers (#489)
- Fix deprecated subscriber callback warnings (#483)
- Add tests for BoundedPlainSequences (#481)
- Use rosidl_get_typesupport_target() (#480)
• Use rcpputils/scope_exit.hpp instead of rclcpp/scope_exit.hpp (#479)
• Add changelogs (#473)
• Contributors: Abrar Rahman Protyasha, Aditya Pande, Chris Lalancette, Christophe Bedard, Ivan Santiago Paunovic, Shane Loretz

test_interface_files

• Revert “Update package.xml (#18)” (#19)
• Update package.xml (#18)
• Update maintainers to Audrow Nash (#17)
• Added BoundedPlainSequences messages (#14)
• Contributors: Audrow Nash, Chris Lalancette, Miguel Company, Nikolai Morin

test_launch_ros

• Increase test time tolerance (#305)
• Use correct namespace when evaluating parameter files for composable nodes (#303)
• Handle empty strings when evaluating parameters (#300)
• Add parameter substitution (#297)
• More Helpful Error Messages (#275)
• Update maintainers in setup.py (#287)
• Set parameters from file for composable nodes (#281)
• Update package maintainers (#284)
• Update node name matcher (#282)
• Support both parameter file configurations for composable nodes (#259)
• Shutdown context after test (#267)
• Handle substitutions in RosTimer (#264)
• Add SetParametersFromFile action (#260)
• Properly support ros_args attribute through launch frontends (#253)
• Add 'push_ros_namespace' alias to 'push-ros-namespace' (#250)
• Add ros_arguments option to Node action (#249)
• ROS Timer Action (#244)
• Support container in frontend (#235)
• Added normalize_remap_rule and types. (launch #173)
• Fixed setup.py versions (launch #155)
• Fixed a bug to ensure that shutdown event is handled correctly (launch #154)
• Made various fixes and added tests for remappings passed to Node actions (launch #137)
• Contributors: Aditya Pande, Audrow Nash, Christophe Bedard, David V. Lu!!, Jacob Perron, Jorge Perez, Kenji Miyake, Michel Hidalgo, Rebecca Butler

**test_launch_testing**

- Update maintainers to Aditya Pande and Michel Hidalgo (#559)
- Updated maintainers (#555)
- Contributors: Aditya Pande, Audrow Nash

**test_msgs**

- Install headers to include/${PROJECT_NAME} and Depend on rosidl_typesupport_* targets directly (#133)
- Update maintainers to Chris Lalancette (#130)
- Add test fixtures for BoundedPlainSequences (#125)
- Added BoundedPlainSequences to test_msgs (#123)
- Contributors: Audrow Nash, Miguel Company, Shane Loretz

**test_quality_of_service**

- Update maintainers to Aditya Pande and Shane Loretz (#491)
- Updated maintainers (#489)
- Fix deprecated subscriber callback warnings (#483)
- Add changelogs (#473)
- Contributors: Abrar Rahman Protyasha, Aditya Pande, Audrow Nash, Ivan Santiago Paunovic

**test_rclcpp**

- Fix include order for cpplint (#493)
- Fix test (#488)
- Updated maintainers (#489)
- Add tests for rclcpp sigterm handler (#485)
- Fix deprecated subscriber callback warnings (#483)
- Fix deprecation warnings and failures after client API update (#482)
- Use rosidl_get_typesupport_target() (#480)
- Use rcpputils/scope_exit.hpp instead of rclcpp/scope_exit.hpp (#479)
- Add test for deferred service callback signature (#478)
- Add changelogs (#473)
- Merge pull request #357 from ros2/ros2_658_leftovers
- Contributors: Abrar Rahman Protyasha, Aditya Pande, Christophe Bedard, Ivan Santiago Paunovic, Jacob Perron, Mauro Passerino, Shane Loretz
test_rmw_implementation

- add content-filtered-topic interfaces (#181)
- Fix linter issues (#200)
- Add client/service QoS getters. (#196)
- Added tests for bounded sequences serialization (#193)
- Add RMW_DURATION_INFINITE basic compliance test. (#194)
- Test SubscriptionOptions::ignore_local_publications. (#192)
- Add rmw_publisher_wait_for_all_acked. (#188)
- Wait for server in test_rmw_implementation service tests. (#191)
- Contributors: Barry Xu, Chen Lihui, Emerson Knapp, Jorge Perez, Jose Antonio Moral, Michel Hidalgo, Miguel Company, mauropasse

test_security

- Updated maintainers (#489)
- Fix deprecated subscriber callback warnings (#483)
- Add changelogs (#473)
- Simplify the test_secure_subscriber code. (#471)
- Update includes after rcutils/get_env.h deprecation (#472)
- Contributors: Abrar Rahman Protyasha, Aditya Pande, Chris Lalancette, Christophe Bedard, Ivan Santiago Paunovic

test_tf2

- Fix more instances of Eigen problems on RHEL. (#515)
- Install includes to include/${PROJECT_NAME} and use modern CMake (#493)
- Fix precision loss from using rclcpp::Time::seconds() (#511)
- More Intuitive CLI for Static Transform Publisher (#392)
- Conversion tests for toMsg() (#423)
- Deprecate tf2_geometry_msgs.h (#418)
- Deprecate tf2_kdl.h (#414)
- Deprecate tf2_bullet.h (#412)
- Contributors: Bjar Ne, Chris Lalancette, Hunter L. Allen, Kenji Brameld, Shane Loretz
test_tracetools

- Introduce constants for tracepoint names
- Move actual tests out of tracetools_test to new test_tracetools package
- Contributors: Christophe Bedard

test_tracetools_launch

- Add support for preloading pthread and dl instrumentation shared libs
- Remove profile_fast option and consider LD_PRELOADing both libs
- Fix multiple LdPreload actions not working and add test
- Deprecate ‘context_names’ param and replace with ‘context_fields’
- Move some tests from tracetools_launch to test_tracetools_launch
- Contributors: Christophe Bedard, Ingo Lütkebohle

tf2

- Install includes to include/${PROJECT_NAME} and use modern CMake (#493)
- forward declare fromMsg to avoid missing symbols in downstream libraries (#485)
- tf2: Enable common linter tests (#469)
- Move time functions into time.cpp.
- Change a for loop to a while loop.
- Switch to C++-style casts.
- Remove totally unused (and unreachable) code.
- Replace NULL with nullptr.
- Fix up some comments.
- Use std::make_shared where we can.
- Replace two comparisons with empty string to empty().
- Make sure to include-what-you-use.
- Remove unnecessary internal method.
- Remove long-deprecated walkToTopParent overload.
- Remove unnecessary test dependencies.
- Remove some references to the ROS 1 wiki.
- Add rosidl_runtime_cpp as build_depend and build_export_depend.
- Minor cleanups in CMakeLists.txt.
- Remove include directory that doesn’t exist.
- Remove completely unnecessary target_link_libraries.
- Remove unused speed_test from tf2.
• Suppress clang warnings about enumerator attributes. (#463)
• Change TF2Error names to be a bit more descriptive. (#349)
• Fixed errors due to missing header link. (#432)
• Deprecate tf2_geometry_msgs.h (#418)
• Speedup covariance unwrapping (#399)
• Contributors: Abrar Rahman Protyasha, Chris Lalancette, Dima Dorezyuk, João C. Monteiro, Shane Loretz, Shivam Pandey

**tf2_bullet**

• Install includes to include/${PROJECT_NAME} and use modern CMake (#493)
• Export a tf2_bullet::tf2_bullet target (#495)
• Fix cpplint errors (#497)
• Remove some references to the ROS 1 wiki.
• Fix tf2_bullet dependency export (#428)
• Deprecate tf2_bullet.h (#412)
• Contributors: Bjar Ne, Chris Lalancette, Jacob Perron, Shane Loretz

**tf2_eigen**

• Workaround broken RHEL FindEigen3.cmake (#513)
• Install includes to include/${PROJECT_NAME} and use modern CMake (#493)
• Disable mem-access warnings on aarch64. (#506)
• Fix cpplint errors (#497)
• Remove some references to the ROS 1 wiki.
• Add doTransform function for twists or wrenches (#406)
• Reenable stamped eigen tests (#429)
• Deprecate tf2_eigen.h (#413)
• Contributors: AndyZe, Bjar Ne, Chris Lalancette, Jacob Perron, Shane Loretz

**tf2_eigen_kdl**

• Fix more instances of Eigen problems on RHEL. (#515)
• Depend on orocos_kdl_vendor (#473)
• Install includes to include/${PROJECT_NAME} and use modern CMake (#493)
• Fix cpplint errors (#497)
• Contributors: Chris Lalancette, Jacob Perron, Shane Loretz
tf2_geometry_msgs

- Make sure to find the right Python executable. (#514)
- Depend on orocos_kdl_vendor (#473)
- Install includes to include/${PROJECT_NAME} and use modern CMake (#493)
- Drop PyKDL dependency in tf2_geometry_msgs (#509)
- Fix cpplint errors (#497)
- Export a tf2_geometry_msgs::tf2_geometry_msgs target (#496)
- Feature: Add doTransform for Wrench messages (#476)
- Remove some references to the ROS 1 wiki.
- Style fixes in tf2_geometry_msgs. (#464)
- Fix for issue #431 - Covariance is not transformed in do_transform_pose_with_covarianceStamped (#453)
- doTransform non stamped msgs (#452)
- tf2_geometry_msgs: Fixing covariance transformation in doTransform<PoseWithCovarianceStamped, TransformStamped> (#430)
- Geometry nitpicks (#426)
- Conversion tests for toMsg() (#423)
- Deprecate tf2_geometry_msgs.h (#418)
- Contributors: Abrar Rahman Protyasha, Bjar Ne, Chris Lalancette, Denis Štogl, Florian Vahl, Jacob Perron, Khasreto, Shane Loretz, vineet131

tf2_kdl

- Depend on orocos_kdl_vendor (#473)
- Install includes to include/${PROJECT_NAME} and use modern CMake (#493)
- KDL python formatting and licenses (#425)
- Deprecate tf2_kdl.h (#414)
- Contributors: Bjar Ne, Chris Lalancette, Jacob Perron, Shane Loretz

tf2_msgs

- Remove dead file from tf2_msgs (#415)
- Contributors: Chris Lalancette
tf2_py

- Make sure to finalize tf2_py BufferCore. (#505)
- Make tf2_py Use FindPython3 (#494)
- Change TF2Error names to be a bit more descriptive. (#349)
- Remove python_compat.h (#417)
- Contributors: Chris Lalancette, Shane Loretz

tf2_ros

- Install includes to include/${PROJECT_NAME} and use modern CMake (#493)
- use dedicated callback group and executor to isolate timer (#447)
- Adding shared pointer definition to tf2 buffer (#508)
- fix for a basic logic (#510)
- Fix precision loss from using rclcpp::Time::seconds() (#511)
- clear relative callback of Buffer if MessageFilter is destroyed (#490)
- More info in tf2_echo output (#468)
- Fix cpplint errors (#497)
- Fixes for uncrustify 0.72 (#486)
- More Intuitive CLI for Static Transform Publisher (#392)
- Reduce transform listener nodes (#442)
- tf2_ros: Fix deprecated subscriber callbacks (#448)
- Fix tf2_echo does not work with ros-args (#407) (#408)
- Contributors: Abrar Rahman Protyasha, Chen Lihui, Chris Lalancette, Hunter L. Allen, Jacob Perron, Kenji Brameld, PGotzmann, Shane Loretz, Steve Macenski, Zhenpeng Ge, gezp, simulacrus

tf2_ros_py

- Drop PyKDL dependency in tf2_geometry_msgs (#509)
- Add in one more destroy call that was missed in testing. (#504)
- Be much more careful about cleanup in the tf2_ros_py tests. (#499)
- Use the correct type for BufferClient timeout_padding. (#498) It should be a duration, not a float.
- Update maintainers to Alejandro Hernandez Cordero and Chris Lalancette (#481)
- Fix buffer_client.py using default timeout_padding (#437)
- Use underscores instead of dashes in setup.cfg. (#403)
- Contributors: Audrow Nash, Carlos Andrés Álvarez Restrepo, Chris Lalancette, Florian Vahl
tf2_sensor_msgs

- Disable mem-access warnings on aarch64. (#506)
- Fix cpplint errors (#497)
- Reenable sensor_msgs test (#422)
- Deprecate tf2_sensor_msgs.h (#416)
- Contributors: Bjar Ne, Chris Lalancette, Jacob Perron

tf2_tools

- Update maintainers to Alejandro Hernandez Cordero and Chris Lalancette (#481)
- Remove unused import (#465)
- Adding date-time to frames filename (#454)
- Use underscores instead of dashes in setup.cfg. (#403)
- Contributors: Audrow Nash, Hannu Henttinen, Nisala Kalupahana

tlsf

- Install headers to include/${PROJECT_NAME} (#11)
- Export a modern CMake target instead of old-style variables (#10)
- Contributors: Shane Loretz

tlsf_cpp

- Install includes to include/${PROJECT_NAME} (#114)
- Export modern CMake targets (#110)
- Remove the use of malloc hooks from the tlsf_cpp tests. (#109)
- Contributors: Chris Lalancette, Shane Loretz

topic_monitor

- Update maintainers to Audrow Nash and Michael Jeronimo (#543)
- Small cleanups to the topic monitor. (#517)
- Fix topic_monitor for high publication rate (#461)
- Use is_alive for threads. (#510)
- Contributors: Audrow Nash, Chris Lalancette, Elias De Coninck
**topic_statistics_demo**

- Install includes to include/${PROJECT_NAME} (#548)
- Additional fixes for documentation in demos. (#538)
- Fixing deprecated subscriber callback warnings (#532)
- Contributors: Abrar Rahman Protyasha, Chris Lalancette, Shane Loretz

**tracetools**

- Install headers to include/${PROJECT_NAME}
- Merge branch ‘update-mentions-of-tracetools-test’ into ‘master’ Update applicable mentions of tracetools_test to test_tracetools See merge request ros-tracing/ros2_tracing!259
- Update applicable mentions of tracetools_test to test_tracetools
- Merge branch ‘version-3-1-0’ into ‘master’ Version 3.1.0 See merge request ros-tracing/ros2_tracing!256
- Correctly handle calls to TRACEPOINT() macro with no tracepoint args
- Move publisher handle tracepoint argument from rclcpp_publish to rcl_publish
- Add support for rmw init/pub, take, and executor instrumentation
- Export target on Windows and export an interface if TRACETOOLS_DISABLED
- Remove deprecated utility functions
- Contributors: Christophe Bedard, Ivan Santiago Paunovic, Shane Loretz

**tracetools_launch**

- Disable kernel tracing by default
- Don’t require kernel tracer and detect when it’s not installed
- Add support for preloading pthread and dl instrumentation shared libs
- Remove profile_fast option and consider LD_PRELOADing both libs
- Improve event matching for shared lib preloading
- Improve LdPreload action’s lib-finding function and add proper tests
- Fix multiple LdPreload actions not working and add test
- Deprecate ‘context_names’ param and replace with ‘context_fields’
- Support per-domain context fields for the Trace action
- Improve LdPreload.get_shared_lib_path() for when a static lib may exist
- Move some tests from tracetools_launch to test_tracetools_launch
- Expose Trace action as frontend action and support substitutions
- Contributors: Christophe Bedard, Ingo Lütkebohle
### tracertools_test

- Allow providing additional actions for TraceTestCase
- Remove default value for ‘package’ kwarg for TraceTestCase
- Move actual tests out of tracertools_test to new test_tracertools package
- Add tests for rmw init/pub, take, and executor instrumentation
- Add field type assertion utilities to TraceTestCase
- Fixing deprecated subscriber callback warnings
- Contributors: Abrar Rahman Protyasha, Christophe Bedard

### tracertools_trace

- Disable kernel tracing by default
- Don’t require kernel tracer and detect when it’s not installed
- Introduce constants for tracepoint names
- Optimize default tracing session channel config values
- Deprecate ‘context_names’ param and replace with ‘context_fields’
- Support per-domain context fields for the Trace action
- Add support for rmw init/pub, take, and executor tracepoints
- Contributors: Christophe Bedard

### trajectory_msgs

- Interface packages should fully <depend> on the interface packages that they depend on (#173)
- Update maintainers to Geoffrey Biggs and Tully Foote (#163)
- Contributors: Audrow Nash, Grey

### turtlesim

- Use double when handling qreal orient\_ (#114)
- Add Rolling Icon (#133)
- Update maintainers to Audrow Nash and Michael Jeronimo (#137)
- Fixing deprecated subscriber callback warnings (#134)
- Use rosidl_get_typesupport_target() (#132)
- Print out the correct node name on startup. (#122)
- Contributors: Abrar Rahman Protyasha, Audrow Nash, Chris Lalancette, Katherine Scott, Seulbae Kim, Shane Loretz
Vulcanexus Documentation, Release 1.0.0

urdf

- Install headers to include/${PROJECT_NAME} (#31)
- Add linter tests and fix errors (#30)
- Add in a Doxyfile to predefine macros. (#28)
- Contributors: Chris Lalancette, Jacob Perron, Shane Loretz

urdf_parser_plugin

- Install headers to include/${PROJECT_NAME} (#31)
- Add linter tests and fix errors (#30)
- Contributors: Jacob Perron, Shane Loretz

visualization_msgs

- Interface packages should fully <depend> on the interface packages that they depend on (#173)
- Update maintainers to Geoffrey Biggs and Tully Foote (#163)
- Marker Textures (#153)
- Document namespace scoped marker deletion. (#151)
- Contributors: Audrow Nash, Greg Balke, Grey, Michel Hidalgo

yaml_cpp_vendor

- Add missing dependency on yaml-cpp (#32)
- Upgrade to yaml-cpp 0.7.0 (#25)
- Contributors: Chris Lalancette, Scott K Logan

zstd_vendor

- Bump version number to avoid conflict
- Use git hash for zstd vendor (#969)
- Update package maintainers (#899)
- Declare missing dependency on ‘git’ in zstd_vendor (#890)
- Switch to using ‘git apply’ for zstd_vendor patches (#846)
- Contributors: Chris Lalancette, Christophe Bedard, Michel Hidalgo, Scott K Logan, Shane Loretz

Table of Contents

- Supported Platforms
- Installation
• Changes in Patch Release 1 (2022-11-23)
  – ros2topic

• New features in this ROS 2 release
  – ament_cmake_gen_version_h
  – launch
  – launch_ros
  – SROS2 Security enclaves support Certificate Revocation Lists
  – Content Filtered Topics
  – ros2cli
  – rviz2

• Changes since the Galactic release
  – C++ headers are installed in a subdirectory
  – common_interfaces
  – rmw
  – rmw_connextdds
  – rcl
  – rclcpp
  – rclcpp_lifecycle
  – rclpy
  – ros1_bridge
  – ros2cli
  – robot_state_publisher
  – rosidl_cmake
  – rviz2
  – geometry2
  – rosbag2

• Known Issues

• Release Timeline

Humble Hawksbill is the eighth release of ROS 2. What follows is highlights of the important changes and features in Humble Hawksbill since the last release. For a list of all of the changes since Galactic, see the long form changelog <Humble-Hawksbill-Complete-Changelog>. 
Supported Platforms

Humble Hawksbill is primarily supported on the following platforms:

Tier 1 platforms:

- Ubuntu 22.04 (Jammy): amd64 and arm64
- Windows 10 (Visual Studio 2019): amd64

Tier 2 platforms:

- RHEL 8: amd64

Tier 3 platforms:

- Ubuntu 20.04 (Focal): amd64
- macOS: amd64
- Debian Bullseye: amd64

For more information about RMW implementations, compiler / interpreter versions, and system dependency versions see REP 2000.

Installation

Install Humble Hawksbill

Changes in Patch Release 1 (2022-11-23)

**ros2topic**

**now** as keyword for **builtin_interfaces.msg.Time** and **auto** for **std_msgs.msg.Header**

**ros2 topic** pub now allows to set a **builtin_interfaces.msg.Time** message to the current time via the **now** keyword. Similarly, a **std_msgs.msg.Header** message will be automatically generated when passed the keyword **auto**. This behavior matches that of ROS 1’s **rostopic** (http://wiki.ros.org/ROS/YAMLCommandLine#Headers.2FTimestamps)

Related PR: ros2/ros2cli#751

New features in this ROS 2 release

**ament_cmake_gen_version_h**

Generating a C/C++ header with version info

A new CMake function to generate a header with the package version info was added to the **ament_cmake_gen_version_h** in **ament/ament_cmake#377**. Here’s the simplest use case:

```
project(my_project)
add_library(my_lib ...)
ament_generate_version_header(my_lib)
```
It will generate a header with version info from the package.xml and make it available to targets that link against the my_lib library.

How to include the header:

```
#include <my_project/version.h>
```

Where the header is installed to:

```
set(VERSION_HEADER ${CMAKE_INSTALL_PREFIX}/include/my_project/my_project/version.h)
```

### launch

**Scoping environment variables in group actions**

Similar to launch configurations, now by default, the state of environment variables are scoped to group actions.

For example, in the following launch files the executed processes will echo the value 1 (before Humble it would echo 2):

**XML**

```
<launch>
    <set_env name="FOO" value="1" />
    <group>
        <set_env name="FOO" value="2" />
    </group>
    <executable cmd="echo $FOO" output="screen" shell="true" />
</launch>
```

**Python**

```python
import launch
import launch.actions

def generate_launch_description():
    return launch.LaunchDescription([
        launch.actions.SetEnvironmentVariable(name='FOO', value='1'),
        launch.actions.GroupAction([
            launch.actions.SetEnvironmentVariable(name='FOO', value='2'),
        ]),
        launch.actions.ExecuteProcess(cmd=['echo', '${FOO}'], output='screen', shell=True),
    ])
```

If you would like disable scoping for launch configurations and environment variables you can set the scoped argument (or attribute) to false.

Related PR: ros2/launch#601
launch_pytest

We’ve added a new package, launch_pytest, that acts as an alternative to launch_testing. launch_pytest is a simple pytest plugin that provides pytest fixtures to manage the lifetime of a launch service.

Check out the package README for details and examples.
Related PR: ros2/launch#528

Allow matching target actions with a callable

Event handlers that take a target action object to match can now also take a callable instead to do the matching.
Related PR: ros2/launch#540

Access to math module when evaluating Python expressions

Inside PythonExpression substitutions (eval) we can now use symbols from Python’s math module. For example,

```xml
<launch>
  <log message="$(eval \ 'ceil(pi)')" />
</launch>
```

Related PR: ros2/launch#557

Boolean substitutions

New substitutions NotSubstitution, AndSubstitution, and OrSubstitution provide a convenient way to perform logical operations, for example

```xml
<launch>
  <let name="p" value="true" />
  <let name="q" value="false" />
  <group if="$(or $(var p) $(var q))">
    <log message="The first condition is true" />
  </group>
  <group unless="$(and $(var p) $(var q))">
    <log message="The second condition is false" />
  </group>
  <group if="$(not $(var q))">
    <log message="The third condition is true" />
  </group>
</launch>
```

Related PR: ros2/launch#598
New actions

- AppendEnvironmentVariable appends a value to an existing environment variable.
  - Related PR: ros2/launch#543
- ResetLaunchConfigurations resets any configuration applied to the launch configuration.
  - Related PR: ros2/launch#515

launch_ros

Passing ROS arguments to node actions

It is now possible to provide *ROS-specific node arguments* `<../../How-To-Guides/Node-arguments>` directly, without needing to use `args` with a leading `--ros-args` flag:

XML

```xml
<launch>
  <node pkg="demo_nodes_cpp" exec="talker" ros_args="--log-level debug" />
</launch>
```

YAML

```yaml
launch:
  - node:
      pkg: demo_nodes_cpp
      exec: talker
      ros_args: '--log-level debug'
```

The corresponding parameter for the Node action in Python launch files is `ros_arguments`:

```python
from launch import LaunchDescription
import launch_ros.actions

def generate_launch_description():
    return LaunchDescription([
        launch_ros.actions.Node(
            package='demo_nodes_cpp',
            executable='talker',
            ros_arguments=['--log-level', 'debug'],
        ),
    ])
```

Related PRs: ros2/launch_ros#249 and ros2/launch_ros#253.
**Frontend support for composable nodes**

We can now start node containers and load components into them from frontend launch files, for example:

```xml
<launch>
  <node_container pkg="rclcpp_components" exec="component_container" name="my_container" namespace="">
    <composable_node pkg="composition" plugin="composition::Talker" name="talker" />
  </node_container>
  <load_composable_node target="my_container">
    <composable_node pkg="composition" plugin="composition::Listener" name="listener" />
  </load_composable_node>
</launch>
```

```yaml
launch:
  - node_container:
    pkg: rclcpp_components
    exec: component_container
    name: my_container
    namespace: ''
    composable_node:
      - pkg: composition
        plugin: composition::Talker
        name: talker
    load_composable_node:
      target: my_container
      composable_node:
        - pkg: composition
          plugin: composition::Listener
          name: listener
```

Related PR: ros2/launch_ros#235

**Parameter substitution**

The new `ParameterSubstitution` lets you substitute the value of a parameter set previously in launch with the `SetParameter` action. For example,

```xml
<launch>
  <set_parameter name="foo" value="bar" />
  <log message="Parameter foo has value $(param foo)" />
</launch>
```

Related PR: ros2/launch_ros#297
New actions

- **RosTimer** acts like the launch **TimerAction**, but uses a ROS clock (so it can use simulation time, for example).
  - Related PRs: ros2/launch_ros#244 and ros2/launch_ros#264
- **SetParametersFromFile** passes a ROS parameters file to all nodes in a launch file (including node components).
  - Related PRs: ros2/launch_ros#260 and ros2/launch_ros#281

**SROS2 Security enclaves support Certificate Revocation Lists**

Certificate Revocation Lists (CRLs) are a concept where particular certificates can be revoked before their expiration. As of Humble, it is now possible to put a CRL in an SROS2 security enclave and have it be honored. See the SROS2 tutorials for an example of how to use it.

**Content Filtered Topics**

Content Filtered Topics supports a more sophisticated subscription that indicates the subscriber does not want to necessarily see all values of each instance published under the Topic. Content Filtered Topics can be used to request content-based subscriptions when underlying RMW implementation supports this feature.

### Table 1: RMW Content Filtered Topics support

<table>
<thead>
<tr>
<th>RMW</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>rmw_fastrtps</td>
<td>supported</td>
</tr>
<tr>
<td>rmw_connextdds</td>
<td>supported</td>
</tr>
<tr>
<td>rmw_cyclonedds</td>
<td>not supported</td>
</tr>
</tbody>
</table>

To learn more, see the content_filtering examples.

Related design PR: ros2/design#282.

**ros2cli**

**ros2 launch has a --launch-prefix argument**

This allows passing a prefix to all executables in a launch file, which is useful in many debugging situations. See the associated pull request, as well as the tutorial for more information.

Relatively, the --launch-prefix-filter command-line option was added to selectively add the prefix from --launch-prefix to executables. See the pull request for more information.
**ros2 topic echo has a --flow-style argument**

This allows the user to force flow style for the YAML representation of data on a topic. Without this option, the output from `ros2 topic echo /tf_static` could look something like:

```yaml
transforms:
  - header:
      stamp:
        sec: 1651172841
        nanosec: 433705575
      frame_id: single_rrbot_link3
      child_frame_id: single_rrbot_camera_link
    transform:
      translation:
        x: 0.05
        y: 0.0
        z: 0.9
      rotation:
        x: 0.0
        y: 0.0
        z: 0.0
        w: 1.0
```

With this option, the output would look something like:

```yaml
transforms: [{header: {stamp: {sec: 1651172841, nanosec: 433705575}, frame_id: single_rrbot_link3}, child_frame_id: single_rrbot_camera_link, transform: {translation: {x: 0.05, y: 0.0, z: 0.9}, rotation: {x: 0.0, y: 0.0, z: 0.0, w: 1.0}}}
```

See the PyYAML documentation for more information.

**ros2 topic echo can filter data based on message contents**

This allows the user to only print out data on a topic that matches a certain Python expression. For instance, using the following argument will only print out string messages that start with ‘foo’:

```bash
ros2 topic echo --filter 'm.data.startswith("foo")' /chatter
```

See the pull request for more information.

**rviz2**

**Apply textures to arbitrary triangle lists**

We’ve added the ability to apply textures defined via URI to arbitrary triangle lists using UV Coordinates. Now we can create a gradient pull from a texture map instead of the default grayscale. This will enable complex coloring of markers. To use this, you should use the `visualization_msgs/Marker.msg` and fill the `texture_resource`, `texture`, `uv_coordinates` and `mesh_file` fields. You can find more information here.
Visualization of mass properties (including inertia)

We also added the ability to visualize inertias. To do this, you select enable ‘Inertia’ in the ‘Mass Properties’ under the robot model:
You can see an image of an inertia below.

Visualize YUV images in RViz

It is now possible to directly visualize YUV images inside of RViz, rather than having to convert to RGB first. See ros2/rviz#701 for details.
Allow rendering of objects > 100 meters

By default, RViz only renders objects that are within 100 meters of a camera. A new configuration property called “Far Plane Distance” in the rviz camera plugin allows that rendering distance to be configured.

See ros2/rviz#849 for more information.

Changes since the Galactic release

C++ headers are installed in a subdirectory

In ROS 2 releases prior to Humble, C++ headers for all packages were installed into a single include directory. For instance, in Galactic, the directory structure looks like this (reduced for brevity):

```
/opt/ros/galactic/include/
   ├── rcl
   │    └── node.h
   ├── rclcpp
   │    └── node.hpp
```

This structure can cause serious problems when trying to use overlays. That is, it is very possible to get the wrong set of header files due to include directory order. See https://colcon.readthedocs.io/en/released/user/overriding-packages.html for a detailed explanation of the problems.

To help combat this, in Humble (and in all ROS 2 releases going forward), the directory structure has changed:

```
/opt/ros/humble/include
   ├── rcl
   │    └── rcl
   │         └── node.h
   └── rclcpp
       └── rclcpp
           └── node.hpp
```

Note that downstream packages that use these headers do not have to change; using `#include <rclcpp/node.hpp>` works as it always did before. However, when using IDEs that are looking for include directories, it may be necessary to add the individual include directories to the search path.

See https://github.com/ros2/ros2/issues/1150 for more information, including the reasoning behind this change.
common_interfaces

Support Textures and Embedded Meshes for Marker Messages

These two additions will improve the ability to both visualize data in new ways with standard messages and, simultaneously, enable the ability to track this data in rosbag.

**Textures** bring the addition of three new fields to markers:

```python
# Texture resource is a special URI that can either reference a texture file in
# a format acceptable to (resource retriever)[https://index.ros.org/p/resource_retriever/
string texture_resource
# An image to be loaded into the rendering engine as the texture for this marker.
# This will be used iff texture_resource is set to embedded.
sensor_msgs/CompressedImage texture
# Location of each vertex within the texture; in the range: [0.0-1.0]
UVCoordinate[] uv_coordinates
```

RViz will fully support texture rendering through the embedded format.

To those familiar with `mesh_resource`, `resource_retriever` should be familiar. This will allow the programmer to choose where they want to load data from, either a local file or a networked file. In the interest of being able to record all data in a rosbag, the ability to embed the texture image is included.

**Meshes** were modified in a similar way to add the ability to embed a raw Mesh file for the purpose of recording and are modified in a similar way. The `Meshfile` message has two fields:

```python
# The filename is used for both debug purposes and to provide a file extension
# for whatever parser is used.
string filename

# This stores the raw text of the mesh file.
uint8[] data
```

The embedded `Meshfile` message is not yet supported in implementation.

Related PRs: ros2/common_interfaces#153 ros2/rviz#719
Added PRISM type to SolidPrimitive

The SolidPrimitive message had a new PRISM type added, along with the appropriate metadata. See ros2/common_interfaces#167 for more information.

rmw

struct type name suffix changed from _t to _s

To avoid type name duplication errors between struct type names and their typedef-ed aliases when generating code documentation, the suffix for all struct type names has been changed from _t to _s. Aliases with _t suffixes remain in place. Thus, this change is a breaking change only for code that uses full struct type specifiers i.e. struct type_name_t.

See ros2/rmw#313 for more details.

rmw_connextdds

Use Connext 6 by default

By default, Humble Hawksbill uses Connext 6.0.1 as the DDS implementation for rmw_connextdds. It is still possible to use Connext 5.3.1 with rmw_connextdds, but it must be rebuilt from source.

rcl

struct type name suffix changed from _t to _s

To avoid type name duplication errors between struct type names and their typedef-ed aliases when generating code documentation, the suffix for all struct type names has been changed from _t to _s. Aliases with _t suffixes remain in place. Thus, this change is a breaking change only for code that uses full struct type specifiers i.e. struct type_name_t.

See ros2/rcl#932 for more details.

ROS_DISABLE_LOANED_MESSAGES environment variable added

This environment variable can be used to disable loaned messages support, independently if the rmw supports them or not. For more details, see the guide Disabling Zero Copy Loaned Messages.

rclcpp

Support Type Adaption for Publishers and Subscriptions

After defining a type adapter, custom data structures can be used directly by publishers and subscribers, which helps to avoid additional work for the programmer and potential sources of errors. This is especially useful when working with complex data types, such as when converting OpenCV’s cv::Mat to ROS’s sensor_msgs/msg/Image type.

Here is an example of a type adapter that converts std_msgs::msg::String to std::string:
template<>
struct rclcpp::TypeAdapter<
    std::string,
    std_msgs::msg::String
>
{
    using is_specialized = std::true_type;
    using custom_type = std::string;
    using ros_message_type = std_msgs::msg::String;

    static void
    convert_to_ros_message(
        const custom_type & source,
        ros_message_type & destination)
    {
        destination.data = source;
    }

    static void
    convert_to_custom(
        const ros_message_type & source,
        custom_type & destination)
    {
        destination = source.data;
    }
};

And an example of how the type adapter can be used:

using MyAdaptedType = TypeAdapter<std::string, std_msgs::msg::String>;

// Publish a std::string
auto pub = node->create_publisher<MyAdaptedType>(...);
std::string custom_msg = "My std::string"
pub->publish(custom_msg);

// Pass a std::string to a subscription's callback
auto sub = node->create_subscription<MyAdaptedType>(
    "topic",
    10,
    [](const std::string & msg) {...});

To learn more, see the publisher and subscription examples, as well as a more complex demo. For more details, see REP 2007.
Client::async_send_request(request) returns a std::future instead of a std::shared_future

This change was implemented in rclcpp#1734. This breaks API, as std::future::get() methods extract the value from the future. That means, if that method is called for a second time it will throw an exception. That doesn’t happen with a std::shared_future, as its get() method returns a const &. Example:

```cpp
auto future = client->async_send_request(req);
...
do_something_with_response(future.get());
...
do_something_else_with_response(future.get());  // this will throw an exception now!!
```

should be updated to:

```cpp
auto future = client->async_send_request(req);
auto response = future.get();
do_something_with_response(response);
...
do_something_else_with_response(response);
```

If a shared future is needed, the std::future::share() method can be used.

wait_for_all_acked method added to Publisher

This new method will block until all messages in the publisher queue are acked by the matching subscriptions or the specified timeout expires. It is only useful for reliable publishers, as in the case of best effort QoS there’s no acking. Examples:

```cpp
auto pub = node->create_publisher<std_msgs::msg::String>...();
...
pub->publish(my_msg);
...
pub->wait_for_all_acked();  // or pub->wait_for_all_acked(timeout)
```

For a more complete example, see here.

get_callback_groups method removed from NodeBase and Node classes

for_each_callback_group() method has replaced get_callback_groups() by providing a thread-safe way to access callback_groups_.vector. for_each_callback_group() accepts a function as an argument, iterates over the stored callback groups, and calls the passed function to ones that are valid.

For more details, please refer to this pull request.
**add_to_wait_set method from Waitable class changes its return type from bool to void**

Before, classes derived from `Waitable` overriding `add_to_wait_set` were returning false when failing to add elements to the wait set, so the caller had to check this return value and throw or handle the error. This error handling should now be done directly on `add_to_wait_set` method, throwing if necessary. It is not required to return anything if no errors happened. Thus, this is a breaking change for downstream uses of `Waitable`.

See [ros2/rclcpp#1612](https://github.com/ros2/rclcpp/issues/1612) for more details.

**get_notify_guard_condition method return type from NodeBaseInterface class changed**

Now `rclcpp` uses the `GuardCondition` class wrapper around `rcl_guard_condition_t`, so `get_notify_guard_condition` returns a reference to the node's `rclcpp::GuardCondition`. Thus, this is a breaking change for downstream uses of `NodeBaseInterface` and `NodeBase`.

See [ros2/rclcpp#1612](https://github.com/ros2/rclcpp/issues/1612) for more details.

**sleep_until and sleep_for methods added to Clock**

Two new methods were added to allow sleeping on a particular clock in [ros2/rclcpp#1814](https://github.com/ros2/rclcpp/issues/1814) and [ros2/rclcpp#1828](https://github.com/ros2/rclcpp/issues/1828). `Clock::sleep_until` will suspend the current thread until the clock reaches a particular time. `Clock::sleep_for` will suspend the current thread until the clock advances a certain amount of time from when the method was called. Both methods will wake early if the `Context` is shutdown.

**rclcpp_lifecycle**

**Active and deactivate transitions of publishers will be triggered automatically**

Before, users needed to override `LifecycleNode::on_activate()` and `LifecycleNode::on_deactivate()` and call the similarly named methods on `LifecyclePublisher` to make the transition actually happen. Now, `LifecycleNode` provides a default interface of these methods that already do this. See the implementation of the `lifecycle_talker` node [here](https://github.com/ros2/lifecycle/wiki/lifecycle_talker).

**rclpy**

**Managed nodes**

Lifecycle nodes support was added to `rclpy`. A complete demo can be found [here](https://github.com/ros2/rclpy/tree/master/examples/lifecycle).

**wait_for_all_acked method added to Publisher**

Similar to the feature added to `rclcpp`. 
sleep_until and sleep_for methods added to Clock

Two new methods were added to allow sleeping on a particular clock in ros2/rclpy#858 and ros2/rclpy#864. sleep_until will suspend the current thread until the clock reaches a particular time. sleep_for will suspend the current thread until the clock advances a certain amount of time from when the method was called. Both methods will wake early if the Context is shutdown.

ros1_bridge

Since there is no official ROS 1 distribution on Ubuntu Jammy and forward, ros1_bridge is now compatible with the Ubuntu-packaged versions of ROS 1. More details about using ros1_bridge with Jammy packages are available in the how-to guides.

ros2cli

ros2 commands disable output buffering by default

Prior to this release, running a command like

```
ros2 echo /chatter | grep "Hello"
```

would not print any data until the output buffer was full. Users could work around this by setting PYTHONUNBUFFERED=1, but that was not very user friendly.

Instead, all ros2 commands now do line-buffering by default, so commands like the above work as soon as a newline is printed. To disable this behavior and use default python buffering rules, use the option --use-python-default-buffering. See the original issue and the pull request for more information.

ros2 topic pub will wait for one matching subscription when using --times/--once/-1

When using --times/--once/-1 flags, ros2 topic pub will wait for one matching subscription to be found before starting to publish. This avoids the issue of the ros2cli node starting to publish before discovering a matching subscription, which results in some of the first messages being lost. This is particularly unexpected when using a reliable qos profile.

The number of matching subscriptions to wait before starting publishing can be configured with the -w/--wait-matching-subscriptions flags, e.g.:

```
ros2 topic pub -1 -w 3 /chatter std_msgs/msg/String "{data: 'foo'}"
```

to wait for three matching subscriptions before starting to publish.

-w can also be used independently of --times/--once/-1 but it only defaults to one when combined with them, otherwise the -w default is zero.

See https://github.com/ros2/ ros2cli/pull/642 for more details.
**ros2 param dump default output changed**

- **--print** option for dump command was deprecated.
  
  It prints to stdout by default:

  ```
  ros2 param dump /my_node_name
  ```

- **--output-dir** option for dump command was deprecated.
  
  To dump parameters to a file, run:

  ```
  ros2 param dump /my_node_name > my_node_name.yaml
  ```

**ros2 param set now accepts more YAML syntax**

Previously, attempting to set a string like “off” to a parameter that was of string type did not work. That’s because ros2 param set interprets the command-line arguments as YAML, and YAML considers “off” to be a boolean type. As of https://github.com/ros2/ros2cli/pull/684, ros2 param set now accepts the YAML escape sequence of “!!str off” to ensure that the value is considered a string.

**ros2 pkg create can automatically generate a LICENSE file**

If the --license flag is passed to ros2 pkg create, and the license is one of the known licenses, ros2 pkg create will now automatically generate a LICENSE file in the root of the package. For a list of known licenses, run ros2 pkg create --license ? <package_name>. See the associated pull request for more information.

**robot_state_publisher**

**Added frame_prefix parameter**

A new parameter frame_prefix was added in ros/robot_state_publisher#159. This parameter is a string which is prepended to all frame names published by robot_state_publisher. Similar to tf_prefix in the original tf library in ROS 1, this parameter can be used to publish the same robot description multiple times with different frame names.

**Removal of deprecated use_tf_static parameter**

The deprecated use_tf_static parameter has been removed from robot_state_publisher. This means that static transforms are unconditionally published to the /tf_static topic, and that the static transforms are published in a transient_local Quality of Service. This was the default behavior, and the behavior which the tf2_ros::TransformListener class expected before, so most code will not have to be changed. Any code that was relying on robot_state_publisher to periodically publish static transforms to /tf will have to be updated to subscribe to /tf_static as a transient_local subscription instead.
rosidl_cmake

Deprecation of rosidl_target_interfaces()

The CMake function rosidl_target_interfaces() has been deprecated, and now issues a CMake warning when called. Users wanting to use messages/services/actions in the same ROS package that generated them should instead call rosidl_get_typesupport_target() and then target_link_libraries() to make their targets depend on the returned typesupport target. See https://github.com/ros2/rosidl/pull/606 for more details, and https://github.com/ros2/demos/pull/529 for an example of using the new function.

rviz2

- improved the efficiency of 3-bytes pixel formats
- changed the way inertias are computed to use ignition math rather than Ogre’s math libraries.

gometry2

Deprecation of TF2Error::NO_ERROR, etc

The tf2 library uses an enumeration called TF2Error to return errors. Unfortunately, one of the enumerators in there is called NO_ERROR, which conflicts with a macro on Windows. To remedy this, a new set of enumerators in TF2Error were created, each with a TF2 prefix. The previous enumerators are still available, but are now deprecated and will print a deprecation warning if used. All code that uses the TF2Error enumerator should be updated to use the new TF2 prefixed errors. See https://github.com/ros2/geometry2/pull/349 for more details.

More intuitive command-line arguments for static_transform_publisher

The static_transform_publisher program used to take arguments like: ros2 run tf2_ros static_transform_publisher 0 0 0 0 0 0 1 foo bar. The first three numbers are the translation x, y, and z, the next 4 are the quaternion x, y, z, and w, and the last two arguments are the parent and child frame IDs. While this worked, it had a couple of problems:

- The user had to specify all of the arguments, even if only setting one number
- Reading the command-line to figure out what it was publishing was tricky

To fix both of these issues, the command-line handling has been changed to use flags instead, and all flags except for --frame-id and --child-frame-id are optional. Thus, the above command-line can be simplified to: ros2 run tf2_ros static_transform_publisher --frame-id foo --child-frame-id bar. To change just the translation x, the command-line would be: ros2 run tf2_ros static_transform_publisher --x 1.5 --frame-id foo --child-frame-id bar.

The old-style arguments are still allowed in this release, but are deprecated and will print a warning. They will be removed in future releases. See https://github.com/ros2/geometry2/pull/392 for more details.
Transform listener spin thread no longer executes node callbacks

tf2_ros::TransformListener no longer spins on the provided node object. Instead, it creates a callback group to execute callbacks on the entities it creates internally. This means if you have set the parameter spin_thread=true when creating a transform listener, you can no longer depend on your own callbacks to be executed. You must call a spin function on your node (e.g. rclcpp::spin), or add your node to your own executor.

Related pull request: geometry2#442

rosbag2

New playback and recording controls

Several pull requests have been added to enhance the user’s control over playback of bags. Pull request 931 adds the ability to specify a time stamp to begin playing from. Due to pull request 789 it is now possible to delay the start of playback by a specified interval.

Relatedly, rosbag2 has gained new ways for users to control playback as it is happening. Pull request 847 adds keyboard controls for pausing, resuming, and playing the next message during playback from a terminal. It is also possible to start playback paused thanks to pull requests 905 and 904, which makes it easy for the user to initiate playback and then step through messages, such as when debugging a pipeline. Pull request 836 adds an interface for seeking within bags, allowing the user to move around within a bag during playback.

Finally, a new snapshot mode has been added to recording in pull request 851. This mode, useful for incident recording, allows recording to begin filling up buffers, but not begin writing data to disc until a service is called.

Burst-mode playback

While the playback of data from a bag in real-time is the most well-known use case for bag files, there are situations where you want the data in the bag as fast as possible. With pull request 977, rosbag2 has gained the ability to “burst” data from the bag. In burst mode, the data is played back as fast as possible. This is useful in applications such as machine learning.

Zero-Copy playback

By default, if loaned message can be used, playback messages are published as loaned message. This can help to reduce the number of data copies, so there is a greater benefit for sending big data. Pull request 981 adds --disable-loan-message option for playback.

Wait for an acknowledgment

This new option will wait until all published messages are acknowledged by all subscribers or until the timeout elapses in millisecond before play is terminated. Especially for the case of sending message with big size in a short time. This option is valid only if the publisher’s QOS profile is RELIABLE. Pull request 951 adds --wait-for-all-acked option for playback.
Bag editing

rosbag2 is taking steps towards enabling the editing of bags, such as removing all messages for one topic or merging multiple bags into a single bag. Pull request 921 adds bag rewriting and the `ros2 bag convert` verb.

Other changes

Pull request 925 makes `rosbag2` ignore “leaf topics” (topics without a publisher) when recording. These topics will no longer be automatically added to the bag.

Known Issues

- When installing ROS 2 on an Ubuntu 22.04 Jammy host it is important to update your system before installing ROS 2 packages. It is particularly important to make sure that systemd and udev are updated to the latest available version otherwise installing `ros-humble-desktop`, which depends on `libudev1`, could cause the removal of system critical packages. Details can be found in `ros2/ros2#1272` and Launchpad #1974196.

- When ROS 2 apt repositories are available, ROS 1 packages in Ubuntu are not installable. See the `ros1_bridge on Ubuntu Jammy` document for more information.

- Some major Linux distributions have started patching Python to install packages to `/usr/local`, which is breaking some parts of `ament_package` and builds with `colcon`. In particular, using Ubuntu Jammy with `setuptools` installed from pip will manifest this misbehavior, and is therefore not recommended. There is currently a proposed solution which requires further testing before widespread release.

- ROS 2 bags that are split by size or duration are not played correctly. Only the last bag recorded is played. It is recommended to avoid splitting bags by size or duration. Details can be found in `ros2/rosbag2#966`.

Release Timeline

**Mon. March 21, 2022 - Alpha + RMW freeze** Preliminary testing and stabilization of ROS Base¹ packages, and API and feature freeze for RMW provider packages.

**Mon. April 4, 2022 - Freeze** API and feature freeze for ROS Base¹ packages in Rolling Ridley. Only bug fix releases should be made after this point. New packages can be released independently.

**Mon. April 18, 2022 - Branch** Branch from Rolling Ridley. `rosdistro` is reopened for Rolling PRs for ROS Base¹ packages. Humble development shifts from `ros-rolling-*` packages to `ros-humble-*` packages.

**Mon. April 25, 2022 - Beta** Updated releases of ROS Desktop² packages available. Call for general testing.

**Mon. May 16, 2022 - Release Candidate** Release Candidate packages are built. Updated releases of ROS Desktop² packages available.

**Thu. May 19, 2022 - Distro Freeze** Freeze rosdistro. No PRs for Humble on the rosdistro repo will be merged (reopens after the release announcement).

**Mon. May 23, 2022 - General Availability** Release announcement. rosdistro is reopened for Humble PRs.

¹ The `ros_base` variant is described in REP 2001 (ros-base).
² The `desktop` variant is described in REP 2001 (desktop-variants).
Rolling Ridley (rolling)

Table of Contents

- Currently Supported Platforms
- Installation
- New features and changes in this release

Rolling Ridley is a rolling development release of ROS 2.

Warning: Rolling Ridley is continuously updated and is subject to in-place updates which will at times include breaking changes. It is used for ROS 2 development and by maintainers who want their packages released and ready for the next stable distribution. We recommend that most users of ROS 2 use the latest stable distribution <latest_release>.

For more information see REP-2002

Currently Supported Platforms

Rolling Ridley is currently supported on the following platforms:

Tier 1 platforms:
- Ubuntu 22.04 (Jammy): amd64 and arm64
- Windows 10 (Visual Studio 2019)

Tier 3 platforms:
- Debian Buster (10): amd64, arm64 and arm32
- Mac macOS 10.14 (Mojave)
- OpenEmbedded Thud (2.6) / webOS OSE: arm32 and x86

Installation

Install Rolling Ridley <../../Installation>

New features and changes in this release

Rolling Ridley is an ongoing development distribution. Changes between the current stable release and the upcoming one can be found on the page for the upcoming release <upcoming-release>.
Development Distribution

Below is the ROS 2 distribution that is currently in development.

**Jazzy Jalisco (codename ‘jazzy’; May, 2024)**

**Table of Contents**

- Supported Platforms
- Installation
- New features in this ROS 2 release
- Development progress

*Jazzy Jalisco* is the tenth release of ROS 2. What follows is highlights of the important changes and features in Jazzy Jalisco since the last release.

**Supported Platforms**

Jazzy Jalisco is primarily supported on the following platforms:

Tier 1 platforms:
- TODO

Tier 2 platforms:
- TODO

Tier 3 platforms:
- TODO

For more information about RMW implementations, compiler / interpreter versions, and system dependency versions see REP 2000.

**Installation**

TODO

**New features in this ROS 2 release**

**Development progress**

For progress on the development of Jazzy Jalisco, see this project board.

For the broad process followed by Jazzy Jalisco, see the process description page.
End-of-Life Distributions

Below is a list of historic ROS 2 distributions that are no longer supported.

Galactic Geochelone (galactic)

Galactic Geochelone changelog

This page is a list of the complete changes in all ROS 2 core packages since the previous release.

Table of Contents

- action_msgs
- action_tutorials_cpp
- action_tutorials_interfaces
- action_tutorials_py
- actionlib_msgs
- ament_clang_format
- ament_clang_tidy
- ament_cmake
- ament_cmake_auto
- ament_cmake_clang_format
- ament_cmake_clang_tidy
- ament_cmake_copyright
- ament_cmake_core
- ament_cmake_cppcheck
- ament_cmake_cpplint
- ament_cmake_export_definitions
- ament_cmake_export_dependencies
- ament_cmake_export_include_directories
- ament_cmake_export_interfaces
- ament_cmake_export_libraries
- ament_cmake_export_link_flags
- ament_cmake_export_targets
- ament_cmake_flake8
- ament_cmake_gmock
- ament_cmake_google_benchmark
- ament_cmake_gtest
Vulcanexus Documentation, Release 1.0.0

- ament_cmake_include_directories
- ament_cmake_libraries
- ament_cmake_lint_cmake
- ament_cmake_mypy
- ament_cmake_nose
- ament_cmake_pclint
- ament_cmake_pep257
- ament_cmake_pycodestyle
- ament_cmake_pyflakes
- ament_cmake_pytest
- ament_cmake_python
- ament_cmake_ros
- ament_cmake_target_dependencies
- ament_cmake_test
- ament_cmake_uncrustify
- ament_cmake_version
- ament_cmake_xmllint
- ament_copyright
- ament_cppcheck
- ament_cpplint
- ament_flake8
- ament_index_cpp
- ament_index_python
- ament_lint
- ament_lint_auto
- ament_lint_cmake
- ament_lint_common
- ament_mypy
- ament_package
- ament_pclint
- ament_pep257
- ament_pycodestyle
- ament_pyflakes
- ament_uncrustify
- ament_xmllint

Chapter 4. Contributing to the documentation
- builtin_interfaces
- camera_calibration_parsers
- camera_info_manager
- class_loader
- common_interfaces
- composition
- composition_interfaces
- cyclonedds
- demo_nodes_cpp
- demo_nodes_cpp_native
- demo_nodes_py
- diagnostic_msgs
- domain_coordinator
- dummy_map_server
- dummy_robot_bringup
- dummy_sensors
- example_interfaces
- examples_rclcpp_cbg_executor
- examples_rclcpp_minimal_action_client
- examples_rclcpp_minimal_action_server
- examples_rclcpp_minimal_client
- examples_rclcpp_minimal_composition
- examples_rclcpp_minimal_publisher
- examples_rclcpp_minimal_service
- examples_rclcpp_minimal_subscriber
- examples_rclcpp_minimal_timer
- examples_rclcpp_multithreaded_executor
- examples_rclpy_executors
- examples_rclpy_guard_conditions
- examples_rclpy_minimal_action_client
- examples_rclpy_minimal_action_server
- examples_rclpy_minimal_client
- examples_rclpy_minimal_publisher
- examples_rclpy_minimal_service
- examples_rclpy_minimal_subscriber
• examples_rclpy_pointcloud_publisher
• examples_tf2_py
• fastrtps_cmake_module
• geometry2
• geometry_msgs
• google_benchmark_vendor
• image_common
• image_tools
• image_transport
• interactive_markers
• intra_process_demo
• kdl_parser
• laser_geometry
• launch
• launch_ros
• launch_testing
• launch_testing_ament_cmake
• launch_testing_ros
• launch_xml
• launch_yaml
• libcurl_vendor
• libstatistics_collector
• libyaml_vendor
• lifecycle
• lifecycle_msgs
• logging_demo
• map_msgs
• message_filters
• mimick_vendor
• nav_msgs
• osrf_pycommon
• osrf_testing_tools_cpp
• pendulum_control
• pendulum_msgs
• performance_test_fixture
• pluginlib
• pybind11_vendor
• python_cmake_module
• python_qt_binding
• qt_dotgraph
• qt_gui
• qt_gui_cpp
• quality_of_service_demo_cpp
• quality_of_service_demo_py
• rcl
• rcl_action
• rcl_interfaces
• rcl_lifecycle
• rcl_logging_interface
• rcl_logging_log4cxx
• rcl_logging_noop
• rcl_logging_spdlog
• rcl_yaml_param_parser
• rclcpp
• rclcpp_action
• rclcpp_components
• rclcpp_lifecycle
• rclpy
• rcpputils
• rcutils
• resource_retriever
• rmw
• rmw_connextdds
• rmw_connextdds_common
• rmw_connextdsmicro
• rmw_cyclonedds_cpp
• rmw_dds_common
• rmw_fastrtps_cpp
• rmw_fastrtps_dynamic_cpp
• rmw_fastrtps_shared_cpp
• rmw_implementation
• rmw_implementation_cmake
• robot_state_publisher
• ros1_bridge
• ros2action
• ros2bag
• ros2cli
• ros2cli_common_extensions
• ros2cli_test_interfaces
• ros2component
• ros2doctor
• ros2interface
• ros2launch
• ros2lifecycle
• ros2lifecycle_test_fixtures
• ros2multicast
• ros2node
• ros2param
• ros2pkg
• ros2run
• ros2service
• ros2test
• ros2topic
• ros_testing
• rosbag2
• rosbag2_compression
• rosbag2_compression_zstd
• rosbag2_cpp
• rosbag2_interfaces
• rosbag2_performance_benchmarking
• rosbag2_py
• rosbag2_storage
• rosbag2_storage_default_plugins
• rosbag2_test_common
• rosbag2_tests
• `rosbag2_transport`
• `rosgraph_msgs`
• `rosidl_adapter`
• `rosidl_cli`
• `rosidl_cmake`
• `rosidl_default_generators`
• `rosidl_default_runtime`
• `rosidl_generator_c`
• `rosidl_generator_cpp`
• `rosidl_generator_dds_idl`
• `rosidl_generator_py`
• `rosidl_parser`
• `rosidl_runtime_c`
• `rosidl_runtime_cpp`
• `rosidl_runtime_py`
• `rosidl_typesupport_c`
• `rosidl_typesupport_cpp`
• `rosidl_typesupport_fastrtps_c`
• `rosidl_typesupport_fastrtps_cpp`
• `rosidl_typesupport_interface`
• `rosidl_typesupport_introspection_c`
• `rosidl_typesupport_introspection_cpp`
• `rpyutils`
• `rqt`
• `rqt_action`
• `rqt_bag`
• `rqt_bag_plugins`
• `rqt_console`
• `rqt_graph`
• `rqt_gui`
• `rqt_gui_cpp`
• `rqt_gui_py`
• `rqt_msg`
• `rqt_plot`
• `rqt_publisher`

4.8. ROS 2 Documentation
• rqt_py_common
• rqt_py_console
• rqt_reconfigure
• rqt_service_caller
• rqt_shell
• rqt_srv
• rqt_top
• rqt_topic
• rti_connext.dds_cmake_module
• rttest
• rviz2
• rviz_assimp_vendor
• rviz_common
• rviz_default_plugins
• rviz_ogre_vendor
• rviz_rendering
• rviz_rendering_tests
• rviz_visual_testing_framework
• sensor_msgs
• sensor_msgs_py
• shape_msgs
• shared_queues_vendor
• spdlog_vendor
• sqlite3_vendor
• sros2
• statistics_msgs
• std_msgs
• std_srvs
• stereo_msgs
• tango_icons_vendor
• test_cli
• test_cli_remapping
• test_communication
• test_interface_files
• test_launch_ros
- test_launch_testing
- test_msgs
- test_quality_of_service
- test_rclcpp
- test_rmw_implementation
- test_security
- test_t2f
- t2
- t2_bullet
- t2_eigen
- t2_eigen_kdl
- t2_geometry_msgs
- t2_kdl
- t2_msgs
- t2_py
- t2_ros
- t2_ros_py
- t2_sensor_msgs
- t2_tools
- tlsf
- tlsf_cpp
- topic_monitor
- topic_statistics_demo
- tracetools
- tracetools_launch
- tracetools_test
- tracetools_trace
- trajectory_msgs
- turtlesim
- unique_identifier_msgs
- urdf
- urdf_parser_plugin
- visualization_msgs
- yaml_cpp_vendor
- zstd_vendor
action_msgs

- Change index.ros.org -> docs.ros.org. (#122)
- Updating Quality Declaration (#120)
- Update quality declaration to QL 1. (#116)
- Update package maintainers. (#112)
- Increase Quality level of packages to 3 (#108)
- Add Security Vulnerability Policy pointing to REP-2006. (#106)
- Updating QD to reflect package versions (#107)
- Contributors: Chris Lalancette, Michel Hidalgo, Stephen Brawner, brawner, shonigmann

action_tutorials_cpp

- Update logging macros (#476)
- Update the package.xml files with the latest Open Robotics maintainers (#466)
- Update goal response callback signature (#463)
- Contributors: Audrow Nash, Jacob Perron, Michael Jeronimo

action_tutorials_interfaces

- Update the package.xml files with the latest Open Robotics maintainers (#466)
- Contributors: Michael Jeronimo

action_tutorials_py

- Use underscores instead of dashes in setup.cfg (#502)
- Update the package.xml files with the latest Open Robotics maintainers (#466)
- Contributors: Ivan Santiago Paunovic, Michael Jeronimo

actionlib_msgs

- Update package maintainers. (#132)
- Contributors: Michel Hidalgo
ament_clang_format

- 0.10.3
- Add Audrow as a maintainer (#294)
- Drop trailing tab from package manifests (#291) Follow-up to 8bf194aa1ac282db5483dd0d3feff8f325b0db8
- Add clang-format-version option to ament_clang_format (#282)
- Update maintainer (#274) * update maintainer * add authors
- Add pytest.ini so local tests don’t display warning. (#259)
- Contributors: Audrow Nash, Chris Lalancette, Claire Wang, Scott K Logan, Tyler Weaver

ament_clang_tidy

- 0.10.3
- Add Audrow as a maintainer (#294)
- Add multiprocessing support to ament_clang_tidy (#288) * add multiprocessing support * fix stylistic lint issues
- Add –packages-select argument to ament_clang_tidy (#287) Add comment explaining handling quoted list of space separated package names Update documentation for ament_clang_tidy
- Update maintainer (#274) * update maintainer * add authors
- Add pytest.ini so local tests don’t display warning. (#259)
- Contributors: Audrow Nash, Chris Lalancette, Claire Wang, M. Mei, Tyler Weaver

ament_cmake

- Update package maintainers. (#286)
- Contributors: Michel Hidalgo

ament_cmake_auto

- Update package maintainers. (#286)
- Contributors: Michel Hidalgo
ament_cmake_clang_format

- 0.10.3
- Add Audrow as a maintainer (#294)
- Drop trailing tab from package manifests (#291) Follow-up to 8bf194aa1ac282db5483dd0d3feff8f325b0db8
- Update maintainer (#274) * update maintainer * add authors
- Contributors: Audrow Nash, Chris Lalancette, Claire Wang, Scott K Logan

ament_cmake_clang_tidy

- 0.10.3
- add TIMEOUT argument to ament_cmake_clang_tidy (#298)
- Add Audrow as a maintainer (#294)
- Fix documentation for ament_cmake_clang_tidy (#285)
- Update maintainer (#274) * update maintainer * add authors
- Contributors: Audrow Nash, Chris Lalancette, Claire Wang, M. Mei, Tyler Weaver

ament_cmake_copyright

- 0.10.3
- Add Audrow as a maintainer (#294)
- Drop trailing tab from package manifests (#291) Follow-up to 8bf194aa1ac282db5483dd0d3feff8f325b0db8
- Update maintainer (#274) * update maintainer * add authors
- increase default timeout for CMake copyright linter to 120s (#261)
- Contributors: Audrow Nash, Chris Lalancette, Claire Wang, Dirk Thomas, Scott K Logan

ament_cmake_core

- Merge pull request #287 from ament/mjeronimo/add-condition-support * Check condition attr in package.xml dependencies The condition attribute was already parsed when reading the XML file. Just needed to check the condition when adding dependencies to the list for a particular key/target. Fixes #266 * Address Dirk’s code review feedback
- Address Dirk’s code review feedback
• Check condition attr in package.xml dependencies. The condition attribute was already parsed when reading the XML file. Just needed to check the condition when adding dependencies to the list for a particular key/target. Fixes #266

• Update package maintainers. (#286)

• Contributors: Michael Jeronimo, Michel Hidalgo

ament_cmake_cppcheck

• Remove Claire as a maintainer. (#312) * Remove Claire as a maintainer. * Remove dead email addresses. * Remove more dead email addresses. * Switch setup.py maintainer to Audrow.

• 0.10.3

• Add Audrow as a maintainer (#294)

• Drop trailing tab from package manifests (#291) Follow-up to 8bf194aa1ac282db5483dd0d3fefff8f325b0db8

• Update maintainer (#274) * update maintainer * add authors

• Increase the ament_cmake_cppcheck timeout to 5 minutes. (#271) This will avoid timeouts on some slower platforms that we’ve started to see.

• parse LANGUAGE argument case insensitive (#255)

• Contributors: Audrow Nash, Chris Lalancette, Claire Wang, Karsten Knese, Scott K Logan

ament_cmake_cpplint

• Remove Claire as a maintainer. (#312) * Remove Claire as a maintainer. * Remove dead email addresses. * Remove more dead email addresses. * Switch setup.py maintainer to Audrow.

• 0.10.3

• Add Audrow as a maintainer (#294)

• Drop trailing tab from package manifests (#291) Follow-up to 8bf194aa1ac282db5483dd0d3fefff8f325b0db8

• Update maintainer (#274) * update maintainer * add authors

• Contributors: Audrow Nash, Chris Lalancette, Claire Wang, Scott K Logan

ament_cmake_export_definitions

• Update package maintainers. (#286)

• Contributors: Michel Hidalgo
ament_cmake_export_dependencies

- fix cmake list(TRANSFORM ) is only available from version 3.12, (#296) convert to string instead
- fix imported targets with multiple configuration (#290) * fix imported targets with multiple configuration * taking into account DEBUG_CONFIGURATIONS global variable
- Update package maintainers. (#286)
- Contributors: Michel Hidalgo, siposcsaba89

ament_cmake_export_include_directories

- Update package maintainers. (#286)
- Contributors: Michel Hidalgo

ament_cmake_export_interfaces

- Update package maintainers. (#286)
- Contributors: Michel Hidalgo

ament_cmake_export_libraries

- Fix variable name in ament_export_libraries.cmake (#314)
- Update package maintainers. (#286)
- Contributors: Alejandro Hernández Cordero, Michel Hidalgo

ament_cmake_export_link_flags

- Update package maintainers. (#286)
- Contributors: Michel Hidalgo

ament_cmake_export_targets

- Update package maintainers. (#286)
- Contributors: Michel Hidalgo
ament_cmake_flake8

- 0.10.3
- Add Audrow as a maintainer (#294)
- Update maintainer (#274) * update maintainer * add authors
- Contributors: Audrow Nash, Chris Lalancette, Claire Wang

ament_cmake_gmock

- Update package maintainers. (#286)
- Contributors: Michel Hidalgo

ament_cmake_google_benchmark

- Serialize benchmarks within CTest by default (#308)
- Handle runtime failures in Google Benchmark (#294) This change will handle runtime failures in Google Benchmark by propagating error information from Google Benchmark to both CTest and the Jenkins benchmark plugin.
- Use consistent string format and resolve flake8 (#295) Follow-up to a5fb3112b5c46c42b1824c96af4171d469eb13bf
- Make ament_cmake_test a dep of ament_cmake_google_benchmark (#293)
- Catch JSONDecodeError and printout some debug info (#291)
- Update package maintainers. (#286)
- Make AMENT_RUN_PERFORMANCE_TESTS a CMake option (#280)
- Skip performance tests using a CMake variable (#278) These tests can be fairly heavy, so we don’t want to run them by default. It would be better if there was a way to skip the tests by default in such a way that they could be specifically un-skipped at runtime, but I can’t find a mechanism in CMake or CTest that would allow us to achieve that behavior without leveraging environment variables.
- Handle Google Benchmark ‘aggregate’ results (#276) Previously, I assumed all results generated by Google Benchmark were of ‘iteration’ type. Now that I have more experience with Google Benchmark, I’ve started generating aggregate results, which contain some different properties. This change adds support for aggregate results and should make it easy to add any other result schemas we encounter in the future. For forward-compatibility, unsupported types will generate a warning message but will not fail the test. This makes the conversion tolerant to Google Benchmark adding new measures for existing mechanisms.
- Initial Google Benchmark results conversion (#275)
- Handle missing results file for Google Benchmark (#265)
- Initial ament_cmake_google_benchmark package (#261)
- Contributors: Michel Hidalgo, Scott K Logan, brawner
ament_cmake_gtest

• Disable gtest warning when building in Release (#298) https://github.com/google/googletest/issues/1303
• Update package maintainers. (#286)
• [ament_cmake_gtest] ensure gtest to consume the correct headers. (#267) * ensure gtest to consume the correct headers. * add another patch.
• Contributors: Michel Hidalgo, Sean Yen, Victor Lopez

ament_cmake_include_directories

• Update package maintainers. (#286)
• Contributors: Michel Hidalgo

ament_cmake_libraries

• Update package maintainers. (#286)
• Contributors: Michel Hidalgo

ament_cmake_lint_cmake

• Remove Claire as a maintainer. (#312) * Remove Claire as a maintainer. * Remove dead email addresses. * Remove more dead email addresses. * Switch setup.py maintainer to Audrow.
• ament_lint_cmake: default linelength in argumentparser for consistency (#306)
• 0.10.3
• Fix ament_lint_cmake line length expression (#236) This regular expression is using the re.VERBOSE flag, meaning that characters after an un-escaped ‘#’ character are interpreted as a comment and are not part of the expression. Also set the default maximum line length to 140 columns.
• Add Audrow as a maintainer (#294)
• Make CMake linter line length configurable (#235) Co-authored-by: Miaofei <miaoife@amazon.com>
• Drop trailing tab from package manifests (#291) Follow-up to 8bf194aa1ac282db5483dd0d3effff8f325b0db8
• Update maintainer (#274) * update maintainer * add authors
• Contributors: Audrow Nash, Chris Lalancette, Claire Wang, Emerson Knapp, Scott K Logan

ament_cmake_mypy

• Remove Claire as a maintainer. (#312) * Remove Claire as a maintainer. * Remove dead email addresses. * Remove more dead email addresses. * Switch setup.py maintainer to Audrow.
• 0.10.3
• Add Audrow as a maintainer (#294)
• Update maintainer (#274) * update maintainer * add authors
• Contributors: Audrow Nash, Chris Lalancette, Claire Wang
ament_cmake_nose

- Update package maintainers. (#286)
- Contributors: Michel Hidalgo

ament_cmake_pclint

- 0.10.3
- Add Audrow as a maintainer (#294)
- Drop trailing tab from package manifests (#291) Follow-up to 8bf194aa1ac282db5483dd0d3feff8f325b0db8
- Update maintainer (#274) * update maintainer * add authors
- Contributors: Audrow Nash, Chris Lalancette, Claire Wang, Scott K Logan

ament_cmake_pep257

- 0.10.3
- Add Audrow as a maintainer (#294)
- Drop trailing tab from package manifests (#291) Follow-up to 8bf194aa1ac282db5483dd0d3feff8f325b0db8
- Update maintainer (#274) * update maintainer * add authors
- Contributors: Audrow Nash, Chris Lalancette, Claire Wang, Scott K Logan

ament_cmake_pycodestyle

- 0.10.3
- Add Audrow as a maintainer (#294)
- Drop trailing tab from package manifests (#291) Follow-up to 8bf194aa1ac282db5483dd0d3feff8f325b0db8
- Update maintainer (#274) * update maintainer * add authors
- Contributors: Audrow Nash, Chris Lalancette, Claire Wang, Scott K Logan
ament_cmake_pyflakes

- 0.10.3
- Add Audrow as a maintainer (#294)
- Drop trailing tab from package manifests (#291) Follow-up to 8bf194aa1ae282db5483dd0d3efff8f325b0db8
- Update maintainer (#274) * update maintainer * add authors
- Contributors: Audrow Nash, Chris Lalancette, Claire Wang, Scott K Logan

ament_cmake_pytest

- Fix ament_get_pytest_cov_version for newer versions of pytest (#315)
- Update package maintainers. (#286)
- Contributors: Christophe Bedard, Michel Hidalgo

ament_cmake_python

- Symlink setup.cfg and sources before building Python egg-info (#327)
- Simplify ament_python_install_package() macro. (#326) Do not delegate to setuptools, install egg-info manually.
- Escape $ENV{DESTDIR} everywhere in ament_python_install_package() (#324) Follow up after f80071e2216e766f7bf1b0792493a5f6523e9226
- Use DESTDIR on ament_python_install_package() (#323) * Use DESTDIR on ament_python_install_package()
- Make ament_python_install_package() install a flat Python egg (#316)
- [ament_cmake_python] ament_cmake_python_get_python_install_dir public (#300) * [ament_cmake_python] make the ament_cmake_python_get_python_install_dir a public interface.
- Update package maintainers. (#286)
- Contributors: Michel Hidalgo, Naveau

ament_cmake_ros

- Update package maintainers. (#11)
- Contributors: Michel Hidalgo
ament_cmake_target_dependencies

- Force SYSTEM keyword in ament_target_dependencies() at the start. (#303)
- Add SYSTEM keyword option to ament_target_dependencies (#297) * Add SYSTEM keyword option to ament_target_dependencies
- Add documentation of SYSTEM keyword for ament_target_dependencies
- Update package maintainers. (#286)
- ordered interface include dirs and use privately to ensure workspace order (#260)
- Contributors: Andre Nguyen, Dirk Thomas, Michel Hidalgo

ament_cmake_test

- Update package maintainers. (#286)
- Fix skipped test reporting in CTest (#279) This is a follow-up to c67cdf2. When the SKIP_RETURN_CODE gets set to 0, the value is interpreted as ‘false’, and the test property is never actually added.
- limit test time to three decimals (#271)
- Add actual test time to xUnit result files (#270) * Add actual test time to xUnit result files
- Report test_time even with skipped test * Set time attribute for testcase element
- Add SKIP_RETURN_CODE argument to ament_add_test (#264) This makes the run_test.py wrapper aware of the SKIP_RETURN_CODE property on CTest tests. In the existing implementation, the wrapper detects that no result file was generated and overrides the special return code coming from the test, making the the CTest feature fail completely. This change makes the wrapper script aware of the special return code, and when detected, will write a ‘skipped’ result file instead of a ‘failed’ result file, and pass along the special return code as-is. Now the gtest result and the ctest results both show the test as ‘skipped’ when the special return flag is used. Note that none of this behavior is enabled by default, which is important because we wouldn’t want a test to fail and return a code which we’ve decided is the special ‘skip’ return code. Only tests which are aware of this feature should use it.
- Contributors: Dirk Thomas, Michel Hidalgo, Ruffin, Scott K Logan

ament_cmake_uncrustify

- Remove Claire as a maintainer. (#312) * Remove Claire as a maintainer.
- Remove dead email addresses. * Remove more dead email addresses.
- Switch setup.py maintainer to Audrow.
- 0.10.3
- Add Audrow as a maintainer (#294)
- Drop trailing tab from package manifests (#291) Follow-up to 8bf194aa1ac282db5483dd0d3efff8f325b0db8
- Update maintainer (#274) * update maintainer * add authors
- parse LANGUAGE argument case insensitive (#255)
- Contributors: Audrow Nash, Chris Lalancette, Claire Wang, Karsten Knese, Scott K Logan

4.8. ROS 2 Documentation
ament_cmake_version

- Update package maintainers. (#286)
- Contributors: Michel Hidalgo

ament_cmake_xmllint

- 0.10.3
- Add Audrow as a maintainer (#294)
- Drop trailing tab from package manifests (#291) Follow-up to 8bf194aa1ac282db5483dd0d3efff8f325b0db8
- Update maintainer (#274) * update maintainer * add authors
- Contributors: Audrow Nash, Chris Lalancette, Claire Wang, Scott K Logan

ament_copyright

- Use non-blind except for open() (307)
- Add optional file header style (#304) * Add optional file header style * Fix test on ament_copyright
- 0.10.3
- Add Audrow as a maintainer (#294)
- Drop trailing tab from package manifests (#291) Follow-up to 8bf194aa1ac282db5483dd0d3efff8f325b0db8
- add mit-0 as a valid license to ament_copyright (#284)
- Support Python 3.8-provided importlib.metadata (#290) The importlib_metadata package is a backport of the importlib.metadata module from Python 3.8. Fedora (and possibly others) no longer package importlib_metadata because they ship Python versions which have the functionality built-in.
- Update maintainer (#274) * update maintainer * add authors
- added bsd 2 clause simplified license to ament_copyright (#267) * added bsd 2 clause simplified license to ament_copyright
- Remove use of pkg_resources from ament_lint. (#260) Replace it with the use of the more modern importlib_metadata library. There are a couple of reasons to do this: 1. pkg_resources is quite slow to import; on my machine, just firing up the python interpreter takes ~35ms, while firing up the python interpreter and importing pkg_resources takes ~175ms. Firing up the python interpreter and importing importlib_metadata takes ~70ms. Removing 100ms per invocation of the command-line both makes it speedier for users, and will speed up our tests (which call out to the command-line quite a lot). 2. pkg_resources is somewhat deprecated and being replaced by importlib. https://importlib-metadata.readthedocs.io/en/latest/using.html describes some of it Note: By itself, this change is not enough to completely remove our dependence on pkg_resources. We’ll also have to do something about the console_scripts that setup.py generates. That will be a separate effort.
- Add pytest.ini so local tests don’t display warning. (#259)
- Contributors: Alfi Maulana, Audrow Nash, Chris Lalancette, Christophe Bedard, Claire Wang, Evan Flynn, M. Mei, Scott K Logan
ament_cppcheck


- 0.10.3

- Fix file exclusion behavior in ament_cppcheck and ament_cpplint (#299) * fix exclude behavior in ament_cppcheck and ament_cpplint * fix flake8 errors * add missing realpath() conversion

- Add Audrow as a maintainer (#294)

- Drop trailing tab from package manifests (#291) Follow-up to 8bf194aa1ac282db5483dd0d3f6ff8f325b0db8

- Suppress unknownMacro (#268) cppcheck creates an unknownMacro error when it cannot resolve a macro. Since we don’t pass in all dependent headers, we don’t expect all macros to be discoverable by cppcheck.

- Update maintainer (#274) * update maintainer * add authors

- Add pytest.ini so local tests don’t display warning. (#259)

- Contributors: Audrow Nash, Chris Lalancette, Claire Wang, Dan Rose, M. Mei, Scott K Logan

ament_cpplint


- 0.10.3

- Fix file exclusion behavior in ament_cppcheck and ament_cpplint (#299) * fix exclude behavior in ament_cppcheck and ament_cpplint * fix flake8 errors * add missing realpath() conversion

- Add Audrow as a maintainer (#294)

- Drop trailing tab from package manifests (#291) Follow-up to 8bf194aa1ac282db5483dd0d3f6ff8f325b0db8

- Update maintainer (#274) * update maintainer * add authors

- Add pytest.ini so local tests don’t display warning. (#259)

- Contributors: Audrow Nash, Chris Lalancette, Claire Wang, M. Mei, Scott K Logan

ament_flake8


- 0.10.3

- Add Audrow as a maintainer (#294)

- Update maintainer (#274) * update maintainer * add authors

- Add pytest.ini so local tests don’t display warning. (#259)

- Contributors: Audrow Nash, Chris Lalancette, Claire Wang
ament_index_cpp

- Remove Claire as the maintainer. (#71)
- Change links from index.ros.org -> docs.ros.org (#70)
- Add Audrow as a maintainer (#68)
- update maintainers (#67)
- Update QD to Quality Level 1 (#66)
- add rational why ament_index pkgs don’t have explicit performance tests (#65)
- Fixed Doxygen warnings (#63)
- Remove the Quality Level from the README.md. (#62)
- Update QD ament_index_cpp to QL 2 (#59)
- Add Security Vulnerability Policy pointing to REP-2006. (#57)
- [Quality Declaration] Update Version Stability to stable version (#58)
- Contributors: Alejandro Hernández Cordero, Audrow Nash, Chris Lalancette, Claire Wang, Dirk Thomas, brawner

ament_index_python

- Remove Claire as the maintainer. (#71)
- Change links from index.ros.org -> docs.ros.org (#70)
- Add Audrow as a maintainer (#68)
- update maintainers (#67)
- add rational why ament_index pkgs don’t have explicit performance tests (#65)
- Remove the Quality Level from the README.md. (#62)
- Fix document link (#61)
- [Quality Declaration] Update Version Stability to stable version (#58)
- Contributors: Alejandro Hernández Cordero, Audrow Nash, Chris Lalancette, Claire Wang, Dirk Thomas, Matthijs van der Burgh

ament_lint

- 0.10.3
- Add Audrow as a maintainer (#294)
- Drop trailing tab from package manifests (#291) Follow-up to 8b1f194aa1ac282db5483dd0d3feff8f325b0db8
- Update maintainer (#274) * update maintainer * add authors
- Add pytest.ini so local tests don’t display warning. (#259)
- Contributors: Audrow Nash, Chris Lalancette, Claire Wang, Scott K Logan
ament_lint_auto


- 0.10.3

- Add Audrow as a maintainer (#294)

- Drop trailing tab from package manifests (#291) Follow-up to 8bf194aa1ae282db5483dd0d3feff8f325b0db8

- Use correct lint package dependencies (#278)

- Update maintainer (#274) * update maintainer * add authors

- Contributors: Audrow Nash, Chris Lalancette, Claire Wang, Esteve Fernandez, Scott K Logan

ament_lint_cmake


- ament_lint_cmake: default linelength in argumentparser for consistency (#306)

- 0.10.3

- Fix ament_lint_cmake line length expression (#236) This regular expression is using the re.VERBOSE flag, meaning that characters after an un-escaped ‘#’ character are interpreted as a comment and are not part of the expression. Also set the default maximum line length to 140 columns.

- Add Audrow as a maintainer (#294)

- Make CMake linter line length configurable (#235) Co-authored-by: Miaofei <miaofei@amazon.com>

- Drop trailing tab from package manifests (#291) Follow-up to 8bf194aa1ae282db5483dd0d3feff8f325b0db8

- Update maintainer (#274) * update maintainer * add authors

- Add pytest.ini so local tests don’t display warning. (#259)

- Contributors: Audrow Nash, Chris Lalancette, Claire Wang, Emerson Knapp, Scott K Logan

ament_lint_common


- 0.10.3

- Add Audrow as a maintainer (#294)

- Drop trailing tab from package manifests (#291) Follow-up to 8bf194aa1ae282db5483dd0d3feff8f325b0db8

- Update maintainer (#274) * update maintainer * add authors

- Contributors: Audrow Nash, Chris Lalancette, Claire Wang, Scott K Logan
Vulcanexus Documentation, Release 1.0.0

ament_mypy

- Remove Claire as a maintainer. (#312)
- Remove Claire as a maintainer.
- Remove dead email addresses.
- Remove more dead email addresses.
- Switch setup.py maintainer to Audrow.
- 0.10.3
- Add Audrow as a maintainer (#294)
- Update maintainer (#274)
- update maintainer
- add authors
- Add pytest.ini so local tests don’t display warning. (#259)
- Contributors: Audrow Nash, Chris Lalancette, Claire Wang

ament_package

- Revert “Generate Setuptools Dict Helper Method (#126)” (#131)
- Generate Setuptools Dict Helper Method (#126)
- Add Audrow as a maintainer (#127)
- Support Python 3.8-provided importlib.metadata (#124)
- Declare missing dependency on python3-importlib-resources (#123)
- make AMENT_TRACE_SETUP_FILES output sourceable (#120)
- update maintainers
- Switch ament_package to using importlib. (#118)
- Add pytest.ini so local tests don’t display warning (#117)
- add configure-time flag to skip parent_prefix_path (#115)
- Contributors: Audrow Nash, Chris Lalancette, David V. Lu!!, Dirk Thomas, Mabel Zhang, Scott K Logan

ament_pclint

- Remove Claire as a maintainer. (#312)
- Remove Claire as a maintainer.
- Remove dead email addresses.
- Remove more dead email addresses.
- Switch setup.py maintainer to Audrow.
- 0.10.3
- Add Audrow as a maintainer (#294)
- Add pytest marks to ament_pclint tests. (#202)
- Add pytest marks to ament_pclint tests.
- fix failed tests
  Co-authored-by: Miaofei <miaofei@amazon.com>
- Drop trailing tab from package manifests (#291)
- Follow-up to 8bf194aa1ac282db5483dd0d3f3eff8f325b0db8
- Update maintainer (#274)
- update maintainer
- add authors
- Add pytest.ini so local tests don’t display warning. (#259)
- Contributors: Audrow Nash, Chris Lalancette, Claire Wang, Scott K Logan, Steven! Ragnarök

Chapter 4. Contributing to the documentation
**ament_pep257**

- 0.10.3
- Add Audrow as a maintainer (#294)
- Drop trailing tab from package manifests (#291) Follow-up to 8bf194aa1ac282db5483dd0d3feff8f325b0db8
- Update maintainer (#274) * update maintainer * add authors
- remove use of “extend” action in argparse (#262)
- Expand ignores to pep257 definition. (#241) * Expand ignores to pep257 definition. (ament #240) * add ‘--allow-undocumented’ flag to enforce pep257 * restore existing default error codes to check * fix no-ignores logic * expose options from pydocstyle * allow user to explicitly set convention to “ament” * fix typo in populating argv for pydocstyle * reformat ament convention list * Add help info for ament convention
- Add pytest.ini so local tests don’t display warning. (#259)
- remove match args to allow pydocstyle defaults (#243)
- Contributors: Audrow Nash, Chris Lalancette, Claire Wang, Scott K Logan, Ted Kern

**ament_pycodestyle**

- 0.10.3
- Add Audrow as a maintainer (#294)
- Drop trailing tab from package manifests (#291) Follow-up to 8bf194aa1ac282db5483dd0d3feff8f325b0db8
- Update maintainer (#274) * update maintainer * add authors
- Add pytest.ini so local tests don’t display warning. (#259)
- Contributors: Audrow Nash, Chris Lalancette, Claire Wang, Scott K Logan

**ament_pyflakes**

- 0.10.3
- Add Audrow as a maintainer (#294)
- Drop trailing tab from package manifests (#291) Follow-up to 8bf194aa1ac282db5483dd0d3feff8f325b0db8
- Update maintainer (#274) * update maintainer * add authors
- Add pytest.ini so local tests don’t display warning. (#259)
- Contributors: Audrow Nash, Chris Lalancette, Claire Wang, Scott K Logan
ament_uncrustify

- 0.10.3
- Allow ‘C++’ as language, but convert it to ‘CPP’ (#302)
- Allow correct languages on uncrustify (#272) * Allow correct languages on uncrustify. * Update dictionary.
- Add Audrow as a maintainer (#294)
- Drop trailing tab from package manifests (#291) Follow-up to 8bf194aa1ac282db5483dd0d3feff8f325b0db8
- Update maintainer (#274) * update maintainer * add authors
- Add pytest.ini so local tests don’t display warning. (#259)
- Contributors: Audrow Nash, Chris Lalancette, Claire Wang, Miguel Company, Scott K Logan

ament_xmllint

- 0.10.3
- Add Audrow as a maintainer (#294)
- Drop trailing tab from package manifests (#291) Follow-up to 8bf194aa1ac282db5483dd0d3feff8f325b0db8
- Update maintainer (#274) * update maintainer * add authors
- Add pytest.ini so local tests don’t display warning. (#259)
- Contributors: Audrow Nash, Chris Lalancette, Claire Wang, Scott K Logan

builtin_interfaces

- Change index.ros.org -> docs.ros.org. (#122)
- Updating Quality Declaration (#120)
- Update quality declaration to QL 1. (#116)
- Update package maintainers. (#112)
- Increase Quality level of packages to 3 (#108)
- Document that Time and Duration are explictly ROS Time (#103)
- Add Security Vulnerability Policy pointing to REP-2006. (#106)
- Updating QD to reflect package versions (#107)
- Contributors: Chris Lalancette, Michel Hidalgo, Stephen Brawner, Tully Foote, brawner, shonigmann
camera_calibration_parsers

- Fix formatting and include paths for linters (#157)
- ROS2 Using the filesystem helper in rcpputils (#133)
- Camera Calibration Parsers ROS2 Port (#105)
- Image Transport ROS2 port (#84)
- Use Boost_LIBRARIES instead of Boost_PYTHON_LIBRARY This was causing issues when building with python3 since then Boost_PYTHON_LIBRARY is not set, instead cmake sets Boost_PYTHON3_LIBRARY. So instead of adding each library separately, using Boost_LIBRARIES seems to be better. For reference, from the cmake docs: `Boost_LIBRARIES - Boost component libraries to be linked Boost\_<C>_LIBRARY - Libraries to link for component <C>`
- Properly detect Boost Python 2 or 3 This fixes #59
- 1.11.11
- update changelogs
- Add install target for python wrapper library
- Only link against needed Boost libraries 9829b02 introduced a python dependency into find_package(Boost..) which results in $Boost_LIBRARIES containing boost_python and such a dependency to libpython at link time. With this patch we only link against the needed libraries.
- Add python wrapper for readCalibration. Reads .ini or .yaml calibration file and returns camera name and sensor_msg_msgs/cameraInfo.
- Use $catkin_EXPORTED_TARGETS
- Remove no-longer-neccessary flags to allow OS X to use 0.3 and 0.5 of yaml-cpp.
- remove buggy CMake message
- fix #39
- make sure test does not fail
- [camera_calibration_parsers] Better error message when calib file can't be written
- add rosbash as a test dependency
- add a test dependency now that we have tests
- parse distortion of arbitraty length in INI This fixes #33
- add a test to parse INI calibration files with 5 or 8 D param
- Add yaml-cpp case for building on Android
- Fix catkin_make failure (due to yaml-cpp deps) for mac os
- fix bad yaml-cpp usage in certain conditions fixes #24
- add a dependency on pkg-config to have it work on Indigo
- fix YAML CPP 0.5.x compatibility
- Contributors: Andreas Klintberg, Gary Servin, Helen Oleynikova, Isaac IY Saito, Jochen Sprickerhof, Kartik Mohta, Markus Roth, Martin Idel, Michael Carroll, Sean Yen, Vincent Rabaud, Yifei Zhang

4.8. ROS 2 Documentation 359
camera_info_manager

- Fix formatting and include paths for linters (#157)
- Enable Windows build. (#159)
- Fix abort criteria for setCameraInfoService callback (#132)
- camera_info_manager ROS2 port (#94)
- Image Transport ROS2 port (#84)
- Fix the find_package(catkin) redundancy
- Add a dependency between the test and the test executable
- Add camera_calibration_parsers dependency to camera_info_manager

1.11.11

- update changelogs
- Return empty CameraInfo when !ros::ok()
- Return empty CameraInfo when !ros::ok()
- fix compilation on Fedora, fixes #42
- simplify target_link_libraries That should fix #35
- Add public member function to manually set camera info (#19)
- make rostest in CMakeLists optional (ros/rosdistro#3010)
- check for CATKIN_ENABLE_TESTING
- add Jack as maintainer
- add gtest libraries linkage
- fix the rostest dependency
- fix catkin gtest and rostest problem
- fix unit test dependencies
- Removed duplicated test dependancy Test dependencies should never duplicate build or run dependencies.
- fix the urls
- Updated package.xml file(s) to handle new catkin buildtool_depend requirement
- remove the brief attribute
- fix bad folder/libraries
- add missing rostest dependency
- fix bad dependency
- fix dependencies
- add catkin as a dependency
- comply to the catkin API
- add missing linkage
- install the include directories
- fix build issues
• make the libraries public
• API documentation review update
• suppress misleading camera_info_manager error messages [#5273]
• remove deprecated global CameraInfoManager symbol for Fuerte (#4971)
• Revert to using boost::mutex, not boost::recursive_mutex.
• Hack saveCalibrationFile() to stat() the containing directory and attempt to create it if necessary. Test for this case.
• Reload camera info when camera name changes.
• Implement most new Electric API changes, with test cases.
• Add ${ROS_HOME} expansion, with unit test cases. Do not use "$$" for a single ‘$’, look for “${" instead.
• Use case-insensitive comparisons for parsing URL tags (#4761). Add unit test cases to cover this. Add unit test case for camera name containing video mode.
• add test for resolving an empty URL
• Deprecate use of global CameraInfoManager symbol in E-turtle (#4786). Modify unit tests accordingly.
• provide camera_info_manager namespace, fixes #4760
• Add support for “package://” URLs.
• Fixed tests to work with new CameraInfo.
• Moved image_common from camera_drivers.
• Contributors: Aaron Blasdel, Enrique Fernandez, Jack O’Quin, Jonathan Bohren, Joseph Schornak, Lukas Bulwahn, Martin Idel, Max Schettler, Michael Carroll, Sean Yen, Vincent Rabaud, blaise, mihelich, mirzashah

class_loader

• Remove travis. (#182)
• Change index.ros.org -> docs.ros.org. (#181)
• Fix ternary null check found by clang static analysis (#176)
• Update QD to QL 1 (#177)
• Updated console_bridge QL in QD
• Update package maintainers. (#169)
• enable building a static library (#163)
• Update Quality Declaration to reflect QL 2 (#160).
• Increase coverage with a graveyard behavior test and unmanaged instance test (#159)
• Add Security Vulnerability Policy pointing to REP-2006. (#157)
• Clean up and improve documentation (#156)
• Contributors: Alejandro Hernández Cordero, Chris Lalancette, Dirk Thomas, Michel Hidalgo, Stephen Brawner, ahorcde, brawner
common_interfaces

- Update package maintainers. (#132)
- Contributors: Michel Hidalgo

composition

- Fix leak (#480) (#481)
- Update the package.xml files with the latest Open Robotics maintainers (#466)
- Contributors: Michael Jeronimo, y-okumura-isp

composition_interfaces

- Change index.ros.org -> docs.ros.org. (#122)
- Updating Quality Declaration (#120)
- Update quality declaration to QL 1. (#116)
- Update package maintainers. (#112)
- Increase Quality level of packages to 3 (#108)
- Add Security Vulnerability Policy pointing to REP-2006. (#106)
- Updating QD to reflect package versions (#107)
- Contributors: Chris Lalancette, Michel Hidalgo, Stephen Brawner, brawner, shonigmann

cyclonedds

- DATA_AVAILABLE was not always triggered when by a dispose and sometimes triggered in the absence of an observable state change (arrival of a dispose for an already-disposed instance where the dispose had not yet been read);
- Restores functionality of the “raw ethernet” mode as well as IPv6 with link-local addresses, both accidentally broken in 0.6.0;
- Fixes a crash in processing endpoint discovery data containing unrecognised locator kinds;
- Fixes type conversion for local historical data (e.g., mixed use of ROS 2 C/C++ type supports in combination with transient-local endpoints within a single process);
- Fixes a use-after-free of “lease” objects with manual-by-topic writers;
- Mark instance as “alive” in the reader history and generate an invalid sample to notify the application even if the sample itself is dropped because the same or a later one is present already (e.g., on reconnecting to a transient-local writer);
- Fix a crash when doing an instance lookup on a built-in topic using the key value;
- No longer autoDispose instances as soon as some registered writer disappears, instead do it only when all of them have unregistered it;
- Fix performance of read_instance and take_instance by performing a proper instance lookup.
demo_nodes_cpp

- Fix small print issue in allocator tutorial. (#509) (#512)
- Small fixes for even_parameters_node. (#500)
- change ParameterEventHandler to take events as const ref instead of shared pointer (#494)
- Fix integer type in RCLCPP_* macro printf. (#492)
- Add a demo for the new ParameterEventHandler class (#486)
- Filter qos overrides in parameter events demos (#491)
- Update code now that parameter types are static by default (#487)
- Update logging macros (#476)
- Make sure to wait for the service before declaring events. (#473)
- Update the package.xml files with the latest Open Robotics maintainers (#466)
- Contributors: Audrow Nash, Chris Lalancette, Ivan Santiago Paunovic, Michael Jeronimo, William Woodall

demo_nodes_cpp_native

- Update demo_nodes_cpp_native to new Fast DDS API (#493)
- Update the package.xml files with the latest Open Robotics maintainers (#466)
- Contributors: Michael Jeronimo, Miguel Company

demo_nodes_py

- Use underscores instead of dashes in setup.cfg (#502)
- Update deprecated qos policy value names (#468)
- Update the package.xml files with the latest Open Robotics maintainers (#466)
- Contributors: Ivan Santiago Paunovic, Michael Jeronimo

diagnosticmsgs

- Change index.ros.org -> docs.ros.org. (#149)
- updating quality declaration links (re: ros2/docs.ros2.org#52) (#145)
- Update QDs to QL 1 (#135)
- Update package maintainers. (#132)
- Updated Quality Level to 2 (#131)
- Update Quality levels to level 3 (#124)
- Add Security Vulnerability Policy pointing to REP-2006. (#120)
- Contributors: Alejandro Hernández Cordero, Chris Lalancette, Michel Hidalgo, Stephen Brawner, brawner, shonigmann
domain_coordinator

- Update package maintainers. (#11)
- Add pytest.ini to suppress warning output locally. (#8)
- Contributors: Chris Lalancette, Michel Hidalgo

dummy_map_server

- Update the package.xml files with the latest Open Robotics maintainers (#466)
- Contributors: Michael Jeronimo

dummy_robot Bringup

- Update the package.xml files with the latest Open Robotics maintainers (#466)
- Contributors: Michael Jeronimo

dummy_sensors

- Update the package.xml files with the latest Open Robotics maintainers (#466)
- Contributors: Michael Jeronimo

domain_sensors

example_interfaces

- Change links from index.ros.org -> docs.ros.org. (#13)
- Update maintainer. (#12)
- Contributors: Chris Lalancette, Jacob Perron

domain_sensors

examples_rclcpp_cbg_executor

- Fix clang warnings about type mismatches. (#309)
- Support for cbg_executor package on QNX (#305)
- Demo for callback-group-level executor concept. (#302)
- Contributors: Chris Lalancette, Ralph Lange, joshua-qnx
examples_rclcpp_minimal_action_client

- Update maintainers (#292)
- Update goal response callback signature (#291)
- Make sure to include what you use in all examples. (#284)
- Added common linters (#265)
- Contributors: Alejandro Hernández Cordero, Chris Lalancette, Jacob Perron, Shane Loretz

examples_rclcpp_minimal_action_server

- Update maintainers (#292)
- Make sure to include what you use in all examples. (#284)
- Added common linters (#265)
- Contributors: Alejandro Hernández Cordero, Chris Lalancette, Shane Loretz

examples_rclcpp_minimal_client

- Update maintainers (#292)
- Make sure to include what you use in all examples. (#284)
- Added common linters (#265)
- Contributors: Alejandro Hernández Cordero, Chris Lalancette, Shane Loretz

examples_rclcpp_minimal_composition

- Update maintainers (#292)
- Added common linters (#265)
- Contributors: Alejandro Hernández Cordero, Shane Loretz

examples_rclcpp_minimal_publisher

- Unique network flows (#296)
- Update maintainers (#292)
- Make sure to include what you use in all examples. (#284)
- Added common linters (#265)
- Contributors: Alejandro Hernández Cordero, Ananya Muddukrishna, Chris Lalancette, Shane Loretz
examples_rclcpp_minimal_service

- Update maintainers (#292)
- Make sure to include what you use in all examples. (#284)
- Added common linters (#265)
- Contributors: Alejandro Hernández Cordero, Chris Lalancette, Shane Loretz

examples_rclcpp_minimal_subscriber

- Unique network flows (#296)
- Update maintainers (#292)
- Make sure to include what you use in all examples. (#284)
- Remove a TODO in the not_composable demo. (#285)
- Add Topic Statistics Example (#281)
- Added common linters (#265)
- Contributors: Alejandro Hernández Cordero, Ananya Muddukrishna, Chris Lalancette, Devin Bonnie, Shane Loretz

examples_rclcpp_minimal_timer

- Update maintainers (#292)
- Added common linters (#265)
- Contributors: Alejandro Hernández Cordero, Shane Loretz

examples_rclcpp_multithreaded_executor

- Use char * in logging macros (#295)
- Update maintainers (#292)
- Added common linters (#265)
- Contributors: Alejandro Hernández Cordero, Audrow Nash, Shane Loretz

examples_rclpy_executors

- Use underscores instead of dashes in setup.cfg (#310)
- Update maintainers (#292)
- Contributors: Ivan Santiago Paunovic, Shane Loretz
examples_rclpy_guard_conditions

- Use underscores instead of dashes in setup.cfg (#310)
- Update maintainers (#292)
- [rclpy] Create a package with an example showing how guard conditions work (#283)
- Contributors: Audrow Nash, Ivan Santiago Paunovic, Shane Loretz

examples_rclpy_minimal_action_client

- Use underscores instead of dashes in setup.cfg (#310)
- Using asyncio with ros2 action client (#301)
- Update maintainers (#292)
- Added missing linting tests (#287)
- Contributors: Allison Thackston, Ivan Santiago Paunovic, Shane Loretz, alemme

examples_rclpy_minimal_action_server

- Use underscores instead of dashes in setup.cfg (#310)
- Update maintainers (#292)
- Added missing linting tests (#287)
- Contributors: Allison Thackston, Ivan Santiago Paunovic, Shane Loretz

examples_rclpy_minimal_client

- Use underscores instead of dashes in setup.cfg (#310)
- Remove bare exception catching (#299)
- Update maintainers (#292)
- Contributors: Ivan Santiago Paunovic, Shane Loretz

examples_rclpy_minimal_publisher

- Use underscores instead of dashes in setup.cfg (#310)
- Update maintainers (#292)
- Contributors: Ivan Santiago Paunovic, Shane Loretz
examples_rclpy_minimal_service

- Use underscores instead of dashes in setup.cfg (#310)
- Update maintainers (#292)
- Contributors: Ivan Santiago Paunovic, Shane Loretz

examples_rclpy_minimal_subscriber

- Use underscores instead of dashes in setup.cfg (#310)
- Update maintainers (#292)
- Contributors: Ivan Santiago Paunovic, Shane Loretz

examples_rclpy_pointcloud_publisher

- Use underscores instead of dashes in setup.cfg (#310)
- add pointcloud publisher example (#276)
- Contributors: Evan Flynn, Ivan Santiago Paunovic

examples_tf2_py

- Use underscores instead of dashes in setup.cfg. (#403) (#404)
- Update maintainers of the ros2/geometry2 fork. (#328)
- Add pytest.ini so local tests don’t display warning (#276)
- Split tf2_ros in tf2_ros and tf2_ros_py (#210)
- Contributors: Alejandro Hernández Cordero, Chris Lalancette

fastrtps_cmake_module

- updating quality declaration links (re: ros2/docs.ros2.org#52) (#69)
- Use CMake config dirs as hint for header/library search (#56)
- Update package maintainers (#55)
- QD Update Version Stability to stable version (#46)
- Contributors: Alejandro Hernández Cordero, Dirk Thomas, Michel Hidalgo, shonigmann
geometry2

- Port eigen_kdl.h/cpp to ROS2 (#311)
- Update maintainers of the ros2/geometry2 fork. (#328)
- Contributors: Chris Lalancette, Jafar Abdi

geometry_msgs

- Change index.ros.org -> docs.ros.org. (#149)
- updating quality declaration links (re: ros2/docs.ros2.org#52) (#145)
- Update QDs to QL 1 (#135)
- Update package maintainers. (#132)
- Updated Quality Level to 2 (#131)
- Update Quality levels to level 3 (#124)
- Finish up API documentation (#123)
- Add Security Vulnerability Policy pointing to REP-2006. (#120)
- Contributors: Alejandro Hernández Cordero, Chris Lalancette, Michel Hidalgo, Stephen Brawner, brawner, shonigmann

google_benchmark_vendor

- Shrink the size of the tz_offset variable. (#13)
- Update the patching to work on Windows without admin. (#11)
- Always preserve source permissions in vendor packages. (#12)
- Update package maintainers. (#10)
- Upgrade google benchmark from v1.5.1 to v1.5.2 to include QNX patch. (#9)
- Set the SOVERSION on benchmark libraries. (#8)
- Set minimum criteria for system package. (#3)
- Work around warnings building Google Benchmark w/Clang. (#2)
- Initial google_benchmark_vendor package. (#1)
- Initial commit.
- Contributors: Ahmed Sobhy, Chris Lalancette, Michel Hidalgo, Scott K Logan
image_common

- [ros2] image_common metapackage (#129)
- 1.11.11
- update changelogs
- add Jack as maintainer
- comply to REP 0127
- add missing description
- define metapackage
- Contributors: Vincent Rabaud, chapulina

image_tools

- Initialize time stamp for published image messages (#475)
- Update the package.xml files with the latest Open Robotics maintainers (#466)
- Added more parameters for camera topic examples (#465)
- Contributors: Jacob Perron, Michael Jeronimo

image_transport

- Fix formatting and include paths for linters (#157)
- Fix QoS initialization from RMW QoS profile (#158)
- add missing set header (#140)
- Update to use new count APIs (#128)
- use latest ros2 API (#127)
- Update ROS2 branch to account for new NodeOptions interface (#120)
- camera_info_manager ROS2 port (#94)
- Pointer api updates (#104)
- Fix rcutils API change by just removing it. (#103)
- [ROS2] corrections to remapping for raw images (#97)
- Make ROS2 ImageTransport conform to old api (#88)
- Image Transport ROS2 Port (#84)
- Disable image publisher plugins by name (#60) * Disable publisher plugins by name * Now have per publisher blacklist instead of image_transport wide.
- update to use non deprecated pluginlib macro
- Extend documentation of getCameraInfoTopic Document the fact that the base_topic argument must be resolved in order to build the correct camera info topic.
- Added cv::waitkey(10) for blank popup Without the cv::waitkey(10), it results in a blank popup which crashes/ leads to a black popup. This change corrects that problem. ROS Kinetic, Ubuntu 16.04.3
- Fix CMake of image_transport/tutorial and polled_camera Fix loads of problems with the CMakeLists.
- image_transport/tutorial: Add dependency on generated msg Without this, build fails on Kinetic because ResizedImage.h has not been generated yet.
- image_transport/tutorial: Add missing catkin_INCLUDE_DIRS Without this, compilation files on Kinetic because ros.h cannot be found.

1.11.11

- update changelogs
- fix linkage in tutorials
- Use $catkin_EXPORTED_TARGETS
- image_transport: fix CameraSubscriber shutdown (circular shared_ptr ref) CameraSubscriber uses a private boost::shared_ptr to share an impl object between copied instances. In CameraSubscriber::CameraSubscriber(), it handed this shared_ptr to boost::bind() and saved the created wall timer in the impl object, thus creating a circular reference. The impl object was therefore never freed. Fix that by passing a plain pointer to boost::bind().
- avoid a memory copy for the raw publisher
- add a way to publish an image with only the data pointer
- Make function inline to avoid duplicated names when linking statically
- add plugin examples for the tutorial
- update instructions for catkin
- remove uselessly linked library fixes #28
- add a tutorial for image_transport
- add Jack as maintainer
- update my email address
- fix the urls
- use the pluginlib script to remove some warnings
- added license headers to various cpp and h files
- get rid of the deprecated class_loader interface
- CMakeLists.txt clean up
- Updated package.xml file(s) to handle new catkin buildtool_depend requirement
- add the right link libraries
- Isolated plugins into their own library to follow new class_loader/pluginlib guidelines.
- remove the brief attribute
- add xml file
- fix bad folder/libraries
- fix dependencies
- add catkin as a dependency
- comply to the catkin API
- install the include directories
- make the libraries public
• catkinize for Groovy
• Initial image_common stack check-in, containing image_transport.
• Contributors: Aaditya Saraiya, Aaron Blasdel, Carl Delsey, Gary Servin, Jacob Perron, Jochen Sprickerhof, Karsten Knese, Lucas Walter, Martin Guenther, Martin Idel, Max Schwarz, Michael Carroll, Mikael Arguedas, Mirza Shah, Thibaud Chupin, Vincent Rabaud, William Woodall, gerkey, kwc, mihelich, mirzashah, pmihelich, straszheim, vrabaud

**interactive_markers**

• Cleanup bsd 3 clause license usage (#61)
• Add missing includes (#81)
• Update maintainers (#79)
• Increase test timeout necessary for Connext (#77)
• Fix clang warnings (#75)
• Remove explicit template parameter in spin_until_future_complete (#72)
• Contributors: Bjar Ne, Dirk Thomas, Jacob Perron, Sarthak Mittal, Tully Foote

**intra_process_demo**

• Update the package.xml files with the latest Open Robotics maintainers (#466)
• Contributors: Michael Jeronimo

**kdl_parser**

• Remove tinyxml dependency from kdl_parser. (#43)
• Remove unused find_library call (#40)
• Contributors: Chris Lalancette, Michael Carroll

**laser_geometry**

• Use rclepp::Duration::from_seconds (#72)
• update maintainers
• increase test timeout
• Contributors: Dirk Thomas, Ivan Santiago Paunovic, Jonathan Binney, Mabel Zhang
launch

• Only try to wrap the fd in a socket on Windows (#498)
• Close the socket pair used for signal management (#497)
• Remove is_winsock_handle() and instead test if wrapping the handle in a socket.socket() works (#494)
• Add frontend substitution for logging directory (#490)
• Add arg_choice arg to DeclareLaunchArguments (#483)
• Support Python 3.8-provided importlib.metadata (#482)
• Workaround asyncio signal handling on Unix (#479)
• Handle signals within the asyncio loop. (#476)
• Support non-interactive launch.LaunchService runs (#475)
• print stderr message when command failed (#474)
• Add frontend support for LogInfo action (#467)
• Validate unparsed attributes and subentities in launch_xml and launch_yml (#468)
• Fix bug in launch.actions.TimerAction.parse() (#470)
• Allow configuring logging directory through environment variables (#460)
• Update package maintainers (#465)
• Expose Timer action in launch xml (#462)
• Fix dollar symbols in substitution grammar (#461)
• Add new conditions for checking launch configuration values (#453)
• Refactor launch service run_async loop to wait on futures and queued events (#449)
• Fix documentation typo (#446)
• Fix type_utils.extract_type() function. (#445)
• Handle empty strings in type coercion. (#443)
• Consolidate type_utils in a way that can be reused in substitution results that need to be coerced to a specific type (#438)
• Delete unnecessary loading of ‘launch.frontend.interpolate_substitution_method’ entry point that was never used (#434)
• Avoid side effect, defer until needed (#432)
• Remove pkg_resources, replace it with the use of the more modern importlib* libraries. (#430)
• Remove the asyncio.wait loop parameter. (#429)
• Add pytest.ini so local tests don’t display warning (#428)
• Defer shutdown if already running (#427)
• Add respawn and respawn_delay support (#426)
• Fix up parser.py (#414)

Contributors: CHEN, Chris Lalancette, Christophe Bedard, Dan Rose, Dirk Thomas, Ivan Santiago Paunovic, Jacob Perron, Jorge Perez, Michel Hidalgo, Scott K Logan, Takamasa Horibe, Victor Lopez
launch_ros

- Support Python 3.8 importlib.metadata, declare dependency (#229)
- Add options extensions to ros2launch and extensibility to the node action (#216)
- Make sure ParameterFile __del__ works without exception. (#212)
- Fix docblock in LoadComposableNodes (#207)
- Validate complex attributes of 'node' action (#198)
- Node.__init__() executable and ComposableNode.__init__() plugin arguments aren't optional (#197)
- Remove constructors arguments deprecated since Foxy (#190)
- Make name and namespace mandatory in ComposableNodeContainer, remove deprecated alternatives (#189)
- Merge pull request #183 from ros2/update-maintainers Update the package.xml files with the latest Open Robotics maintainers
- Move previous maintainer to <author>
- Update the package.xml files with the latest Open Robotics maintainers
- Fix AttributeError when accessing component container name (#177)
- Handle any substitution types for SetParameter name argument (#182)
- Asynchronously wait for load node service response (#174)
- Fix case where list of composable nodes is zero (#173)
- Do not use event handler for loading composable nodes (#170)
- Fix race with launch context changes when loading composable nodes (#166)
- Substitutions in parameter files (#168)
- Fix documentation typo (#167)
- Fix problems when parsing a Command Substitution as a parameter value (#137)
- Add a way to set remapping rules for all nodes in the same scope (#163)
- Resolve libyaml warning when loading parameters from file (#161)
- Fix ComposableNode ignoring PushRosNamespace actions (#162)
- Add a SetParameter action that sets a parameter to all nodes in the same scope (#158)
- Make namespace parameter mandatory in LifecycleNode constructor (#157)
- Avoid using a wildcard to specify parameters if possible (#154)
- Fix no specified namespace (#153)
- Add pytest.ini so local tests don’t display warning (#152)
- Contributors: Chris Lalancette, Dereck Wonnacott, Geoffrey Biggs, Ivan Santiago Paunovic, Jacob Perron, Michael Jeronimo, Scott K Logan
**launch_testing**

- Use unittest.mock instead of mock (#487)
- Update package maintainers (#465)
- Disable cleanup of test cases once they have been run (#406)
- Fix max() with empty sequence (#440)
- Use unittest.TestCase.id() for pytest failure reprs. (#436)
- Use unittest.TestCase.id() to put together jUnit XML output. (#435)
- Claim ownership (#433)
- Contributors: Dirk Thomas, Michel Hidalgo, Scott K Logan, William Woodall

**launch_testing_ament_cmake**

- Update package maintainers (#465)
- Add bsd license to launch due to files from roslaunch (#456)
- Use launch_test CMake target as output file basename (#448)
- Find Python debug interpreter on Windows (#437)
- Contributors: Dirk Thomas, Michel Hidalgo, William Woodall

**launch_testing_ros**

- Use underscores in setup.cfg instead of dashes. (#227)
- Merge pull request #183 from ros2/update-maintainers
- Move Pete to author, per clalancette
- Update the package.xml files with the latest Open Robotics maintainers
- Add pytest.ini so local tests don’t display warning (#152)
- Contributors: Chris Lalancette, Michael Jeronimo, Mike Purvis

**launch_xml**

- Add frontend support for LogInfo action (#467)
- Validate unparsed attributes and subentities in launch_xml and launch_yam (#468)
- Add test for launch.actions.TimerAction (#470)
- Update package maintainers (#465)
- Use new type_utils functions (#438)
- Add pytest.ini so local tests don’t display warning (#428)
- Contributors: Chris Lalancette, Ivan Santiago Paunovic, Jacob Perron, Michel Hidalgo
launch_yaml

- Add frontend support for LogInfo action (#467)
- Validate unparsed attributes and subentities in launch_xml and launch_yaml (#468)
- Update package maintainers (#465)
- Use new type_utils functions (#438)
- Close YAML file when we’re done. (#415)
- Add pytest.ini so local tests don’t display warning (#428)
- Contributors: Chris Lalancette, Dan Rose, Ivan Santiago Paunovic, Jacob Perron, Michel Hidalgo

libcurl_vendor

- Update libcurl_vendor to the latest version (7.75.0). (#60)
- Add an override flag to force vendored build (#58)
- Update maintainers (#53)
- bump curl version to 7.68 (#47)
- Contributors: Alejandro Hernández Cordero, Chris Lalancette, Dirk Thomas, Scott K Logan

libstatistics_collector

- fix: measured values after the decimal point are truncated #79 (#80)
- Update linter to run on rolling+focal (#81)
- Add automerge.yml config file (#70)
- Update QD to QL 1 (#68)
- Updated QD (#64)
- Updated QD Performance tests (#58)
- Added benchmark test to libstatistics_collector (#57)
  Added benchmark test to libstatistics_collector * cppcheck supressed unknown macro warning - macos * Reset heap counters * Added feedback * Remove unknownMacro suppression from CMakeLists.txt * Added feedback * moved benchmark test to test/benchmark * Added feedback
  Co-authored-by: Devin Bonnie <47613035+dabonnie@users.noreply.github.com>
- Report failed workflows (#56) Allow codecov failures to be silent
- Add default CODEOWNERS file (#55)
- Remove repo activity from individual repositories in favor of centralized reporting (#52)
- Don’t attempt to report if originating from a fork (#43)
- Removed doxygen warnings (#41) Co-authored-by: Anas Abou Allaban <allabana@amazon.com>
- Add autoapprove action for dependabot (#40)
- Create Dependabot config file (#31)
  Create Dependabot config file * Randomize time of run Co-authored-by: dependabot-preview[bot] <27856297+dependabot-preview[bot]@users.noreply.github.com> Co-authored-by: Prajakta Gokhale <prajaktg@amazon.com>
- Updated QD to 3 (#30)
• Add Security Vulnerability Policy pointing to REP-2006. (#24) Co-authored-by: Emerson Knapp
  <537409+emersonknapp@users.noreply.github.com>
• Contributors: Alejandro Hernández Cordero, Chris Lalancette, Devin Bonnie, Emerson Knapp, Lucas Han, Prajikta Gokhale, Stephen Brawner, hsgwa

**libyaml_vendor**

• updating quality declaration links (re: ros2/docs.ros2.org#52) (#38)
• Update libyaml_vendor to 0.2.5. (#37)
• Fix linker flags for tests when CMake < 3.13 (#35)
• Always preserve source permissions in vendor packages (#31)
• Fix target\_link\_directories/link\_directories in cmake (#29)
• Included benchmark tests (#20)
• Update Quality Declaration (#23)
• Update package maintainers. (#22)
• Bump QD to 3 and some minor style fixes (#19)
• Add Security Vulnerability Policy pointing to REP-2006. (#18)
• Add quality declaration libyaml\_vendor (#12)
  • Contributors: Alejandro Hernández Cordero, Chris Lalancette, Jorge Perez, Michel Hidalgo, Scott K Logan, shonigmann

**lifecycle**

• Cleanup the README.rst for the lifecycle demo. (#508)
• change ParameterEventHandler to take events as const ref instead of shared pointer (#494)
• Update the package.xml files with the latest Open Robotics maintainers (#466)
• Add missing required parameter in LifecycleNode launch action (#456)
  • Contributors: Chris Lalancette, Ivan Santiago Paunovic, Michael Jeronimo, William Woodall

**lifecycle\_msgs**

• Change index.ros.org -> docs.ros.org. (#122)
• Updating Quality Declaration (#120)
• Update quality declaration to QL 1. (#116)
• Update package maintainers. (#112)
• Increase Quality level of packages to 3 (#108)
• Add Security Vulnerability Policy pointing to REP-2006. (#106)
• Updating QD to reflect package versions (#107)
  • Contributors: Chris Lalancette, Michel Hidalgo, Stephen Brawner, brawner, shonigmann
logging_demo

• Update logging macros (#476)
  • Update the package.xml files with the latest Open Robotics maintainers (#466)
  • Contributors: Audrow Nash, Michael Jeronimo

map_msgs

• update maintainers
  • Contributors: Mabel Zhang, Steve Macenski

message_filters

• Find and export dependencies properly (#54)
  • Add pytest.ini so local tests don’t display warning (#47)
  • Contributors: Chris Lalancette, Michel Hidalgo

mimick_vendor

• Always preserve source permissions in vendor packages (#19)
  • Suppress update of pinned git repository (#17)
  • Don’t overwrite -Wno-dev CMake argument (#18)
  • Add missing build tool dependency on ‘git’ (#16)
  • Update tag for armv7I support. (#15)
  • Update tag for new cmake version requirement (#14)
  • Export include directories (#13)
  • Update package maintainers (#10)
  • Suppress cppcheck for MMK_MANGLE_ (#8)
  • Change Mimick tagged version. (#7)
  • Change tag to pull latest Mimick version (#6)
  • Pin Mimick version. (#5)
  • Change imported dep to match ROS 2 fork (#4)
  • Avoid CMAKE_BUILD_TYPE warnings on Windows. (#3)
  • Remove dep tag + add maintainer(#2)
  • Configure MSVC x64 builds when appropriate. (#1)
  • First iteration vendor for Mimick library
  • Contributors: Jorge Perez, Michel Hidalgo, Scott K Logan, Stephen Brawner, brawner
Vulcanexus Documentation, Release 1.0.0

**nav_msgs**

- Change index.ros.org -> docs.ros.org. (#149)
- updating quality declaration links (re: ros2/docs.ros2.org#52) (#145)
- Update QDs to QL 1 (#135)
- Update package maintainers. (#132)
- Updated Quality Level to 2 (#131)
- Add LoadMap service (#129)
- Update Quality levels to level 3 (#124)
- Finish up API documentation (#123)
- Add Security Vulnerability Policy pointing to REP-2006. (#120)
- Contributors: Alejandro Hernández Cordero, Chris Lalancette, Michel Hidalgo, Stephen Brawner, Steve Macenski, brawner, shonigmann

**osrf_pycommon**

- Fix osrf.py_common.process_utils.get_loop() implementation (#70)
- Python 2/3 version conflict (#69)
- remove jessie because we no longer support 3.4 (#67)
- Remove deprecated use of asyncio.coroutine decorator. (#64)
- Fix the __str__ method for windows terminal_color. (#65)
- Contributors: Chris Lalancette, Jochen Sprickerhof, Michel Hidalgo, William Woodall

**osrf_testing_tools_cpp**

- [osrf_testing_tools_cpp] Add warnings (#54)
- Update cmake minimum version to 2.8.12 (#61)
- Add googletest v1.10.0 (#55)
- Workarounds for Android (#52) (#60)
- Change WIN32 to __WIN32 (#53)
- fix execinfo.h not found for QNX (#50)
- Contributors: Ahmed Sobhy, Audrow Nash, Dan Rose, Jacob Perron, Stephen Brawner

4.8. ROS 2 Documentation
**pendulum_control**

- Replace rmw_connext_cpp with rmw_connextdds (#489)
- Remove ineffective log output (#450) (#477)
- Update the package.xml files with the latest Open Robotics maintainers (#466)
- Remove deprecated warning (#459)
- Follow API/file name changes (ros2/realtine_support#94) (#451)
- Contributors: Anas Abou Allaban, Andrea Sorbini, Michael Jeronimo, y-okumura-isp

**pendulum_msgs**

- Update the package.xml files with the latest Open Robotics maintainers (#466)
- Contributors: Michael Jeronimo

**performance_test_fixture**

- Record calls to calloc, update tests (#15)
- Make allocation counter atomic (#13) Even if the benchmark itself isn’t threaded, the process we’re testing could be. In any case, this should prevent those shenanigans from messing up the measurement.
- Add methods for pausing/resuming performance metrics (#10) * Add methods for pausing/resuming performance metrics
- Add benchmarks to evaluate overhead (#11) * Add benchmarks to evaluate overhead in performance tests
- Add namespace performance_test_fixture to .cpp (#9)
- Export dependency on benchmark and osrf_testing_tools_cpp (#8)
- Update maintainers (#7)
- Expose a function for resetting the heap counters (#6)
- Stop recording memory operations sooner (#5)
- Suppress memory tools warning if tests will be skipped (#4)
- Export dependency on ament_cmake_google_benchmark (#3)
- Add missing dependency on ament_cmake_google_benchmark (#2)
- Initial ‘performance_test_fixture’ package (#1)
- Initial commit
- Contributors: Alejandro Hernández Cordero, Scott K Logan, brawner
pluginlib

- Use rcpputils for the filesystem implementation. (#212)
- Check for NULL in XML::Attribute
- Check for NULL in XML::GetText
- Check for NULL in XML::Value
- Remove unused variable output_library (#211)
- Make Chris a maintainer of pluginlib. (#210)
- Add QNX C++ fs library compiler option (#205)
- Fix cmake 3.5 compatibility (#203)
- Add function for same-package pluginlib tests (#201)
- Remove deprecated boost functions (#199)
- Contributors: Ahmed Sobhy, Chris Lalancette, Jeremie Deray, Karsten Knese, Shane Loretz

pybind11_vendor

- Update maintainers (#7)
- Merge pull request #3 from ros2/fix_windows_warning
- remove passing in CMAKE_BUILD_TYPE Signed-off-by: Mabel Zhang <mabel@openrobotics.org>
- cleanup Signed-off-by: Mabel Zhang <mabel@openrobotics.org>
- do not define CMAKE_BUILD_TYPE on windows Signed-off-by: Mabel Zhang <mabel@openrobotics.org>
- suppress all developer warnings Signed-off-by: Mabel Zhang <mabel@openrobotics.org>
- suppress warning on windows Signed-off-by: Mabel Zhang <mabel@openrobotics.org>
- attempt to fix windows warning Signed-off-by: Mabel Zhang <mabel@openrobotics.org>
- Disable building pybind11 tests (#1) Signed-off-by: Karsten Knese <karsten@openrobotics.org>
- Update to pybind 2.5.0 (#2) Signed-off-by: Mabel Zhang <mabel@openrobotics.org>
- Create pybind11 vendor package. Signed-off-by: Michael Carroll <michael@openrobotics.org>
- Contributors: Karsten Knese, Mabel Zhang, Michael Carroll, Shane Loretz

python_cmake_module

- Update maintainers (#2)
- Contributors: Shane Loretz
python_qt_binding

- Add repo README
- Shorten some long lines of CMake (#99)
- Update maintainers (#96) (#98)
- Add pytest.ini so local tests don’t display warning (#93)
- Contributors: Chris Lalancette, Scott K Logan, Shane Loretz

qt_dotgraph

- add API to set edge tooltip (#237)

qt_gui

- Always prefer ‘Tango’ icon theme (#250)
- Fix ‘dict_keys’ object not subscriptable (#243)
- allow hide title in standalone (#235)
- add logic to load qt_gui_icons on windows and macOS (#222)
- fix exporting perspective for Python 3.6 (#228)
- remove tango-icon-theme dependency (#224)
- Contributors: Michael Jeronimo, Scott K Logan

qt_gui_cpp

- Fix duplicated QMap to QMultiMap (#244)
- Switch to using the filesystem implementation in rcputils. (#239)
- avoid a warning about C++ plugins on Windows (#232)
- qt_gui_cpp_sip: declare private assignment operator for SIP (#226)
- Contributors: Chris Lalancette, Homalozoa X

quality_of_service_demo_cpp

- Add demo of how to use qos overrides (#474)
- Update the package.xml files with the latest Open Robotics maintainers (#466)
- Update comments in quality_of_service_demo_cpp message_lost_talker and message_lost_listener (#458)
- Add message lost status event demo using rclcpp (#453)
- Contributors: Ivan Santiago Paunovic, Michael Jeronimo
quality_of_service_demo_py

- Use underscores instead of dashes in setup.cfg (#502)
- QoS overrides demo in python (#479)
- Update deprecated qos policy value names (#468)
- Update the package.xml files with the latest Open Robotics maintainers (#466)
- Add rclpy message lost status event demo (#457)
- Contributors: Ivan Santiago Paunovic, Michael Jeronimo

rcl

- Fix up test_network_flow_endpoints. (#912)
- Make test_two_timers_ready_before_timeout less flaky (#911)
- Add publishing instrumentation (#905)
- Unique network flows (#880)
- updating quality declaration links (re: ros2/docs.ros2.org#52) (#909)
- Add functions for waiting for publishers and subscribers (#907)
- Revert “Mark cyclonedds test_service test as flakey (#648)” (#904)
- Guard against returning NULL or empty node names (#570)
- Remove exceptions for rmw_connext_cpp tests. (#903)
- Add support for rmw_connextdds (#895)
- Put an argument list of ‘void’ where no arguments are expected. (#899)
- Cleanup documentation for doxygen. (#896)
- Reference test resources directly from source tree (#554)
- Re-add “Improve trigger test for graph guard condition (#811)” (#884)
- Revert “Improve trigger test for graph guard condition (#811)” (#883)
- Move the guard condition cleanup after removing callback. (#877)
- Make test_subscription_nominal_string_sequence more reliable (#881)
- Improve trigger test for graph guard condition (#811)
- Add NULL check in remap.c (#879)
- Add const to constant rcl_context functions (#872)
- Fix another failing test on CentOS 7 (#863)
- Update QDs to QL 1 (#866)
- Address clang static analysis issues (#865)
- Fix flaky test_info_by_topic (#859)
- Update QL (#858)
- Refactor for removing unnecessary source code (#857)
• Clarify storing of current_time (#850)
• Make tests in test_graph.cpp more reliable (#854)
• Fix for external log segfault after SIGINT (#844)
• Update tracetools QL and add to rcl_lifecycle's QD (#845)
• Make test logging rosout more reliable (#846)
• Return OK when finalizing zero-initialized contexts (#842)
• Zero initialize events an size_of_events members of rcl_wait_set_t (#841)
• Update deprecated gtest macros (#818)
• Make sure to check the return value of rcl APIs. (#838)
• Add convenient node method to get a final topic/service name (#835)
• Remove redundant error formatting (#834)
• Fix memory leak in rcl_subscription_init()/rcl_publisher_init() (#794)
• Update maintainers (#825)
• Add a semicolon to RCUTILS_LOGGING_AUTOINIT. (#816)
• Improve error messages in rcl_lifecycle (#742)
• Fix memory leak on serialized message in test_publisher/subscription.cpp (#801)
• Fix memory leak because of mock test (#800)
• Spelling correction (#798)
• Fix that not to deallocate event impl in some failure case (#790)
• calling fini functions to avoid memory leak (#791)
• Bump rcl arguments’ API test coverage (#777)
• Fix rcl arguments’ API memory leaks and bugs (#778)
• Add coverage tests wait module (#769)
• Fix wait set allocation cleanup (#770)
• Improve test coverage in rcl (#764)
• Check if rcutils_strdup() outcome immediately (#768)
• Cleanup rcl_get_secure_root() implementation (#762)
• Add fault injection macros to rcl functions (#727)
• Yield rcl_context_fini() error codes (#763)
• Do not invalidate context before successful shutdown (#761)
• Zero initialize guard condition on failed init (#760)
• Adding tests to arguments API (#752)
• Extend rcl_expand_topic_name() API test coverage (#758)
• Add coverage tests 94% to service API (#756)
• Clean up rcl_expand_topic_name() implementation (#757)
• Complete rcl enclave validation API coverage (#751)
• Cope with base function restrictions in mocks (#753)
• Fix allocation when copying arguments (#748)
• Complete rcl package’s logging API test coverage (#747)
• Improve coverage to 95% in domain id, init option, rmw implementation id and log level modules (#744)
• Fix rcl package’s logging API error code documentation and handling (#746)
• Fix bug error handling in get_param_files (#743)
• Complete subscription API test coverage (#734)
• increase timeouts in test_services fixtures for Connext (#745)
• Tweaks to client.c and subscription.c for cleaner init/fini (#728)
• Improve error checking and handling in subscription APIs (#739)
• Add deallocate calls to free strdup allocated memory (#737)
• Add missing calls to rcl_convert_rmw_ret_to_rcl_ret (#738)
• Add mock tests, publisher 95% coverage (#732)
• Restore env variables set in the test_failing_configuration. (#733)
• Expose qos setting for /rosout (#722)
• Reformat rmw_impl_id_check to call a testable function (#725)
• Add extra check for invalid event implementation (#726)
• Consolidate macro duplication (#653)
• Add test for subscription message lost event (#705)
• Add function rcl_event_is_valid (#720)
• Move actual domain id from node to context (#718)
• Removed doxygen warnings (#712)
• Remove some dead code.
• Make sure to call rcl_arguments_fini at the end of the test.
• Add remap needed null check (#711)
• Make public init/fini rosout publisher (#704)
• Move rcl_remap_copy to public header (#709)
• Implement a generic way to change logging levels (#664)
• Remove domain_id and localhost_only from node_options (#708)
• Add coverage tests (#703)
• Add bad arguments tests for coverage (#698)
• Remove unused internal prototypes (#699)
• Update quality declaration and coverage (#674)
• Add setter and getter for domain_id in rcl_init_options_t (#678)
• Remove unused pytest dependency from rcl. (#695)
• Fix link to latest API docs (#692)
• Keep domain id if ROS_DOMAIN_ID is invalid. (#689)
• Remove unused check context.c (#691)
• Add check rcl_node_options_copy invalid out (#671)
• Update tracetools’ QL to 2 in rcl’s QD (#690)
• Improve subscription coverage (#681)
• Improve rcl timer test coverage (#680)
• Improve wait sets test coverage (#683)
• Improve rcl init test coverage. (#684)
• Improve clock test coverage. (#685)
• Add message lost event (#673)
• Minor fixes to rcl clock implementation. (#688)
• Improve enclave validation test coverage. (#682)
• Use RCL_RET_* codes only. (#686)
• Fixed doxygen warnings (#677)
• Add tests for rcl package (#668)
• Remove logging_external_interface.h, provided by rcl_logging_interface package now (#676)
• Print RCL_LOCALHOST_ENV_VAR if error happens via rcutils_get_env. (#672)
• Contributors: Ada-King, Alejandro Hernández Cordero, Ananya Muddukrishna, Andrea Sorbini, Audrow Nash, Barry Xu, Chen Lihui, Chris Lalancette, Christophe Bedard, Dan Rose, Dirk Thomas, Geoffrey Biggs, Ivan Santiago Puunovic, Jacob Perron, Jorge Perez, Lei Liu, Michel Hidalgo, Nikolai Morin, Scott K Logan, Stephen Brawner, Thijs Raymakers, brawner, shonigmann, tomoya

rcl_action

• updating quality declaration links (re: ros2/docs.ros2.org#52) (#909)
• Don’t expect RCL_RET_TIMEOUT to set an error string (#900)
• Add support for rmw_connextdds (#895)
• Avoid setting error message twice. (#887)
• Address various clang static analysis fixes (#864)
• Update QDs to QL 1 (#866)
• Update QL (#858)
• Make sure to always check return values (#840)
• Update deprecated gtest macros (#818)
• Make sure to check the return value of rcl APIs. (#838)
• Update maintainers (#825)
• Store reference to rcl_clock_t instead of copy (#797)
• Use valid clock in case of issue in rcl_timer_init (#795)
• Add fault injection macros and unit tests to rcl_action (#730)
• Change some EXPECT_EQ to ASSERT_EQ in test_action_server. (#759)
• Removed doxygen warnings (#712)
• Address issue 716 by zero initializing pointers and freeing memory (#717)
• Update quality declaration and coverage (#674)
• Fixed doxygen warnings (#677)
• Contributors: Alejandro Hernández Cordero, Andrea Sorbini, Audrow Nash, Chen Lihui, Chris Lalancette, Ivan Santiago Paunovic, Shane Loretz, Stephen Brawner, brawner, shonigmann

**rcl_interfaces**

• Change index.ros.org -> docs.ros.org. (#122)
• Updating Quality Declaration (#120)
• Add field to the parameter description to specify dynamic/static typing. (#118)
• Update quality declaration to QL 1. (#116)
• Update package maintainers. (#112)
• Increase Quality level of packages to 3 (#108)
• Add Security Vulnerability Policy pointing to REP-2006. (#106)
• Updating QD to reflect package versions (#107)
• Contributors: Chris Lalancette, Ivan Santiago Paunovic, Michel Hidalgo, Stephen Brawner, brawner, shonigmann

**rcl_lifecycle**

• updating quality declaration links (re: ros2/docs.ros2.org#52) (#909)
• make rcl_lifecycle_com_interface optional in lifecycle nodes (#882)
• Update QDs to QL 1 (#866)
• Update QL (#858)
• Make sure to always check return values (#840)
• Update tracetools QL and add to rcl_lifecycle’s QD (#845)
• Add compiler warnings (#830)
• Make sure to check the return value of rcl APIs. (#838)
• Add lifecycle node state transition instrumentation (#804)
• Update maintainers (#825)
• Improve error messages in rcl_lifecycle (#742)
• Fix test_rcl_lifecycle (#788)
• Add fault injection macros and unit tests to rcl_lifecycle (#731)
• Remove std::cout line from test_rcl_lifecycle.cpp (#773)
• Set transition_map->states/transition size to 0 on fini (#729)

4.8. ROS 2 Documentation
• Topic fix rcl lifecycle test issue (#715)
• Removed doxygen warnings (#712)
• Update quality declaration and coverage (#674)
• Contributors: Alejandro Hernández Cordero, Audrow Nash, Barry Xu, Chris Lalancette, Christophe Bedard, Ivan Santiago Paunovic, Karsten Knese, Lei Liu, Stephen Brawner, brawner, shonigmann

rcl_logging_interface

• Update QD to QL 1 (#66)
• Use rcutils_expand_user in rcl_logging_get_logging_directory (#59)
• Allow configuring logging directory through environment variables (#53)
• Update the maintainers. (#55)
• Add new package with rcl logging interface (#41)
• Contributors: Chris Lalancette, Christophe Bedard, Stephen Brawner

rcl_logging_log4cxx

• Allow configuring logging directory through environment variables (#53)
• Update the maintainers. (#55)
• Remove unused pytest dependency. (#43)
• Use new package with rcl logging interface (#41)
• Contributors: Chris Lalancette, Christophe Bedard

rcl_logging_noop

• Make internal dependencies private (#60)
• Update the maintainers. (#55)
• Remove unused pytest dependency. (#43)
• Use new package with rcl logging interface (#41)
• Contributors: Chris Lalancette, Shane Loretz

rcl_logging_spdlog

• updating quality declaration links (re: ros2/docs.ros2.org#52) (#73)
• Include what you use (#71)
• Update QD to QL 1 (#66)
• Make sure to check return value from external_initialize. (#65)
• updated QD section 3.i and 3.ii and spelling error (#63)
• rcl_logging_spdlog: Increased QL to 2 in QD
• Updated spdlog QL in QD
• Make internal dependencies private (#60)
• [rcllogging_spdlog] Add warnings (#54)
• Allow configuring logging directory through environment variables (#53)
• Update the maintainers. (#55)
• Added benchmark test to rcllogging_spdlog (#52)
• Used current_path() function from rcpputils (#51)
• Add fault injection unittest to increase coverage (#49)
• Bump QD to level 3 and updated QD (#44)
• Added Doxygen file and fixed related warnings (#42)
• Use new package with rcl logging interface (#41)
• Increased test coverage (#40)
• Add Security Vulnerability Policy pointing to REP-2006.
• Rename Quality_Declaration.md -> QUALITY_DECLARATION.md
• Contributors: Alejandro Hernández Cordero, Audrow Nash, Chris Lalancette, Christophe Bedard, Ivan Santiago Paunovic, Scott K Logan, Shane Loretz, Stephen Brawner, ahcorde, brawner, shonigmann

**rclyamlparamparser**

• updating quality declaration links (re: ros2/docs.ros2.org#52) (#909)
• Enable compiler warnings (#831)
• Update QDs to QL 1 (#866)
• Rearrange test logic to avoid reference to null (#862)
• Update QL (#858)
• Make sure to initialize the end_mark for yaml_event_t (#849)
• Check for valid node names in parameters files (#809)
• Update maintainers (#825)
• Updated performance section QD (#817)
• Several memory-related fixes for rclvariant_t benchmarks (#813)
• Improved rclyamlparamparser benchmark test (#810)
• Added benchmark test to rclyamlparamparser (#803)
• Remove MAX_NUM_PARAMS_PER_NODE and MAX_NUM_NODE_ENTRIES limitation. (#802)
• Add mocking unit tests for rclyamlparamparser (coverage part 3/3) (#772)
• Add fault-injection unit tests (coverage part 2/3) (#766)
• Add basic unit tests for refactored functions in rclyamlparamparser (coverage part 1/3) (#771)
• Fix yaml parser error when meets .nan (refactor on #754) (#781)
• Refactor parser.c for better testability (#754)

4.8. ROS 2 Documentation 389
• Don’t overwrite cur_ns pointer if reallocation fails (#780)
• Fix mem leaks in unit test from 776 (#779)
• Fix rcl_parse_yaml_file() error handling. (#776)
• Don’t overwrite string_array pointer on reallocation failure (#775)
• Set yaml_variant values to NULL on finalization (#765)
• Remove debugging statements. (#755)
• Removed doxygen warnings (#712)
• Update quality declaration and coverage (#674)
• Contributors: Alejandro Hernández Cordero, Audrow Nash, Chen Lihui, Chris Lalancette, Ivan Santiago Paunovic, Michel Hidalgo, Scott K Logan, Stephen Brawner, brawner, shonigmann, tomoya

rclcpp
• Use OnShutdown callback handle instead of OnShutdown callback (#1639) (#1650)
• use dynamic_pointer_cast to detect allocator mismatch in intra process manager (#1643) (#1644)
• Increase cppcheck timeout to 500s (#1634)
• Clarify node parameters docs (#1631)
• Avoid returning loan when none was obtained. (#1629)
• Use a different implementation of mutex two priorities (#1628)
• Do not test the value of the history policy when testing the get_publishers/subscriptions_info_by_topic() methods (#1626)
• Check first parameter type and range before calling the user validation callbacks (#1627)
• Restore test exception for Connext (#1625)
• Fix race condition in TimeSource clock thread setup (#1623)
• remove deprecated code which was deprecated in foxy and should be removed in galactic (#1622)
• Change index.ros.org -> docs.ros.org. (#1620)
• Unique network flows (#1496)
• Add spin_some support to the StaticSingleThreadedExecutor (#1338)
• Add publishing instrumentation (#1600)
• Create load_parameters and delete_parameters methods (#1596)
• refactor AnySubscriptionCallback and add/deprecate callback signatures (#1598)
• Add generic publisher and generic subscription for serialized messages (#1452)
• use context from node_base\_ for clock executor. (#1617)
• updating quality declaration links (re: ros2/docs.ros2.org#52) (#1615)
• Initialize integers in test_parameter_event_handler.cpp to avoid undefined behavior (#1609)
• Namespace tracetools C++ functions (#1608)
• Revert “Namespace tracetools C++ functions (#1603)” (#1607)
• Namespace tracetools C++ functions (#1603)
• Clock subscription callback group spins in its own thread (#1556)
• Remove rmw_connext_cpp references. (#1595)
• Add API for checking QoS profile compatibility (#1554)
• Document misuse of parameters callback (#1590)
• use const auto & to iterate over parameters (#1593)
• Guard against integer overflow in duration conversion (#1584)
• get_parameters service should return empty if undeclared parameters are allowed (#1514)
• Made ‘Context::shutdown_reason’ function a const function (#1578)
• Document design decisions that were made for statically typed parameters (#1568)
• Fix doc typo in CallbackGroup constructor (#1582)
• Enable qos parameter overrides for the /parameter_events topic (#1532)
• Add support for rmw_connextdds (#1574)
• Remove ‘struct’ from the rcl_time_jump_t. (#1577)
• Add tests for declaring statically typed parameters when undeclared parameters are allowed (#1575)
• Quiet clang memory leak warning on “DoNotOptimize”. (#1571)
• Add ParameterEventsSubscriber class (#829)
• When a parameter change is rejected, the parameters map shouldn’t be updated. (#1567)
• Fix when to throw the NoParameterOverrideProvided exception. (#1567)
• Fix SEGV caused by order of destruction of Node sub-interfaces (#1469)
• Fix benchmark test failure introduced in #1522 (#1564)
• Fix documented example in create_publisher (#1558)
• Enforce static parameter types (#1522)
• Allow timers to keep up the intended rate in MultiThreadedExecutor (#1516)
• Fix UBSAN warnings in any_subscription_callback. (#1551)
• Fix runtime error: reference binding to null pointer of type (#1547)
• Reference test resources directly from source tree (#1543)
• clear statistics after window reset (#1531) (#1535)
• Fix a minor string error in the topic_statistics test. (#1541)
• Avoid Resource deadlock avoided if use intra_process_comms (#1530)
• Avoid an object copy in parameter_value.cpp. (#1538)
• Assert that the publisher_list size is 1. (#1537)
• Don’t access objects after they have been std::move (#1536)
• Update for checking correct variable (#1534)
• Destroy msg extracted from LoanedMessage. (#1305)
• Add instrumentation for linking a timer to a node (#1500)
• Fix error when using IPC with StaticSingleThreadExecutor (#1520)
• Change to using unique_ptrs for DummyExecutor. (#1517)
• Allow reconfiguring ‘clock’ topic qos (#1512)
• Allow to add/remove nodes thread safely in rclcpp::Executor (#1505)
• Call rclcpp::shutdown in test_node for clean shutdown on Windows (#1515)
• Reapply “Add get_logging_directory method to rclcpp::Logger (#1509)” (#1513)
• use describe_parameters of parameter client for test (#1499)
• Revert “Add get_logging_directory method to rclcpp::Logger (#1509)” (#1511)
• Add get_logging_directory method to rclcpp::Logger (#1509)
• Better documentation for the QoS class (#1508)
• Modify excluding callback duration from topic statistics (#1492)
• Make the test of graph users more robust. (#1504)
• Make sure to wait for graph change events in test_node_graph. (#1503)
• add timeout to SyncParametersClient methods (#1493)
• Fix wrong test expectations (#1497)
• Update create_publisher/subscription documentation, clarifying when a parameters interface is required (#1494)
• Fix string literal warnings (#1442)
• support describe_parameters methods to parameter client. (#1453)
• Add getters to rclcpp::qos and rclcpp::Policy enum classes (#1467)
• Change nullptr checks to use ASSERT_TRUE. (#1486)
• Adjust logic around finding and erasing guard_condition (#1474)
• Update QDs to QL 1 (#1477)
• Add performance tests for parameter transport (#1463)
• Move ownership of shutdown_guard_condition to executors/graph_listener (#1404)
• Add options to automatically declare qos parameters when creating a publisher/subscription (#1465)
• Add take_data to Waitable and data to AnyExecutable (#1241)
• Add benchmarks for node parameters interface (#1444)
• Remove allocation from executor::remove_node() (#1448)
• Fix test crashes on CentOS 7 (#1449)
• Bump rclcpp packages to Quality Level 2 (#1445)
• Added executor benchmark tests (#1413)
• Add fully-qualified namespace to WeakCallbackGroupsToNodesMap (#1435)
• Deprecate Duration(rcl_duration_value_t) in favor of static Duration::from_nanoseconds(rcl_duration_value_t) (#1432)
• Avoid parsing arguments twice in rclcpp::init_and_remove_ros_arguments (#1415)
• Add service and client benchmarks (#1425)
• Set CMakeLists to only use default rmw for benchmarks (#1427)
• Update tracetools’ QL in rclcpp’s QD (#1428)
• Add missing locking to the rclcpp_action::ServerBase. (#1421)
• Initial benchmark tests for rclcpp::init/shutdown create/destroy node (#1411)
• Refactor test CMakeLists in prep for benchmarks (#1422)
• Add methods in topic and service interface to resolve a name (#1410)
• Update deprecated gtest macros (#1370)
• Clear members for StaticExecutorEntitiesCollector to avoid shared_ptr dependency (#1303)
• Increase test timeouts of slow running tests with rmw_connext_cpp (#1400)
• Avoid self dependency that not destoryed (#1301)
• Update maintainers (#1384)
• Add clock qos to node options (#1375)
• Fix NodeOptions copy constructor (#1376)
• Make sure to clean the external client/service handle. (#1296)
• Increase coverage of WaitSetTemplate (#1368)
• Increase coverage of guard_condition.cpp to 100% (#1369)
• Add coverage statement (#1367)
• Tests for LoanedMessage with mocked loaned message publisher (#1366)
• Add unit tests for qos and qos_event files (#1352)
• Finish coverage of publisher API (#1365)
• Finish API coverage on executors. (#1364)
• Add test for ParameterService (#1355)
• Add time API coverage tests (#1347)
• Add timer coverage tests (#1363)
• Add in additional tests for parameter_client.cpp coverage.
• Minor fixes to the parameter_service.cpp file.
• reset rcl_context shared_ptr after calling rcl_init sucessfully (#1357)
• Improved test publisher - zero qos history depth value exception (#1360)
• Covered resolve_use_intra_process (#1359)
• Improve test_subscription_options (#1358)
• Add in more tests for init_options coverage. (#1353)
• Test the remaining node public API (#1342)
• Complete coverage of Parameter and ParameterValue API (#1344)
• Add in more tests for the utilities. (#1349)
• Add in two more tests for expand_topic_or_service_name. (#1350)
• Add tests for node_options API (#1343)
• Add in more coverage for expand_topic_or_service_name. (#1346)
• Test exception in spin_until_future_complete. (#1345)
• Add coverage tests graph_listener (#1330)
• Add in unit tests for the Executor class.
• Allow mimick patching of methods with up to 9 arguments.
• Improve the error messages in the Executor class.
• Add coverage for client API (#1329)
• Increase service coverage (#1332)
• Make more of the static entity collector API private.
• Const-ify more of the static executor.
• Add more tests for the static single threaded executor.
• Many more tests for the static_executor_entities_collector.
• Get one more line of code coverage in memory_strategy.cpp
• Bugfix when adding callback group.
• Fix typos in comments.
• Remove deprecated executor::FutureReturnCode APIs. (#1327)
• Increase coverage of publisher/subscription API (#1325)
• Not finalize guard condition while destructing SubscriptionIntraProcess (#1307)
• Expose qos setting for /rosout (#1247)
• Add coverage for missing API (except executors) (#1326)
• Include topic name in QoS mismatch warning messages (#1286)
• Add coverage tests context functions (#1321)
• Increase coverage of node_interfaces, including with mocking rcl errors (#1322)
• Make node_graph::count_graph_users() const (#1320)
• Add coverage for wait_set_policies (#1316)
• Only exchange intra_process waitable if nonnull (#1317)
• Check waitable for nullptr during constructor (#1315)
• Call vector.erase with end iterator overload (#1314)
• Use best effort, keep last, history depth 1 QoS Profile for ‘/clock’ subscriptions (#1312)
• Add tests type_support module (#1308)
• Replace std_msgs with test_msgs in executors test (#1310)
• Add set_level for rclcpp::Logger (#1284)
• Remove unused private function (rclcpp::Node and rclcpp_lifecycle::Node) (#1294)
• Adding tests basic getters (#1291)
• Adding callback groups in executor (#1218)
• Refactor Subscription Topic Statistics Tests (#1281)
- Add operator!= for duration (#1236)
- Fix clock thread issue (#1266) (#1267)
- Fix topic stats test, wait for more messages, only check the ones with samples (#1274)
- Add get_domain_id method to rclcpp::Context (#1271)
- Fixes for unit tests that fail under cyclonedds (#1270)
- initialize_logging_ should be copied (#1272)
- Use static_cast instead of C-style cast for instrumentation (#1263)
- Make parameter clients use template constructors (#1249)
- Ability to configure domain_id via InitOptions. (#1165)
- Simplify and fix allocator memory strategy unit test for connext (#1252)
- Use global namespace for parameter events subscription topic (#1257)
- Increase timeouts for connext for long tests (#1253)
- Adjust test_static_executor_entities_collector for rmw_connext_cpp (#1251)
- Fix failing test with Connext since it doesn’t wait for discovery (#1246)
- Fix node graph test with Connext and CycloneDDS returning actual data (#1245)
- Warn about unused result of add_on_set_parameters_callback (#1238)
- Unittests for memory strategy files, except allocator_memory_strategy (#1189)
- EXPECT_THROW_EQ and ASSERT_THROW_EQ macros for unittests (#1232)
- Add unit test for static_executor_entities_collector (#1221)
- Parameterize test executors for all executor types (#1222)
- Unit tests for allocator_memory_strategy.cpp part 2 (#1198)
- Unit tests for allocator_memory_strategy.hpp (#1197)
- Derive and throw exception in spin_some spin_all for StaticSingleThreadedExecutor (#1220)
- Make ring buffer thread-safe (#1213)
- Add missing RCLCPP_PUBLIC to ~StaticExecutorEntitiesCollector (#1227)
- Document graph functions don’t apply remap rules (#1225)
- Remove recreation of entities_collector (#1217)
- Fix rclcpp::NodeOptions::operator= (#1211)
- Link against thread library where necessary (#1210)
- Unit tests for node interfaces (#1202)
- Remove usage of domain id in node options (#1205)
- Remove deprecated set_on_parameters_set_callback function (#1199)
- Fix conversion of negative durations to messages (#1188)
- Fix implementation of NodeOptions::use_global_arguments() (#1176)
- Bump to QD to level 3 and fixed links (#1158)
- Fix pub/sub count API tests (#1203)
- Update trace tools’ QL to 2 in rclcpp’s QD (#1187)
- Fix exception message on rcl_clock_init (#1182)
- Throw exception if rcl_timer_init fails (#1179)
- Unit tests for some header-only functions/classes (#1181)
- Callback should be perfectly-forwarded (#1183)
- Add unit tests for logging functionality (#1184)
- Add create_publisher include to create_subscription (#1180)
- Check period duration in create_wall_timer (#1178)
- Fix get_node_time_source_interface() docstring (#988)
- Add message lost subscription event (#1164)
- Add spin_all method to Executor (#1156)
- Reorganize test directory and split CMakeLists.txt (#1173)
- Check if context is valid when looping in spin_some (#1167)
- Add check for invalid topic statistics publish period (#1151)
- Fix spin_until_future_complete: check spinning value (#1023)
- Fix doxygen warnings (#1163)
- Fix reference to rclcpp in its Quality declaration (#1161)
- Allow spin_until_future_complete to accept any future like object (#1113)

**rclcpp_action**

- Returns CancelResponse::REJECT while goal handle failed to transit to CANCELING state (#1641) (#1653)
- Fix action server deadlock issue that caused by other mutexes locked in CancelCallback (#1635) (#1646)
- updating quality declaration links (re: ros2/docs.ros2.org#52) (#1615)
- Add support for rmw_connextdds (#1574)
- node_handle must be destroyed after client_handle to prevent memory leak (#1562)
- Finalize rcl_handle to prevent leak (#1528) (#1529)
- Fix #1526. (#1527)
- Fix action server deadlock (#1285) (#1313)
- Goal response callback compatibility shim with deprecation of old signature (#1495)
- [rclcpp_action] Add warnings (#1405)
- Update QDs to QL 1 (#1477)
• Add `take_data` to `Waitable` and `data` to `AnyExecutable` (#1241)
• Fix test crashes on CentOS 7 (#1449)
• Bump `rclcpp` packages to Quality Level 2 (#1445)
• Add `rclcpp_action::action_server` benchmarks (#1433)
• Benchmark `rclcpp_action::action_client` (#1429)
• Add missing locking to the `rclcpp_action::ServerBase`. (#1421)
• Increase test timeouts of slow running tests with `rmw_connext_cpp` (#1400)
• Update maintainers (#1384)
• Increase coverage `rclcpp_action` to 95% (#1290)
• Pass goal handle to goal response callback instead of a future (#1311)
• Remove deprecated client goal handle method for getting result (#1309)
• Increase test timeout necessary for `Connext` (#1256)
• Bump to QD to level 3 and fixed links (#1158)
• Add `rcl_action_client_options` when creating action client. (#1133)
• Fix doxygen warnings (#1163)
• Increase `rclcpp_action` test coverage (#1153)
• Contributors: Alejandro Hernández Cordero, Andrea Sorbini, Audrow Nash, Chris Lalancette, Daisuke Sato, Dirk Thomas, Ivan Santiago Paunovic, Jacob Perron, Kaven Yau, Louise Poubel, Michel Hidalgo, Stephen Brawner, Tomoya Fujita, William Woodall, brawner, shonigmann, tomoya, y-okumura-isp

**rclcpp_components**

• updating quality declaration links (re: ros2/docs.ros2.org#52) (#1615)
• Use std compliant non-method std::filesystem::exists function (#1502)
• Fix string literal warnings (#1442)
• Update QDs to QL 1 (#1477)
• Add benchmarks for components (#1476)
• Bump `rclcpp` packages to Quality Level 2 (#1445)
• Update maintainers (#1384)
• `ComponentManager`: switch off parameter services and event publisher (#1333)
• Bump to QD to level 3 and fixed links (#1158)
• Include original exception in `ComponentManagerException` (#1157)
• Contributors: Alejandro Hernández Cordero, Audrow Nash, Ivan Santiago Paunovic, Josh Langsfeld, Louise Poubel, Martijn Buiks, Scott K Logan, Stephen Brawner, Tomoya Fujita, shonigmann
rclcpp_lifecycle

- Add generic publisher and generic subscription for serialized messages (#1452)
- updating quality declaration links (re: ros2/docs.ros2.org#52) (#1615)
- Fix flaky lifecycle node tests (#1606)
- Clock subscription callback group spins in its own thread (#1556)
- Delete debug messages (#1602)
- add automatically_add_executor_with_node option (#1594)
- make rcl_lifecycle_com_interface optional in lifecycle nodes (#1507)
- Add support for rmw_connextdds (#1574)
- Fix SEGV caused by order of destruction of Node sub-interfaces (#1469)
- Enforce static parameter types (#1522)
- add LifecycleNode::get_transition_graph to match services. (#1472)
- Update QDs to QL 1 (#1477)
- Benchmark lifecycle features (#1462)
- Reserve vector capacities and use emplace_back for constructing vectors (#1464)
- [rclcpp_lifecycle] Change uint8_t iterator variables to size_t (#1461)
- Bump rclcpp packages to Quality Level 2 (#1445)
- Increase test timeouts of slow running tests with rmw_connext_cpp (#1400)
- Update maintainers (#1384)
- Add clock qos to node options (#1375)
- Increase test coverage of rclcpp_lifecycle to 96% (#1298)
- Log error instead of throwing exception in Transition and State reset(), mark no except (#1297)
- Remove unused private function (rclcpp::Node and rclcpp_lifecycle::Node) (#1294)
- Remove rmw-dependent unit-test checks (#1293)
- Added missing tests for rclcpp lifecycle (#1240)
- Warn about unused result of add_on_set_parameters_callback (#1238)
- Remove deprecated set_on_parameters_set_callback function (#1199)
- Bump to QD to level 3 and fixed links (#1158)
- Fix race in test_lifecycle_service_client (#1204)
- Fix doxygen warnings (#1163)
**rclpy**

- Break log function execution ASAP if configured severity is too high (#776) (#783)
- typo fix. (#768)
- Restore exceptions for Connext and message timestamps on Windows (#765)
- Use correct type when creating test publisher (#764)
- Add a test for destroy_node while spinning (#663)
- Add __enter__ and __exit__ to Waitable (#761)
- Check if shutdown callback weak method is valid before calling it (#754)
- Change index.ros.org -> docs.ros.org. (#755)
- Use py::class_ for rcl_event_t (#750)
- Convert Clock to use a C++ Class (#749)
- Convert Service to use C++ Class (#747)
- Fix windows warning by using consistent types (#753)
- Use py::class_ for rmw_service_info_t and rmw_request_id_t (#748)
- Convert Timer to use a C++ Class (#745)
- Add PythonAllocator (#746)
- Use py::class_ for rmw_qos_profile_t (#741)
- Combine pybind11 modules into one (#743)
- Use py::class_ for rcl_duration_t (#744)
- Fix bug in unique_ptr type argument (#742)
- Convert Client to use C++ Class (#739)
- Converting last of _rclpy.c to pybind11 (#738)
- Make sure only non-empty std::vector of arguments are indexed (#740)
- Use py::class_ for rcl_time_point_t (#737)
- Convert logging mutex functions to pybind11 (#735)
- Document misuse of of parameter callbacks (#734)
- Convert QoS APIs to pybind11 (#736)
- Add API for checking QoS profile compatibility (#708)
- Replace rmw_connext_cpp with rmw_connextdds (#698)
- Convert last of pub/sub getters to pybind11 (#733)
- Pybind 11: count_subscribers and count_publishers (#732)
- Convert more node accessors to pybind11 (#730)
- Pybind11-ify rclpy_get_node_parameters (#718)
- Modify parameter service behavior when allow_undeclared_parameters is false and the requested parameter doesn’t exist (#661)
- Include pybind11 first to fix windows debug warning (#731)
• Convert init/shutdown to pybind11 (#715)
• Convert take API to pybind11 (#721)
• Migrate qos event APIs to pybind11 (#723)
• Remove pybind11 from rclpy common (#727)
• Look up pybind11 package once (#726)
• typo fix. (#729)
• [pybind11] Node Accessors (#719)
• Convert serialize/deserialize to pybind11 (#712)
• Convert names_and_types graph APIs to pybind11 (#717)
• Use Pybind11 for name functions (#709)
• Better checks for valid msg and srv types (#714)
• Convert duration to pybind11 (#716)
• Convert wait_set functions to pybind11 (#706)
• Explicitly populate tuple with None (#711)
• Change the time jump time type to just rcl_time_jump_t. (#707)
• Convert rclpy service functions to pybind11 (#703)
• Bump the cppcheck timeout by 2 minutes (#705)
• Convert subscription functions to pybind11 (#696)
• Convert rclpy client functions to pybind11 (#701)
• Fix static typing when allow undeclared (#702)
• Convert publisher functions to pybind11 (#695)
• Convert clock and time functions to pybind11 (#699)
• Set destructor on QoS Profile struct (#700)
• Convert timer functions to pybind11 (#693)
• Convert guard conditions functions to pybind11 (#692)
• Convert service info functions to pybind11 (#694)
• Enforce static parameter types when dynamic typing is not specified (#683)
• rclpy_ok and rclpy_create_context to pybind11 (#691)
• Include Pybind11 before Python.h (#690)
• Clean up exceptions in _rclpy_action (#685)
• Clean windows flags on _rclpy_pybind11 and _rclpy_action (#688)
• Use pybind11 for _rclpy_handle (#668)
• Split rclpy module for easier porting to pybind11 (#675)
• Use Pybind11 to generate _rclpy_logging (#659)
• Copy windows debug fixes for pybind11 (#681)
• Use pybind11 for _rclpy_action (#678)
• Update just pycapsule lib to use pybind11 (#652)
• remove maintainer (#682)
• Use Pybind11’s CMake code (#667)
• Don’t call destroy_node while spinning (#674)
• Check the rcl_action return value on cleanup. (#672)
• Fix the NULL check for destroy_ros_message. (#677)
• Use Py_XDECREF for pynode_names_and_namespaces (#673)
• Use Py_XDECREF for pyresult_list. (#670)
• Fix dead stores. (#669)
• Fix two clang static analysis warnings. (#664)
• Add method to get the current logging directory (#657)
• Fix docstring indent error in create_node (#655)
• use only True to avoid confusion in autodoc config
• document QoS profile constants
• Merge pull request #649 from ros2/clalancette/dont-except-while-sleep
• Fixes from review/CI.
• Make sure to catch the ROSInterruptException when calling rate.sleep.
• memory leak (#643) (#645)
• Don’t throw an exception if timer canceled while sleeping.
• Wake executor in Node.create_subscription() (#647)
• Fix Enum not being comparable with ints in get_parameter_types service
• Qos configurability (#635)
• Use Py_XDECREF for pytopic_names_and_types. (#638)
• qos_policy_name_from_kind() should accept either a QoS Policy Kind or an int (#637)
• Add method in Node to resolve a topic or service name (#636)
• Deprecate verbose qos policy value names (#634)
• Remove deprecated set_parameters_callback (#633)
• Make sure to use Py_XDECREF in rclpy_get_service_names_and_types (#632)
• Update maintainers (#627)
• Add in semicolon on RCUTILS_LOGGING_AUTOINIT. (#624)
• Add in the topic name when QoS events are fired. (#621)
• Use best effort, keep last, history depth 1 QoS Profile for ‘/clock’ subscriptions (#619)
• PARAM_REL_TOL documentation fix (#559)
• Node get fully qualified name (#598)
• MultiThreadedExecutor spin_until_future_complete should not continue waiting when the future is done (#605)
• skip test relying on source timestamps with Connext (#615)
• Use the rpyutils shared import_c_library function. (#610)
• Add ability to configure domain ID (#596)
• Use absolute parameter events topic name (#612)
• Destroy event handlers owned by publishers/subscriptions when calling publisher.destroy()/subscription.destroy() (#603)
• Default incompatible qos callback should be set when there’s no user specified callback (#601)
• relax rate jitter test for individual periods (#602)
• add QoSProfile.__str__ (#593)
• Add useful debug info when trying to publish the wrong type (#581)
• Pass rutils_include_dirs to cppcheck (#577)
• wrap lines to shorten line length (#586)
• fix moved troubleshooting url (#579)
• improve error message if rclpy C extensions are not found (#580)
• Add message lost subscription event (#572)
• Fix executor behavior on shutdown (#574)
• Add missing rutils/macros.h header (#573)
• Add topic_name property to Subscription (#571)
• Add topic_name property to publisher (#568)
• Fix and document rclpy_handle_get_pointer_from_capsule() (#569)
• Fix docstrings (#566)


cpputils

• Update quality declaration links (#130)
• Add functions for getting library path and filename (#128)
• Add path equality operators (#127)
• Add create_temp_directory filesystem helper (#126)
• Use new noexcept specifier. (#123)
• Add stream operator for paths to make it easier to log (#120)
• Path join operator is const (#119)
• No windows.h in header files (#118)
• Fix rcpputils::SharedLibrary tests. (#117)
• Update QD to QL 1 (#114)
• Make sure to not try to index into an empty path. (#113)
• Fix working with filesystem parent paths. (#112)
• Cleanup mislabeled BSD license (#37)
• overload functions for has_symbol and get_symbol with raw string literal (#110)
• Add an ASSERT to the pointer traits tests. (#111)
• replace custom get env login into rcutils_get_env(). (#99)
• Removed Github Actions (#105)
• Update the package.xml files with the latest Open Robotics maintainers (#102)
• Make sure that an existing path is a directory for create_directories (#98)
• Transfer ownership to Open Robotics (#100)
• Ensure -fPIC is used when building a static lib (#93)
• Removed doxygen warnings (#86) (#87)
• Add clamp header (#85)
• Removed doxygen warnings (#86)
• Split get_env_var() into header and implementation (#83)
• Add cstring include for stremp (#81)
• filesystem helpers: adding remove_all to remove non-empty directories (#79)
• Add scope_exit helper (#78)
• Bump setup-ros to 0.0.23, action-ros-lint to 0.0.6, action-ros-ci to 0.0.17 (#77)
• Fix parent_path() for empty paths and paths of length one (#73)
• Add get_executable_name() function (#70)
• Address memory leak in remove pointer test (#72)
• Add current_path to filesystem_helpers (#63)
• Align path combine behavior with C++17 (#68)
• Update quality declaration to QL 2 (#71)

rcutils

• Declare dependency on libatomic (#338)
• updating quality declaration links (re: ros2/docs.ros2.org#52) (#335)
• Quiet down a warning in release mode. (#334)
• Make the logging separate char an implementation detail. (#332)
• Performance tests demo (#288)
• Remove references of __xstat (#330)
• Update the documentation to be more consistent. (#331)
- Shorten some excessively long lines of CMake (#328)
- qnx-support: include sys/link.h & avoid using dlinfo (#327)
- QNX uses XSI-compliant (#326)
- Add an API for directory iteration (#323)
- Fix a leak during error handling in dir size calculation (#324)
- Fix rcutils_shared_library_t path on Windows. (#322)
- Check linker flags instead of assuming compiler correlation. (#321)
- Improve shared library relative paths handling (#320)
- Update rcutils_calculate_directory_size() to support recursion (#306)
- Updating QD to QL 1 (#317)
- Address unused return values found in scan-build (#316)
- use one copy for continuous area instead of loop copy (#312)
- use a better way to check whether string is empty (#315)
- Use helper function to copy string (#314)
- Disable a Windows platform warning. (#311)
- Fix format of code description on document (#313)
- Make sure to check the return values of rcutils APIs. (#302)
- Add rcutils_expand_user() to expand user directory in path (#298)
- Update the maintainers. (#299)
- Remove the temporary variable in RCUTILS_LOGGING_AUTOINIT (#290)
- Add RCUTILS_NO_FAULT_INJECTION() macro. (#295)
- Inject faults on rcutils_get_env() and rcutils_set_env() call. (#292)
- env.h and get_env.h docblock fixes (#291)
- Introduce rcutils_strcasecmp, case insensitive string compare. (#280)
- Stop using fprintf to avoid using file handles by changing as few lines of code as possible. (#289)
- Defines QNX implementation for rcutils_get_platform_library_name (#287)
- Add RCUTILS_CAN_SET_ERROR_MSG_AND_RETURN_WITH_ERROR_OF() macro. (#284) To fault inject error messages as well as return codes.
- Change rcutils_fault_injection_set_count to use int64_t (#283)
- adds QNX support for rcutils_get_executable_name (#282)
- Add fault injection hooks to default allocator (#277)
- Fault injection macros and functionality (plus example) (#264)
- ensure -fPIC is used when building a static lib (#276)
- Drop vsnprintf mocks entirely. (#275) Binary API is not portable across platforms and compilation config.
- Fix vsnprintf mocks for Release builds. (#274)
- Improve test coverage mocking system calls (#272)
• Use mimick/mimick.h header (#273)
• Add mock test for rcutils_strerror (#265)
• Add compiler option -Wconversion and add explicit casts for conversions that may alter the value or change the sign (#263) See https://github.com/ros2/rcutils/pull/263#issuecomment-663252537.
• Removed doxygen warnings (#266) (#268)
• Removed doxygen warnings (#266)
• Force _GNU_SOURCE if glibc is used. (#267)
• Add parenthesis around the argument in time conversion macros defined in time.h (#261)
• Add token join macros (#262)
• Add rcutils_string_array_sort function (#248)
• Add rcutils_string_array_resize function (#247)
• Increase testing coverage of rcutils to 95% (#258)
• Update QUALITY_DECLARATION to reflect QL 2 status (#260)
• Update version stability section of quality declaration for 1.0 (#256)
• Contributors: Ahmed Sobhy, Alejandro Hernández Cordero, Barry Xu, Chen Lihui, Chris Lalancette, Christophe Bedard, Dirk Thomas, Felix Endres, Homalozoa X, Ivan Santiago Paunovic, Johannes Meyer, Jorge Perez, Karsten Knese, Michel Hidalgo, Scott K Logan, Stephen Brawner, Steven! Ragnarök, brawner, shonigmann, tomoya

resource_retriever

• Throw exception if package name is empty (#54)
• Update maintainers (#53)
• Add .hpp header and deprecate .h (#51)
• Add pytest.ini so local tests don’t display warning (#48)
• Contributors: Alejandro Hernández Cordero, Chris Lalancette, Jacob Perron, Shane Loretz

rmw

• Document which QoS policies are correctly read by rmw_get_publishers/subscriptions_info_by_topic (#308)
• Unique network flows (#294)
• updating quality declaration links (re: ros2/docs.ros2.org#52) (#307)
• Introduce RMW_DURATION_INFINITE constant and API return value promise (#301)
• Add declaration for function to check QoS profile compatibility (#299)
• Update the rmw_take_sequence documentation. (#297)
• Update rmw QD to QL 1 (#289)
• Extend rmw_qos_policy_kind_t, add functions to convert it to/from a string (#285)
• Add functions to convert between qos policy values and strings (#284)
• Update maintainers (#282)
• Update service request/response API documentation (#279)
• Update rmw_get_serialized_message_size docblock (#281)
• Update rmw_service_server_is_available doc (#280)
• Update wait and wait sets’ API documentation (#275)
• Update graph API documentation (#272)
• Update service server/client creation/destruction API documentation. (#276)
• Update rmw_*_*_allocation return values (#278)
• Update gid API documentation (#274)
• Do not link against pthread on Android (#267)
• Update taking API documentation (#271)
• Update publishing API documentation (#270)
• Add fault injection macros for use in other packages (#254)
• Add bad_alloc return to topic_endpoint_info functions (#269)
• Update publisher/subscription matched count API documentation (#262)
• Update publisher/subscription QoS query API documentation (#263)
• Extend rmw_serialized_message_t tests (#261)
• Update serialization/deserialization API documentation (#258)
• Update subscription API documentation (#256)
• Update publisher creation/destruction API documentation (#252)
• Add actual domain id to rmw_context_t (#251)
• Update node creation/destruction API documentation. (#249)
• Correct parameter names to match documentation (#250)
• Remove domain_id and localhost_only from node API (#248)
• Require enclave upon rmw_init() call. (#247)
• Update init/shutdown API documentation. (#243)
• Update init options API documentation. (#244)
• Add message lost subscription event (#232)
• Move statuses definitions to rmw/events_statuses/*.h (#232)
• Increase rmw testing coverage above 95% (#238)
• Handle zero-length names_and_types properly (#239)
• Add missing RMW_PUBLIC to security_options_set_root_path (#236)
• Update Quality Declaration for QL 2 (#233)
• Add Security Vulnerability Policy pointing to REP-2006. (#230)
**rmw_connextdds**

- Use rmw_qos_profile_unknown when adding entity to graph (#28)
- Resolve issues identified while investigating #21 (#22)
- Use Rolling in README’s Quick Start
- Improved implementation of client::is_service_available for Connext Pro
- Only add request header to typecode with Basic req/rep profile
- Remove commented/unused code
- Avoid topic name validation in get_info functions
- Reduce shutdown period to 10ms
- Pass HistoryQosPolicy to graph cache
- Reset error string after looking up type support
- Remove DDS-based WaitSet implementation
- Merge pull request #13 from Ericsson/unique_network_flows
- Refactor common API
- Update branch master to support Rolling only (#15)
- Add ability to override of endpoint qos settings based on topic name.
- Optimize QoS for reliable large data.
- Only trigger data condition if samples were loaned from reader.
- Alternative WaitSet implementation based on C++ std, selectable at compile-time.
- Add `<buildtool_export_depend>` for ament_cmake.
- Use default `dds.transport.UDPv4.builtin.ignore_loopback_interface`.
- Renamed environment variables: `RMW_CONNEXT_USE_DEFAULT_PUBLISH_MODE`, `RMW_CONNEXT_LEGACY_RMW_COMPATIBILITY_MODE`.
- Support a list of initial peers via `RMW_CONNEXT_INITIAL_PEERS`.
- Initial release.
- Contributors: Ananya Muddukrishna, Andrea Sorbini, Ivan Santiago Paunovic, William Woodall

**rmw_connextdds_common**

- Correctly detect empty messages (#33)
- Use rmw_qos_profile_unknown when adding entity to graph (#28)
- Resolve issues identified while investigating #21 (#22)
- Use Rolling in README’s Quick Start
- Improved implementation of client::is_service_available for Connext Pro
- Only add request header to typecode with Basic req/rep profile
- Remove commented/unused code
- Avoid topic name validation in get_info functions
- Reduce shutdown period to 10ms
- Pass HistoryQosPolicy to graph cache
- Reset error string after looking up type support
- Remove DDS-based WaitSet implementation
- Merge pull request #13 from Ericsson/unique_network_flows
- Remove superfluous header inclusion
- Remove conflicting linkage
- Further remove feature-based exclusion
- Remove feature-based exclusion
- Uncrustify
- Refactor common API
- Include required headers if feature is enabled
- Add conditional compilation support
- Prefer more generic file name
- Restrict unique flow endpoint check to versions beyond Foxy
- Indicate missing support for unique network flows
- Update branch master to support Rolling only (#15)
- Add ability to override of endpoint qos settings based on topic name (Pro).
- Optimize QoS for reliable large data (Pro).
- Only trigger data condition if samples were loaned from reader.
- Alternative WaitSet implementation based on C++ std, selectable at compile-time.
- Add `<buildtool_export_depend>` for ament_cmake.
- Use default `dds.transport.UDPv4.builtin.ignore_loopback_interface`.
- Don’t log an error on WaitSet::wait() timeout.
- Initial release.
- Contributors: Ananya Muddukrishna, Andrea Sorbini, Ivan Santiago Paunovic, William Woodall

**rmw_connextddsmicro**

- Use `rmw_qos_profile_unknown` when adding entity to graph (#28)
- Resolve issues identified while investigating #21 (#22)
- Use Rolling in README’s Quick Start
- Remove commented/unused code
- Avoid topic name validation in get_info functions
- Pass HistoryQosPolicy to graph cache
- Reset error string after looking up type support
- Remove DDS-based WaitSet implementation
• Merge pull request #13 from Ericsson/unique_network_flows
• Refactor common API
• Update branch master to support Rolling only (#15)
• Only trigger data condition if samples were loaned from reader.
• Alternative WaitSet implementation based on C++ std, selectable at compile-time.
• Add `<buildtool_export_depend>` for `ament_cmake`.
• Initial release.
• Contributors: Ananya Muddukrishna, Andrea Sorbini, Ivan Santiago Paunovic, William Woodall

**rmw_cyclonedds_cpp**

• Fix the history depth for KEEP_ALL. (#305)
• Use the macros from Cyclone DDS to work with sample payload when using SHM (#300)
• Add loaned sample zero-copy API support (#297)
• Indicate missing support for unique network flows (#282)
• Take and return new RMW_DURATION_INFINITE correctly (#288)
• Add RMW function to check QoS compatibility (#286)
• Fix use-after-free in error handling bug
• Drop compatibility with ancient cyclone versions
• Update to use Cyclone’s renamed ddsi_sertype
• Use init-on-first-use for global state (#275)
• Make sure to reset the error when a typesupport can’t be found.
• Switch to using the generic functions for the typesupport handles.
• Handle typesupport errors on fetch. (#271)
• Handle potential divide by 0 (#267)
• Fix incorrect log message(rmw_fastrtps_shared_cpp -> rmw_cyclonedds_cpp) (#260)
• Update maintainers (#254)
• Change wrong use of %ld to print std::size_t to %zu
• Return RMW_RET_UNSUPPORTED in rmw_get_serialized_message_size (#250)
• Update service/client request/response API error returns (#249)
• Updated publisher/subscription allocation and wait set API return codes (#246)
• Fix array `get_function` semantics (#248)
• Update service/client construction/destruction API return codes. (#247)
• Update gid API return codes. (#244)
• Update graph API return codes. (#243)
• Check for message_info on take where appropriate. (#245) Fix for regression introduced in #241.
• Updated error returns on rmw_take_serialized() and rmw_take_with_message_info() (#242)
• Updated error returns on rmw_take() (#241)
• Add quality declaration for Cyclone DDS (#218)
• Fix that not to delete some objects after destroying functions (#236)
• Update rmw_publish_serialized_message() error returns (#240)
• Update rmw_publish() error returns (#239)
• Remove public declarations (#230)
• Use quotes for non-system includes (#231)
• Use correct functions to resize and get an item, avoiding memory leaks in typesupport code (#228)
• Fix context cleanup. (#227)
• Fix memory leak that type support not deleted. (#225)
• Ensure compliant matched pub/sub count API. (#223)
• Fix memory leak that string not deleted. (#224)
• Change RET_WRONG_IMPLID() to return RMW_RET_INCORRECT_IMPLEMENTATION (#226)
• Fix bad conditional in rmw_serialize(). (#217)
• Ensure compliant subscription API. (#214)
• Ensure compliant publisher API (#210)
• rmw_destroy_node must remove node from graph cache (#213)
• Add space between ‘ROS’ and ‘2’ (#195)
• Set context actual domain id (#208)
• Ensure compliant node construction/destruction API (#206)
• Remove domain_id and localhost_only from node API (#205)
• Amend rmw_init() implementation: require enclave. (#204)
• Ensure compliant init/shutdown API implementations. (#202)
• Ensure compliant init options API implementations. (#200)
• Finalize context iff shutdown. (#196)
• Handle RMW_DEFAULT_DOMAIN_ID. (#194)
• Add support to message lost event (#192)
• Mitigate lost service responses discovery issue (#187)

Contributors: Alejandro Hernández Cordero, Ananya Muddukrishna, Chen Lihui, Chris Lalancette, Christophe Bedard, Dan Rose, Emerson Knapp, Erik Boasson, Ivan Santiago Paunovic, Jacob Perron, Joe Speed, José Tomas Lorente, Lobotuerk, Michel Hidalgo, Scott K Logan, Stephen Brawner, Sumanth Nirmal, Sven Brinkmann, eboasson, pluris
rmw_dds_common

- Fix one more instance of index.ros.org. (#49)
- updating quality declaration links (re: ros2/docs.ros2.org#52) (#46)
- Add function for checking QoS profile compatibility (#45)
- Shorten some excessively long lines of CMake (#44)
- Fix test_graph_cache ASAN errors (#41) (#42)
- Update QD to QL 1 (#38)
- Create a utility function to limit rmw_time_t to 32-bit values (#37)
- Update maintainers (#34)
- Updated performance section QD (#30)
- Update Quality Declaration to QL2 (#31)
- Added benchmark test to rmw_dds_common (#29)
- Fix potential memory leak (#28)
- Add fault injection macro unit tests (#27)
- Fixed some doxygen warnings (#26)
- Update Quality Declaration to QL3 (#24)
- Update QD and documentation (#23)
- Contributors: Alejandro Hernández Cordero, Chen Lihui, Chris Lalancette, Ivan Santiago Paunovic, Jacob Perron, Michael Jeronimo, Michel Hidalgo, Scott K Logan, Stephen Brawner, shonigmann, y-okumura-isp

rmw_fastrtps_cpp

- Refactor to use DDS standard API (#518)
- Unique network flows (#502)
- updating quality declaration links (re: ros2/docs.ros2.org#52) (#520)
- Add RMW function to check QoS compatibility (#511)
- Capture cdr exceptions (#505)
- Load profiles based on topic names (#335)
- Set rmw_dds_common::GraphCache callback after init succeeds. (#496)
- Handle typesupport errors on fetch. (#495)
- Check for correct context shutdown (#486)
- New environment variable to change easily the publication mode (#470)
- Discern when the Client has gone from when the Client has not completely matched (#467) * Workaround when the client is gone before server sends response * Change add to the map to listener callback
- Update the package.xml files with the latest Open Robotics maintainers (#459)
- Update Quality Declarations and READMEs (#455) * Add QD links for dependencies to rmw_fastrtps_cpp QD * Provide external dependencies QD links * Update rmw_fastrtps README to use Fast DDS * Update rmw_fastrtps_cpp QD: Fast DDS & unit test * Update README rmw_fastrtps_cpp to QL2
• Perform fault injection in all creation/destruction APIs. (#453)
• Ensure rmw_destroy_node() completes despite run-time errors. (#458)
• Update rmw_fastrtps_cpp and rmw_fastrtps_shared_cpp QDs to QL2. (#456)
• Return RMW_RET_UNSUPPORTED in rmw_get_serialized_message_size (#452)
• Updated publisher/subscription allocation and wait set API return codes (#443)
• Added rmw_logging tests (#442)
• Make service/client construction/destruction implementation compliant (#445)
• Make sure type can be unregistered successfully (#437)
• Add tests for native entity getters. (#439)
• Avoid deadlock if graph update fails. (#438)
• Call Domain::removePublisher while failure occurs in create_publisher (#434)
• Ensure compliant matched pub/sub count API. (#424)
• Ensure compliant publisher QoS queries. (#425)
• Ensure compliant subscription API. (#419)
• Ensure compliant publisher API. (#414)
• Set context actual domain id (#410)
• Ensure compliant node construction/destruction API. (#408)
• Remove domain_id and localhost_only from node API (#407)
• Amend rmw_init() implementation: require enclave. (#406)
• Update Quality Declarations to QL3. (#404)
• Ensure compliant init/shutdown API implementation. (#401)
• Update Quality Declaration to QL3. (#403)
• Finalize context iff shutdown. (#396)
• Make service wait for response reader (#390)
• Contributors: Alejandro Hernández Cordero, Barry Xu, Eduardo Ponz Segrelles, Ignacio Montesino Valle, Ivan Santiago Paunovic, JLBuenoLopez-eProsima, Jacob Perron, Jaime Martin Losa, José Luis Bueno López, Michael Jeronimo, Michel Hidalgo, Miguel Company, shonigmann

rmw_fastrtps_dynamic_cpp

• Refactor to use DDS standard API (#518)
• Unique network flows (#502)
• updating quality declaration links (re: ros2/docs.ros2.org#52) (#520)
• Add RMW function to check QoS compatibility (#511)
• Capture cdr exceptions (#505)
• Load profiles based on topic names in rmw_fastrtps_dynamic_cpp (#497)
• Set rmw_dds_common::GraphCache callback after init succeeds. (#496)
• Handle typesupport errors on fetch. (#495)
• Check for correct context shutdown (#486)
• New environment variable to change easily the publication mode (#470)
• Discriminate when the Client has gone from when the Client has not completely matched (#467) * Workaround when the client is gone before server sends response * Change add to the map to listener callback
• Update the package.xml files with the latest Open Robotics maintainers (#459)
• Update Quality Declarations and READMEs (#455) * Add QL of external dependencies to rmw_fastrtps_dynamic_cpp QD * Add QD links for dependencies to rmw_fastrtps_dynamic_cpp QD * Provide external dependencies QD links * Add README to rmw_fastrtps_dynamic * Add QD for rmw_fastrtps_dynamic
• Ensure rmw_destroy_node() completes despite run-time errors. (#458)
• Return RMW_RET_UNSUPPORTED in rmw_get_serialized_message_size (#452)
• Updated publisher/subscription allocation and wait set API return codes (#443)
• Added rmw_logging tests (#442)
• Fix array get_function semantics (#448)
• Make service/client construction/destruction implementation compliant (#445)
• Make sure type can be unregistered successfully (#437)
• Add tests for native entity getters. (#439)
• Avoid deadlock if graph update fails. (#438)
• Call Domain::removePublisher while failure occurs in create_publisher (#434)
• Avoid memory leaks and undefined behavior in rmw_fastrtps_dynamic_cpp typesupport code (#429)
• Ensure compliant matched pub/sub count API. (#424)
• Ensure compliant publisher QoS queries. (#425)
• Ensure compliant subscription API. (#419)
• Ensure compliant publisher API. (#414)
• Set context actual domain id (#410)
• Ensure compliant node construction/destruction API. (#408)
• Remove domain_id and localhost_only from node API (#407)
• Amend rmw_init() implementation: require enclave. (#406)
• Ensure compliant init/shutdown API implementation. (#401)
• Finalize context iff shutdown. (#396)
• Make service wait for response reader (#390)
• Contributors: Alejandro Hernández Cordero, Barry Xu, Eduardo Ponz Segrelles, Ignacio Montesino Valle, Ivan Santiago Paunovic, JLBuenoLopez-eProsima, Jacob Perron, Jaime Martin Losa, José Luis Bueno López, Michael Jeronimo, Michel Hidalgo, Miguel Company, shonigmann
rmw_fastrtps_shared_cpp

- Refactor to use DDS standard API (#518)
- Unique network flows (#502)
- updating quality declaration links (re: ros2/docs.ros2.org#52) (#520)
- Take and return new RMW_DURATION_INFINITE correctly (#515)
- Add RMW function to check QoS compatibility (#511)
- Capture cdr exceptions (#505)
- Make sure to lock the mutex protecting client_endpoints_. (#492)
- Use interface whitelist for localhost only (#476)
- Make use of error return value in decrement_context_impl_ref_count (#488)
- Remove unnecessary includes (#487)
- Use new time_utils function to limit rmw_time_t values to 32-bits (#485)
- New environment variable to change easily the publication mode (#470)
- Remove unused headers MessageTypeSupport.hpp and ServiceTypeSupport.hpp (#481)
- Discriminate when the Client has gone from when the Client has not completely matched (#467) * Workaround when the client is gone before server sends response * Change add to the map to listener callback
- Update the package.xml files with the latest Open Robotics maintainers (#459)
- Update Quality Declarations and READMEs (#455) * Add QD links for dependencies to rmw_fastrtps_shared_cpp QD. * Provide external dependencies QD links. * Update rmw_fastrtps_shared_cpp QD: Fast DDS * Update README rmw_fastrtps_shared_cpp to QL2
- Perform fault injection in all creation/destruction APIs. (#453)
- Ensure rmw_destroy_node() completes despite run-time errors. (#458)
- Handle too large QoS queue depths. (#457)
- Update rmw_fastrtps_cpp and rmw_fastrtps_shared_cpp QDs to QL2. (#456)
- checked client implementation and return RMW_RET_INCORRECT_RMW_IMPLEMENTATION (#451)
- Update service/client request/response API error returns (#450)
- Updated publisher/subscriber allocation and wait set API return codes (#443)
- Added rmw_logging tests (#442)
- Add tests for RMW QoS to DDS attribute conversion. (#449)
- Make service/client construction/destruction implementation compliant (#445)
- Inject faults on __rmw_publish() and run_listener_thread() call. (#441)
- Update gid API return codes. (#440)
- Update graph API return codes. (#436)
- Update rmw_take_serialized() and rmw_take_with_message_info() error returns (#435)
- Update rmw_take() error returns (#432)
- Update rmw_publish() error returns (#430)
- Update rmw_publish_serialized_message() error returns (#431)
• Improve __rmw_create_wait_set() implementation. (#427)
• Ensure compliant matched pub/sub count API. (#424)
• Ensure compliant publisher QoS queries. (#425)
• Fix memory leak that wait_set might be not destroyed in some case. (#423)
• Avoid unused identifier variable warnings. (#422)
• Fix trying to get topic data that was already removed. (#417)
• Ensure compliant subscription API. (#419)
• Use package path to TypeSupport.hpp headers in ServiceTypeSupport and MessageTypeSupport (#415) Use package in path to TypeSupport header for ServiceTypeSupport/MessageTypeSupport
• Ensure compliant publisher API. (#414)
• Set context actual domain id (#410)
• Add missing thread-safety annotation in ServicePubListener (#409)
• Ensure compliant node construction/destruction API. (#408)
• Update Quality Declarations to QL3. (#404)
• Do not use string literals as implementation identifiers in tests. (#402)
• Ensure compliant init options API implementations. (#399)
• Finalize context iff shutdown. (#396)
• Handle RMW_DEFAULT_DOMAIN_ID. (#394)
• Make service wait for response reader (#390)
• Contributors: Alejandro Hernández Cordero, Chen Lihui, Chris Lalancette, Emerson Knapp, Ivan Santiago Paunovic, JL Bueno López-eProsima, Jacob Perron, Jaime Martín Losa, Jose Luis Rivero, Jose Tomas Lorente, José Luis Bueno López, Lobotuerk, Michael Jeronimo, Michel Hidalgo, Miguel Company, Stephen Brawner, shonigmann

rmw_implementation

• Unique network flows (#170)
• updating quality declaration links (re: ros2/docs.ros2.org#52) (#185)
• Remove rmw_connext_cpp. (#183)
• Add support for rmw_connextdds (#182)
• Add function for checking QoS profile compatibility (#180)
• Shorten some excessively long lines of CMake (#179)
• Add rmw_fastrtps_dynamic_cpp to the explicit group deps (#177)
• Accept any RMW implementation, not just the default (#172)
• Defer path resolution of rmw implementation libraries to dynamic linker. (#169)
• Update QD to QL 1 (#166)
• Fix up C functions to never throw. (#149)
• Restored Dirk as author (#155)
• Update maintainers (#154)
• Updated performance QD section (#153)
• Update Quality Declaration to QL2. (#151)
• Add nominal test for symbol prefetch() and unload. (#145)
• Added benchmark test to rmw_implementation (#127)
• Test load and lookup functionality. (#135)
• Remove domain_id and localhost_only from node API (#114)
• Move the quality declaration into the rmw_implementation subdirectory. (#111)
• Contributors: Alejandro Hernández Cordero, Ananya Muddukrishna, Andrea Sorbini, Chris Lalancette, Ivan Santiago Paunovic, Jacob Perron, Michel Hidalgo, Scott K Logan, Stephen Brawner, shonigmann

rmw_implementation_cmake

• Shorten some excessively long lines of CMake (#300)
• Change default RMW vendor to CycloneDDS. (#293)
• Update rmw QD to QL 1 (#289)
• Update maintainers (#282)
• Contributors: Chris Lalancette, Ivan Santiago Paunovic, Scott K Logan, Stephen Brawner

robot_state_publisher

• Stop rejecting unknown parameters. (#161)
• clean up license to be standard bsd 3 clause (#130)
• Update the maintainers. (#151)
• fix types in range loops to avoid copy due to different type (#143)
• Make sure not to crash on an invalid URDF. (#141)
• Don’t export exe as library (#25) (ros2 #28)
• Contributors: Chris Lalancette, Dirk Thomas, Tully Foote

ros1_bridge

• Fix logging for updated rclcpp interface (#303)
• Fix typo in comments (#297)
• Update to use rosidl_parser and .idl files rather than rosidl_adapter and .msg files (#296)
• Update maintainers (#286)
• use hardcoded QoS (keep all, transient local) for /tf_static topic in dynamic_bridge (#282)
• document explicitly passing the topic type to ‘ros2 topic echo’ (#279)
• Fix multiple definition if message with same name as service exists (#272)
• Contributors: Dirk Thomas, Jacob Perron, Michael Carroll, Vicidel, William Woodall

ros2action

• Add changelog. (#636)
• Remove maintainer. (#597)
• Add Audrow as a maintainer. (#591)
• Update maintainers. (#568)
• Contributors: Audrow Nash, Claire Wang, Ivan Santiago Paunovic

ros2bag

• /clock publisher in Player (#695)
• Introducing Reindexer CLI (#699)
• rosbag2_py pybind wrapper for “record” - remove rosbag2_transport_py (#702)
• Add rosbag2_py::Player::play to replace rosbag2_transport_python version (#693)
• Explicitly add emersonknapp as maintainer (#692)
• use rosbag2_py for ros2 bag info (#673)
• CLI query rosbag2_py for available storage implementations (#659)
• Recorder –regex and –exclude options (#604)
• Fix the tests on cyclonedds by translating qos duration values (#606)
• SQLite storage optimized by default (#568)
• Fix a bug on parsing wrong description in plugin xml file (#578)
• Compress bag files in separate threads (#506)

Co-authored-by: Piotr Jaroszek <piotr.jaroszek@robotec.ai>

• read yaml config file (#497)
• List all storage plugins in plugin xml file (#554)
• add storage_config_uri (#493)
• Update deprecated qos policy value names (#548)
• Add record test for ros2bag (#523)
• Removed duplicated code in record (#534)
• Change default cache size for sequential_writer to a non zero value (#533)
• Update the package.xml files with the latest Open Robotics maintainers (#535)
• [ros2bag test_record] Gets rid of time.sleep and move to using command.wait_for_output (#525)
• Add pytest.ini back to ros2bag. (#492)
• performance testing packages (#442)
• Validate QoS profile values are not negative. (#483)
• catch parent exception (#472)
• add wait for closed file handles on Windows (#470)
• introduce ros2 bag list <plugins> (#468)
• move wait_for_shutdown() call out of the context manager (#466)
• Adding db directory creation to rosbag2_cpp (#450)
• use a single temp dir for the test class (#462)
• Add per-message ZSTD compression (#418)
• Add split by time to recording (#409)
• Add pytest.ini so local tests don’t display warning (#446)
• Contributors: Adam Dąbrowski, Barry Xu, Chris Lalancette, Dirk Thomas, Emerson Knapp, Ivan Santiago Paunovic, Jacob Perron, Jaison Titus, Jesse Ikawa, Karsten Knese, Marwan Taher, Michael Jeronimo, P. J. Reed, jhdcs

ros2cli

• Add changelog. (#636)
• Ensure only one daemon can run at a time. (#622)
• Remove maintainer. (#597)
• Add option to support use_sim_time. (#581)
• Bugfix for #563. (#570)
• Add Audrow as a maintainer. (#591)
• Support Python 3.8-provided importlib.metadata. (#585)
• Update maintainers. (#568)
• Added dependency to python3-argcomplete to ros2cli. (#564)
• Remove use of pkg_resources from ros2cli. (#537)
• Contributors: Audrow Nash, Chris Lalancette, Claire Wang, Daisuke Sato, Ivan Santiago Paunovic, Michel Hidalgo, Scott K Logan, Tomoya Fujita, Yoan Mollard
ros2cli_common_extensions

- remove maintainer (#5)
- update maintainer (#4)
- First implementation (#2)
- Contributors: Bo Sun, Claire Wang

ros2cli_test_interfaces

- Add changelog. (#636)
- Remove maintainer. (#597)
- Add Audrow as a maintainer. (#591)
- Make ros2cli_test_interfaces version equal to other packages.
- Remove ros2interface test dependencies on builtin interface. (#579)
- Contributors: Audrow Nash, Chris Lalancette, Claire Wang, Ivan Santiago Paunovic

ros2component

- Add changelog. (#636)
- Remove maintainer. (#597)
- Add Audrow as a maintainer. (#591)
- Update maintainers. (#568)
- Ensure consistent timeout in ros2component list. (#526)
- Contributors: Audrow Nash, Claire Wang, Ivan Santiago Paunovic, Michel Hidalgo

ros2doctor

- Improve ros2 doctor on Windows. (#631) (#634)
- Add changelog. (#636)
- Continue to next iteration after exceptions in generate_reports. (#623)
- Remove maintainer. (#597)
- Add Audrow as a maintainer. (#591)
- Support Python 3.8-provided importlib.metadata. (#585)
- Update maintainers. (#568)
- Remove pkg_resources from ros2doctor. (#537)
- Make ros2doctor depend on ros_environment and fix platform.py bug on error. (#538)
- Refactor ros2doctor hello verb. (#521)
- Contributors: Alberto Soragna, Audrow Nash, Chris Lalancette, Claire Wang, Ivan Santiago Paunovic, Michel Hidalgo, Scott K Logan, mergify[bot]
ros2interface

- Add changelog. (#636)
- Remove maintainer. (#597)
- Add Audrow as a maintainer. (#591)
- Remove ros2interface test dependencies on builtin interface. (#579)
- Update maintainers. (#568)
- Handle inline comments on constants correctly. (#548)
- Update quoted comments in the test (#540)
- Add option to include/remove whitespace and comments. (#527)
- Show “expanded” message definition. (#524)
- Contributors: Audrow, Audrow Nash, Claire Wang, Ivan Santiago Paunovic, Tully Foote

ros2launch

- Add options extensions to ros2launch and extensibility to the node action (#216)
- Support non-interactive ros2 launch executions (#210)
- Merge pull request #183 from ros2/update-maintainers
- Move previous maintainer to <author>
- Update the package.xml files with the latest Open Robotics maintainers
- Add pytest.ini so local tests don’t display warning (#152)
- Contributors: Chris Lalancette, Geoffrey Biggs, Michael Jeronimo, Michel Hidalgo

ros2lifecycle

- Add changelog. (#636)
- Remove maintainer. (#597)
- Add Audrow as a maintainer. (#591)
- Update maintainers. (#568)
- Contributors: Audrow Nash, Claire Wang, Ivan Santiago Paunovic

ros2lifecycle_test_fixtures

- Add changelog. (#636)
- Depend on rclcpp::rclcpp target. (#618) Contributors: Audrow Nash
- Remove maintainer. (#597)
- Add Audrow as a maintainer. (#591)
- Update maintainers. (#568)
- Contributors: Audrow Nash, Claire Wang, Ivan Santiago Paunovic
ros2multicast

- Add changelog. (#636)
- Remove maintainer. (#597)
- Add Audrow as a maintainer. (#591)
- Update maintainers. (#568)
- Contributors: Audrow Nash, Claire Wang, Ivan Santiago Paunovic

ros2node

- Add changelog. (#636)
- Remove maintainer. (#597)
- Add Audrow as a maintainer. (#591)
- Update maintainers. (#568)
- Contributors: Audrow Nash, Claire Wang, Ivan Santiago Paunovic

ros2param

- Add changelog. (#636)
- Make the ros2param –filter test more reliable. (#606)
- Add wildcard loading to ros2 param load. (#602)
- Ros2 param dump/load should use fully qualified node names. (#600)
- Add –filter options for ‘ros2 param list’. (#592)
- Remove maintainer. (#597)
- Add rosparam verb load. (#590)
- Add Audrow as a maintainer. (#591)
- Add “–param-type” option to ros2param list. (#572)
- Update maintainers. (#568)
- Contributors: Audrow Nash, Claire Wang, Ivan Santiago Paunovic, Victor Lopez, tomoya

ros2pkg

- Add changelog. (#636)
- Use underscores in setup.cfg.em instead of dashes. (#627)
- Add space for “ROS 2”. (#617)
- Use target_compile_features for c/c++ standards. (#615)
- Remove maintainer. (#597)
- Add Audrow as a maintainer. (#591)
- Declare missing dependency on python3-importlib-resources. (#584)
- Update maintainers. (#568)
- Fix incorrect EXPORT for executables. (#545)
- Switch ros2pkg to using importlib.
- Contributors: Audrow Nash, Chris Lalancette, Claire Wang, Dirk Thomas, Ivan Santiago Paunovic, Scott K Logan, Shane Loretz

**ros2run**

- Add changelog. (#636)
- Remove maintainer. (#597)
- Add Audrow as a maintainer. (#591)
- Update maintainers. (#568)
- Contributors: Audrow Nash, Claire Wang, Ivan Santiago Paunovic

**ros2service**

- Add changelog. (#636)
- Remove maintainer. (#597)
- Add Audrow as a maintainer. (#591)
- Update maintainers. (#568)
- Check that passed type is actually a service. (#559)
- Contributors: Audrow Nash, Claire Wang, Dirk Thomas, Ivan Santiago Paunovic

**ros2test**

- Add pytest.ini so local tests don’t display warning (#8)
- Contributors: Chris Lalancette

**ros2topic**

- Add changelog. (#636)
- Add verbose info for topic list. (#351)
- Remove maintainer. (#597)
- Add option to support use_sim_time. (#581)
- Add Audrow as a maintainer. (#591)
- Add filter option to ros2topic . (#575)
- Update deprecated qos policy value names. (#571)
- Update maintainers. (#568)
- Fix the test to use the topic name. (#566)
• Improve the error message for invalid message types. (#558)
• Use reliable QoS for ros2topic tests. (#555)
• Add option to echo serialized messages. (#470)
• Enable –no-daemon flag for some cli tools. (#514)
• Use transient_local and longer keep-alive for pub tests. (#546)
• Add –keep-alive option to ‘topic pub’. (#544)
• Add option to ros2 topic echo to report lost messages. (#542)
• Support QoS Depth and History via ros2 topic pub/echo. (#528)
• Contributors: Audrow Nash, ChenYing Kuo, Chris Lalancette, Claire Wang, Dereck Wonnacott, Dirk Thomas, Ivan Santiago Paunovic, Jacob Perron, Scott K Logan, Tomoya Fujita, tomoya

ros_testing

• Use rostest CMake target as output file basename. (#9)
• Contributors: Michel Hidalgo

rosbag2

• Explicitly add emersonknapp as maintainer (#692)
• RMW-implementation-searcher converter in rosbag2_cpp (#670)
• Move zstd compressor to its own package (#636)
• add storage_config_uri (#493)
• Update the package.xml files with the latest Open Robotics maintainers (#535)
• AMENT_IGNORE rosbag2_py for now (#509)
• rosbag2_py reader and writer (#308)
• Contributors: Emerson Knapp, Karsten Knese, Mabel Zhang, Michael Jeronimo

rosbag2_compression

• Explicitly add emersonknapp as maintainer (#692)
• Reindexer core (#641) Add a new C++ Reindexer class for reconstructing metadata from bags that are missing it.
• CLI query rosbag2_py for available storage implementations (#659)
• Move zstd compressor to its own package (#636)
• Remove rosbag2_compression test dependencies on zstd implementation in prep for moving it into a separate package (#637)
• Make compressor implementations into a plugin via pluginlib (#624)
• Use ZSTD’s streaming interface for [de]compressing files (#543)
• Fix build issues when rosbag2_storage is binary installed (#585)
• Fix relative metadata paths in SequentialCompressionWriter (#613)
• Fix deadlock race condition on compression shutdown (#616)
• Deduplicate SequentialCompressionReader business logic, add fallback to find bagfiles in incorrectly-written metadata (#612)
• Compress bag files in separate threads (#506)
• Sqlite storage double buffering (#546)
• add storage_config_uri (#493)
• Update the package.xml files with the latest Open Robotics maintainers (#535)
• Do not expect empty StorageOptions URI to work in CompressionWriterTest (#526)
• Remove some code duplication between SequentialWriter and SequentialCompressionWriter (#527)
• Fix exception thrown given invalid arguments with compression enabled (#488)
• Adding db directory creation to rosbag2_cpp (#450)
• Consolidate ZSTD utility functions (#459)
• Add per-message ZSTD compression (#418)
• Contributors: Adam Dąbrowski, Christophe Bedard, Devin Bonnie, Emerson Knapp, Jaison Titus, Karsten Knese, Marwan Taher, Michael Jeronimo, P. J. Reed, jhdcs

rosbag2_compression_zstd

• Add test_depend ament_cmake_gmock (#639)
• Move zstd compressor to its own package (#636)
• Contributors: Emerson Knapp, Shane Loretz

rosbag2_cpp

• Add set_rate to PlayerClock (#727)
• Enforce non-null now_fn in TimeControllerClock (#731)
• Fix pause snapshot behavior and add regression test (#730)
• Pause/resume PlayerClock (#704)
• Remove -Werror from builds, enable it in Action CI (#722)
• Enable thread safety analysis for rosbag2_cpp and add annotations in TimeControllerClock (#710)
• PlayerClock initial implementation - Player functionally unchanged (#689)
• Explicitly add emersonknapp as maintainer (#692)
• Reindexer core (#641) Add a new C++ Reindexer class for reconstructing metadata from bags that are missing it.
• use rclepp serialized messages to write data (#457)
• alternative write api (#676)
• RMW-implementation-searcher converter in rosbag2_cpp (#670)
• CLI query rosbag2_py for available storage implementations (#659)
• Fix -topics flag for ros2 bag play being ignored for all bags after the first one. (#619)
• Fix a crash in test_message_cache. (#635)
• Fix build issues when rosbag2_storage is binary installed (#585)
• Deduplicate SequentialCompressionReader business logic, add fallback to find bagfiles in incorrectly-written metadata (#612)
  • include what you use (#600)
• Only dereference the data pointer if it is valid. (#581)
• Add back rosbag2_cpp::StorageOptions as deprecated (#563)
• Sqlite storage double buffering (#546)
• correct master build (#552)
• add storage_config_uri (#493)
• Mutex around writer access in recorder (#491)
• if cache data exists, it needs to flush the data into the storage before shutdown (#541)
• Change default cache size for sequential_writer to a non zero value (#533)
• SequentialWriter to cache by message size instead of message count (#530)
• Update the package.xml files with the latest Open Robotics maintainers (#535)
• Remove some code duplication between SequentialWriter and SequentialCompressionWriter (#527)
• disable sanitizer by default (#517)
• Fix typo in error message (#475)
• introduce defaults for the C++ API (#452)
• Adding db directory creation to rosbag2_cpp (#450)
• comment out unused variable (#460)
• minimal c++ API test (#451)
• Add split by time to recording (#409)
• Contributors: Adam Dąbrowski, Alexander, Chris Lalancette, Dirk Thomas, Emerson Knapp, Ivan Santiago Paunovic, Jacob Perron, Jaison Titus, Karsten Knese, Marwan Taher, Michael Jeronimo, P. J. Reed, Patrick Spieler, Tomoya Fujita, jhdcs

rosbag2_interfaces

• Add rosbag2_interfaces package with playback service definitions (#728)
• Contributors: Emerson Knapp

4.8. ROS 2 Documentation
rosbag2_performance_benchmarking

- fixed a memory leak in no-transport benchmark (#674)
- report of performance improvements in rosbag2 (roughly since Foxy) (#651)
- Performance benchmarking improvements (#634)
- Performance benchmarking refactor (#594)
- Sqlite storage double buffering (#546)
- read yaml config file (#497)
- add storage_config_uri (#493)
- Update the package.xml files with the latest Open Robotics maintainers (#535)
- performance testing packages (#442)
- Contributors: Adam Dąbrowski, Karsten Knese, Michael Jeronimo, Piotr Jaroszek

rosbag2_py

- Remove -Werror from builds, enable it in Action CI (#722)
- Split Rosbag2Transport into Player and Recorder classes - first pass to enable further progress (#721)
- /clock publisher in Player (#695)
- Introducing Reindexer CLI (#699)
- Fix rosbag2_py transport test for py capsule (#707)
- rosbag2_py pybind wrapper for “record” - remove rosbag2_transport_py (#702)
- Add rosbag2_py::Player::play to replace rosbag2_transport_python version (#693)
- Explicitly add emersonknapp as maintainer (#692)
- RMW-implementation-searcher converter in rosbag2_cpp (#670)
- use rosbag2_py for ros2 bag info (#673)
- CLI query rosbag2_py for available storage implementations (#659)
- Fix build issues when rosbag2_storage is binary installed (#585)
- Fix the tests on cyclonedds by translating qos duration values (#606)
- add storage_config_uri (#493)
- Workaround pybind11 bug on Windows when CMAKE_BUILD_TYPE=RelWithDebInfo (#538)
- Update the package.xml files with the latest Open Robotics maintainers (#535)
- Fix rosbag2_py on Windows debug and stop ignoring the package (#531)
- Fix rosbag2_py bug when using libc++ (#529)
- AMENT_IGNORE rosbag2_py for now (#509)
- rosbag2_py reader and writer (#308)
- Contributors: Emerson Knapp, Ivan Santiago Paunovic, Karsten Knese, Mabel Zhang, Michael Jeronimo, P. J. Reed, jhdcs
robag2_storage

• Remove -Werror from builds, enable it in Action CI (#722)
• PlayerClock initial implementation - Player functionally unchanged (#689)
• Explicitly add emersonknapp as maintainer (#692)
• Reindexer core (#641) Add a new C++ Reindexer class for reconstructing metadata from bags that are missing it.
• Remove outdated pluginlib cmake script from robag2_storage (#661)
• CLI query robag2_py for available storage implementations (#659)
• Shorten some excessively long lines of CMake (#648)
• SQLite storage optimized by default (#568) * Use optimized pragmas by default in sqlite storage. Added option to use former behavior
• Use std::filesystem compliant non-member exists function for path object (#593)
• Update codes since rcutils_calculate_directory_size() is changed (#567)
• add storage_config_uri (#493)
• Update the package.xml files with the latest Open Robotics maintainers (#535)
• Add split by time to recording (#409)
• Contributors: Adam Dąbrowski, Barry Xu, Emerson Knapp, Josh Langsfeld, Karsten Knese, Michael Jeronimo, Scott K Logan, jhdcs

robag2_storage_default_plugins

• Remove -Werror from builds, enable it in Action CI (#722)
• Explicitly add emersonknapp as maintainer (#692)
• Reindexer core (#641) Add a new C++ Reindexer class for reconstructing metadata from bags that are missing it.
• Fix build issues when robag2_storage is binary installed (#585)
• Mutex protection for db writing and stl collections in writer & storage (#603)
• SQLite storage optimized by default (#568)
• read yaml config file (#497)
• add storage_config_uri (#493)
• Update the package.xml files with the latest Open Robotics maintainers (#535)
• Contributors: Adam Dąbrowski, Emerson Knapp, Karsten Knese, Michael Jeronimo, P. J. Reed, jhdcs
Vulcanexus Documentation, Release 1.0.0

rosbag2_test_common

- Remove -Werror from builds, enable it in Action CI (#722)
- Fix bad_function_call by replacing rclcpp::spin_some with SingleThreadedExecutor (#705)
- Explicitly add emersonknapp as maintainer (#692)
- Remove temporary directory platform-specific logic from test fixture (#660)
- Stabilize test_record by reducing copies of executors and messages (#576)
- Update the package.xml files with the latest Open Robotics maintainers (#535)
- Contributors: Emerson Knapp, Michael Jeronimo

rosbag2_tests

- Remove -Werror from builds, enable it in Action CI (#722)
- Explicitly add emersonknapp as maintainer (#692)
- Reindexer core (#641) Add a new C++ Reindexer class for reconstructing metadata from bags that are missing it.
- use rclcpp serialized messages to write data (#457)
- Alternative write api (#676)
- RMW-implementation-searcher converter in rosbag2_cpp (#670)
- Use rosbag2_py for ros2 bag info (#673)
- Remove temporary directory platform-specific logic from test fixture (#660)
- Fix --topics flag for ros2 bag play being ignored for all bags after the first one. (#619)
- Move zstd compressor to its own package (#636)
- Fix relative metadata paths in SequentialCompressionWriter (#613)
- Recorder -regex and --exclude options (#604)
- Fix the tests on cyclonedds by translating qos duration values (#606)
- add storage_config_uri (#493)
- Removed duplicated code in record (#534)
- Change default cache size for sequential_writer to a non zero value (#533)
- Update the package.xml files with the latest Open Robotics maintainers (#535)
- Mark flaky tests as xfail for now (#520)
- introduce defaults for the C++ API (#452)
- Adding db directory creation to rosbag2_cpp (#450)
- minimal c++ API test (#451)
- Add split by time to recording (#409)
- Contributors: Adam Dąbrowski, Alexander, Emerson Knapp, Jaison Titus, Karsten Knese, Marwan Taher, Michael Jeronimo, jhdcs

428 Chapter 4. Contributing to the documentation
Vulcanexus Documentation, Release 1.0.0

rosbag2_transport

- cleanup cmakelists (#726)
- turn recorder into a node (#724)
- turn player into a node (#723)
- Remove -Werror from builds, enable it in Action CI (#722)
- Split Rosbag2Transport into Player and Recorder classes - first pass to enable further progress (#721)
- /clock publisher in Player (#695)
- use rclcpp logging macros (#715)
- use rclcpp::Node for generic pub/sub (#714)
- PlayerClock initial implementation - Player functionally unchanged (#689)
- Fix bad_function_call by replacing rclcpp::spin_some with SingleThreadedExecutor (#705)
- rosbag2_py pybind wrapper for “record” - remove rosbag2_transport_py (#702)
- Add rosbag2_py::Player::play to replace rosbag2_transport_python version (#693)
- Fix and clarify logic in test_play filter test (#690)
- Explicitly add emersonknapp as maintainer (#692)
- Add QoS decoding translation for infinite durations to RMW_DURATION_INFINITE (#684)
- Add support for rmw_connextdds (#671)
- Use rosbag2_py for ros2 bag info (#673)
- Fix build issues when rosbag2_storage is binary installed (#585)
- Regex and exclude fix for rosbag recorder (#620)
- Recorder –regex and –exclude options (#604)
- SQLite storage optimized by default (#568)
- Fixed playing if unknown message types exist (#592)
- Compress bag files in separate threads (#506)
- Stabilize test_record by reducing copies of executors and messages (#576)
- add storage_config_uri (#493)
- Update the package.xml files with the latest Open Robotics maintainers (#535)
- resolve memory leak for serialized message (#502)
- Use shared logic for importing the rosbag2_transport_py library in Python (#482)
- fix missing target dependencies (#479)
- reenable cppcheck for rosbag2_transport (#461)
- More reliable topic remapping test (#456)
- Add split by time to recording (#409)
- export shared_queues_vendor (#434)

Contributors: Adam Dąbrowski, Andrea Sorbini, Chen Lihui, Dirk Thomas, Emerson Knapp, Karsten Knese, Michael Jeronimo, P. J. Reed, Piotr Jaroszek, jhdcs
rosgraph_msgs

- Change index.ros.org -> docs.ros.org. (#122)
- Updating Quality Declaration (#120)
- Update README.md (#119)
- Update quality declaration to QL 1. (#116)
- Update package maintainers. (#112)
- Increase Quality level of packages to 3 (#108)
- Add Security Vulnerability Policy pointing to REP-2006. (#106)
- Updating QD to reflect package versions (#107)
- Contributors: Chris Lalancette, Michel Hidalgo, Stephen Brawner, brawner, shonigmann

rosidl_adapter

- Expose .msg/.srv/.action to .idl conversion via rosidl translate CLI (#576)
- Support hex constants in msg files (#559)
- Treat t as whitespace (#557)
- Update the maintainers of this repository. (#536)
- Refactor regex for valid package/field names (#508)
- Add pytest.ini so tests succeed locally (#502)
- Contributors: Chris Lalancette, Dereck Wonnacott, Dirk Thomas, Michel Hidalgo

rosidl_cli

- Align rosidl_cli package version with the rest of the repo. (#579)
- Expose an API for each rosidl CLI command. (#577)
- Add rosidl translate CLI. (#575)
- Add rosidl generate CLI. (#567)
- Contributors: Michel Hidalgo, Shane Loretz

rosidl_cmake

- Shorten some excessively long lines of CMake (#571)
- Update the maintainers of this repository. (#536)
- Modifications to python generator lib to return generated files (#511)
- Contributors: Alex Tyshka, Chris Lalancette, Scott K Logan
**rosidl_default_generators**

- Update maintainers (#13)
- Contributors: Shane Loretz

**rosidl_default_runtime**

- updating quality declaration links (re: ros2/docs.ros2.org#52) (#18)
- Update QD to QL 1 (#15)
- Update maintainers (#13)
- Updated QD to 2 in README.md (#12)
- Update rosidl_default_runtime QD to QL2. (#11)
- Bump the QUALITY_DECLARATION to level 3. (#10)
- Add Security Vulnerability Policy pointing to REP-2006. (#9)
- Contributors: Alejandro Hernández Cordero, Chris Lalancette, Michel Hidalgo, Shane Loretz, Stephen Brawner, shonigmann

**rosidl_generator_c**

- Expose C code generation via rosidl generate CLI (#569)
- Strip action service suffixes from C include prefix (#538)
- Update the maintainers of this repository. (#536)
- Fix the declared language for a few packages (#530)
- Do not depend on rosidl_runtime_c when tests are disabled (#503)
- Contributors: Ben Wolsieffer, Chris Lalancette, Jacob Perron, Michel Hidalgo, Scott K Logan

**rosidl_generator_cpp**

- Expose C++ code generation via rosidl generate CLI (#570)
- Switch to std::allocator_traits. (#564)
- Remove unnecessary assert on pointer created with new (#555)
- Use ASSERT_TRUE to check for nullptr. (#543)
- Update the maintainers of this repository. (#536)
- Add to_yaml() function for C++ messages (#527)
- Add function for getting a types fully qualified name (#514)
- Declaring is_message in namespace rosidl_generator_traits (#512)
- Contributors: Chris Lalancette, Devin Bonnie, Dirk Thomas, Jacob Perron, Michel Hidalgo, Sebastian Hößner, Stephen Brawner
Vulcanexus Documentation, Release 1.0.0

**rosidl_generator_dds_idl**

- Expose .idl to DDS .idl conversion via rosidl translate CLI. (#55)
- Update maintainers. (#54)
- Contributors: Michel Hidalgo, Shane Loretz

**rosidl_generator_py**

- Remove dependency from rosidl_typesupport_connext_c (#127)
- Expose Python code generation via rosidl generate CLI (#123)
- remove maintainer (#126)
- Update maintainers (#119)
- Fix too early decref of WString when converting from Python to C (#117)
- Add pytest.ini so tests succeed locally. (#116)
- Contributors: Andrea Sorbini, Chris Lalancette, Claire Wang, Dirk Thomas, Michel Hidalgo

**rosidl_parser**

- Update and add package.xml descriptions to README (#553)
- Finish support for fixed-point literals.
- Fix parsing of small floats.
- Update the maintainers of this repository. (#536)
- Allow zero length string constants (#507)
- Add pytest.ini so tests succeed locally (#502)
- Contributors: Chris Lalancette, Dirk Thomas, Shane Loretz

**rosidl_runtime_c**

- updating quality declaration links (re: ros2/docs.ros2.org#52) (#581)
- Shorten some excessively long lines of CMake (#571)
- Update and add package.xml descriptions to README (#553)
- Fix item number in QD (#546)
- Update the maintainers of this repository. (#536)
- Add rutils dependency. (#534)
- QD: Add links to hosted API docs (#533)
- Updated Quality Level to 1 (#532)
- Add benchmarks for rosidl_runtime_* packages (#521)
- Fix the declared language for a few packages (#530)
- Add fault injection macros and test (#509)
• Update rosidl_runtime_c QD to QL 2 (#500)
• Contributors: Alejandro Hernández Cordero, Chris Lalancette, Louise Poubel, Scott K Logan, Shane Loretz, Stephen Brawner, brawner, shonigmann

**rosidl_runtime_cpp**

• updating quality declaration links (re: ros2/docs.ros2.org#52) (#581)
• Fix typo of package name in README heading (#561)
• Update and add package.xml descriptions to README (#553)
• Fix item number in QD (#546)
• Update the maintainers of this repository. (#536)
• QD: Add links to hosted API docs (#533)
• Updated Quality Level to 1 (#532)
• Add benchmarks for rosidl_runtime_* packages (#521)
• Add to_yaml() function for C++ messages (#527)
• Add function for getting a types fully qualified name (#514)
• Fix misuses of input iterators in BoundedVector (#493)
• Update QD to reflect QL 2 statuses (#499)
• Contributors: Alejandro Hernández Cordero, Chris Lalancette, Devin Bonnie, Dirk Thomas, Jacob Perron, Jonathan Wakely, Louise Poubel, Scott K Logan, Shane Loretz, Stephen Brawner, Tully Foote, shonigmann

**rosidl_runtime_py**

• Add pytest.ini so local tests don’t display warning (#12)
• Contributors: Chris Lalancette

**rosidl_typesupport_c**

• updating quality declaration links (re: ros2/docs.ros2.org#52) (#108)
• Remove dependencies from Connext type support (#106)
• Expose C typesupport generation via rosidl generate CLI (#105)
• Typo typesupport_identidentifier (#103)
• Remove type_support_dispatch.cpp files. (#101)
• Defer path resolution of rosidl typesupport libraries to dynamic linker. (#98)
• Ensure typesupport handle functions do not throw. (#99)
• Explicitly check lib pointer for null (#95)
• Update Quality Declaration to QL 1 (#96)
• Add mock for rcutils_get_symbol failure (#93)
• Update the maintainers (#89)
• Catch exception from has_symbol (#86)
• Added benchmark test to rosidl_typesupport_c/cpp (#84)
• Handle rcpputils::find_library_path() failure (#85)
• Add fault injection macros and unit tests (#80)
• Remove rethrow in extern c code (#82)
• Add Security Vulnerability Policy pointing to REP-2006 (#76)
• Contributors: Alejandro Hernández Cordero, Andrea Sorbini, Chris Lalancette, Jose Luis Rivero, Jose Tomas Lorente, Louise Poubel, Michel Hidalgo, Shane Loret, Stephen Brawner, shonigmann

rosidl_typesupport_cpp

• updating quality declaration links (re: ros2/docs.ros2.org#52) (#108)
• Remove dependencies from Connext type support (#106)
• Expose C++ typesupport generation via rosidl generate CLI (#104)
• Remove type_support_dispatch.cpp files. (#101)
• Defer path resolution of rosidl typesupport libraries to dynamic linker. (#98)
• Ensure typesupport handle functions do not throw. (#99)
• Explicitly check lib pointer for null (#95)
• Update Quality Declaration to QL 1 (#96)
• Update the maintainers (#89)
• Added benchmark test to rosidl_typesupport_c/cpp (#84)
• Handle rcpputils::find_library_path() failure (#85)
• De-duplicate type_support_map.h header (#81)
• Add fault injection macros and unit tests (#80)
• Add Security Vulnerability Policy pointing to REP-2006 (#76)
• Contributors: Alejandro Hernández Cordero, Andrea Sorbini, Chris Lalancette, Jose Luis Rivero, Louise Poubel, Michel Hidalgo, Stephen Brawner, shonigmann

rosidl_typesupport_fastrtps_c

• updating quality declaration links (re: ros2/docs.ros2.org#52) (#69)
• Expose FastRTPS C typesupport generation via rosidl generate CLI (#65)
• Update QDs with up-to-date content (#64)
• Fix item number in QD (#59)
• Update QL to 2
• Update package maintainers (#55)
• Update QD (#53)
• Fix invalid return on deserialize function (#51)
• Added benchmark test to rosidl_typesupport_fastrtps_c/cpp (#52)
• Update exec dependencies (#50)
• Add Security Vulnerability Policy pointing to REP-2006 (#44)
• QD Update Version Stability to stable version (#46)
• Contributors: Alejandro Hernández Cordero, Chris Lalancette, Jorge Perez, Louise Poubel, Michel Hidalgo, Stephen Brawner, shonigmann, sung-goo-kim

rosidl_typesupport_fastrtps_cpp

• updating quality declaration links (re: ros2/docs.ros2.org#52) (#69)
• Expose FastRTPS C++ typesupport generation via rosidl generate CLI (#66)
• Update QDs with up-to-date content (#64)
• Fix item number in QD (#59)
• Update QL to 2
• Update package maintainers (#55)
• Update QD (#53)
• Add benchmark test to rosidl_typesupport_fastrtps_c/cpp (#52)
• Update exec dependencies (#50)
• Add Security Vulnerability Policy pointing to REP-2006 (#44)
• QD Update Version Stability to stable version (#46)
• Contributors: Alejandro Hernández Cordero, Chris Lalancette, Jorge Perez, Louise Poubel, Michel Hidalgo, Stephen Brawner, shonigmann

rosidl_typesupport_interface

• updating quality declaration links (re: ros2/docs.ros2.org#52) (#581)
• Fix item number in QD (#546)
• Update the maintainers of this repository. (#536)
• QD: Add links to hosted API docs (#533)
• Update Quality Declaration to QL 1 for rosidl_typesupport_interface (#519)
• Update QD to reflect QL 2 statuses (#499)
• Contributors: Chris Lalancette, Louise Poubel, Stephen Brawner, brawner, shonigmann
Vulcanexus Documentation, Release 1.0.0

rosidl_typesupport_introspection_c

- Expose C introspection typesupport generation via rosidl generate CLI (#572)
- Update the maintainers of this repository. (#536)
- Fix get_function and get_const_function semantics for arrays (#531)
- Fix the declared language for a few packages (#530)
- Contributors: Chris Lalancette, Ivan Santiago Paunovic, Michel Hidalgo, Scott K Logan

rosidl_typesupport_introspection_cpp

- Expose C++ introspection typesupport generation via rosidl generate CLI (#573)
- Update the maintainers of this repository. (#536)
- Contributors: Chris Lalancette, Michel Hidalgo

rpyutils

- Create a shared function for importing c libraries (#4)
- Add pytest.ini so local tests don’t display warning (#3)
- Contributors: Chris Lalancette, Emerson Knapp

rqt

- 1.0.7 (#243)
- Remove Dirk from maintainers in package.xml files per request. (#236)
- Update maintainers for the crystal-devel branch (#234)
- Contributors: Michael Jeronimo, Scott K Logan

rqt_action

- Updated Open Robotics maintainer
- Fixed package to run with ros2 run (#8)
- Contributors: Alejandro Hernández Cordero, Mabel Zhang
rqt_bag

• Remove an invalid import statement (#101)
• Reset timeline zoom after loading a new bag. (#98)
• Refactor the Rosbag2 class (#91)
• Fix exec_depend (#89)
• Use updated HistoryPolicy values to avoid deprecation warnings (#88)
• Enable recording for ROS2 (#87)
• Enable the playback functionality for ROS2 (#85)
• Port the topic and node selection dialogs to ROS2 (#86)
• Save the serialization format and offered_qos_profiles when exporting (#84)
• Enable the export/save bag functionality for ROS2 (#82)
• Update known message types and associated colors (#81)
• Open the bag directory instead of a single file (#80)
• Port the image_view plugin to ROS2 (#78)
• Clean up widgets in plot_view layout correctly (#69) (#77)
• Fix tuples for bisect calls (#67) (#76)
• Fix issue: no vertical scroll bar until window is resized (#63) (#75)
• Update the basic plugins for ROS2 (#72)
• Update the rosbag2 python module (#71)
• Dynamically resize the timeline when recording (#66)
• Starting point for resuming the ROS2 port (#70)
• Fix a bug with the status line progress bar (#62)
• Update a few minor status bar-related items (#61)
• Make the tree controls in the Raw View and Plot View consistent (#57)
• Update the package.xml files with the latest Open Robotics maintainers (#58)
• fix Python 3 issue: long/int (#52)
• save last directory opened to load a bag file (#40)
• fix shebang line for Python 3 (#48)
• bump CMake minimum version to avoid CMP0048 warning
• fix Python 3 exception, wrap filter call in list() (#46)
• add Python 3 conditional dependencies (#44)
• autopep8 (#30)
• add support for opening multiple bag files at once (#25)
• fix debug/warning messages for unicode filenames (#26)
• fix regression from version 0.4.10 (#17)
• fix regression from version 0.4.9 (#16)
- handle errors happening while loading a bag (#14)
- add rqt_bag.launch file (#440)
- fix Python 2 regression from version 0.4.4 (#424)
- use Python 3 compatible syntax (#421)
- fix race condition reading bag files (#412)
- add “From nodes” button to record mode (#348)
- show file size of bag file in the status bar (#347)
- fix mouse wheel delta in Qt 5 (#376)
- Support Qt 5 (in Kinetic and higher) as well as Qt 4 (in Jade and earlier) (#359)
- fix publishing wrong topic after scrolling (#362)
- RQT_BAG: Ensure monotonic clock publishing. Due to parallelism issues, a message can be published with a simulated timestamp in the past. This lead to undesired behaviors when using TF for example.
- Added step-by-step playback capability
- fix viewer plugin relocation issue (#306)
- fix topic type retrieval for multiple bag files (#279)
- fix region_changed signal emission when no start/end stamps are set
- improve right-click menu
- improve popup management (#280)
- implement recording of topic subsets
- sort the list of topics
- update plugin scripts to use full name to avoid future naming collisions
- fix visibility with dark Qt theme (#263)
- fix compatibility with Groovy, use queue_size for Python publishers only when available (#243)
- use thread for loading bag files, emit region changed signal used by plotting plugin (#239)
- export architecture_independent flag in package.xml (#254)
- fix closing and reopening topic views
- use queue_size for Python publishers
- fix raw view not showing fields named ‘msg’ (#226)
- add option to publish clock tim from bag (#204)
- add groups for rqt plugins, renamed some plugins (#167)
- fix high cpu load when idle (#194)
- update rqt_bag plugin interface to work with qt_gui_core 0.2.18
- fix rendering of icons on OS X (ros-visualization/rqt#83)
- fix shutdown of plugin (#31)
- fix saving parts of a bag (#96)
- fix long topic names (#114)
• fix zoom behavior (#76)
• Fix; skips time when resuming playback (#5)
• Fix; timestamp printing issue (#6)
• expose command line arguments to rqt_bag script
• added fix to set play/pause button correctly when fastforwarding/rewinding, adjusted time headers to 0m00s instead of 0:00m for ease of reading
• support passing bagfiles on the command line (currently behind –args)
• first release of this package into Groovy
• Contributors: Aaron Blasdel, Chris Lalancette, Michael Grupp, Michael Jeronimo, lsouchet, sambrose

rqt_bag_plugins

• Refactor the Rosbag2 class (#91)
• Port the plot view to ROS2 (#79)
• Port the image_view plugin to ROS2 (#78)
• Starting point for resuming the ROS2 port (#70)
• Make the tree controls in the Raw View and Plot View consistent (#57)
• Update the package.xml files with the latest Open Robotics maintainers (#58)
• initialize pil_mode when image is compressed (#54)
• support 16-bit bayer images (#45)
• maintain image aspect ratio (#32)
• fix Python 3 issue: long/int (#51)
• fix Python 3 issue: ensure str is encoded before decoding (#50)
• bump CMake minimum version to avoid CMP0048 warning
• add Python 3 conditional dependencies (#44)
• add cairocffi as the fallback module (#43)
• autopep8 (#30)
• fix Python 2 regression from version 0.4.4 (#426)
• use Python 3 compatible syntax (#421)
• fix crash when toggling thumbnail (#380)
• lock bag when reading for plotting (#382)
• Support Qt 5 (in Kinetic and higher) as well as Qt 4 (in Jade and earlier) (#359)
• add missing dependency on rqt_plot (#316)
• work around Pillow segfault if PyQt5 is installed (#289, #290)
• add displaying of depth image thumbnails
• add missing dependency on python-cairo (#269)
• fix missing installation of resource subfolder
• add plotting plugin (#239)
• fix rqt_bag to plot array members (#253)
• export architecture_independent flag in package.xml (#254)
• fix PIL/Pillow error (#224)
• first release of this package into Groovy
• Contributors: John Stechschulte, Michael Jeronimo

rqt_console

• Use underscores in setup.cfg instead of dashes (#31)
• Fix regression introduced in #21 (#28)
• Changed the build type to ament_python and added setup.cfg (#21)
• Contributors: Alejandro Hernández Cordero, Michel Hidalgo

rqt_graph

• Make topics that have qos incompatibilities red in the graph, add information to the node tooltip (#61)
• Add node name, topic name, and endpoint kind to the qos edge tooltip (#60)
• Update maintainers for ROS2 branches (#55)
• add edge tooltip with QoS of publishers and subscribers (#53)
• install executable rqt_graph (#49)
• add setup.cfg with script install directories (#42)
• add pytest.ini so local tests don’t display warning (#48)
• Contributors: Ivan Santiago Paunovic, Michael Jeronimo

rqt_gui

• getiterator() renamed to iter() in Python 3.9 (#239)
• Contributors: goekce

rqt_gui_cpp

• use tgt compile features (#247)
• Contributors: Audrow Nash
rqt_gui_py

- Fix a crash at shutdown (#248)
- Contributors: Michael Jeronimo

rqt_msg

- Changed the build type to ament_python and fixed package to run with ros2 run (#8)
- Use rosidl_runtimetype_py instead of message_helpers where possible (#11)
- Contributors: Alejandro Hernández Cordero, Ivan Santiago Paunovic

rqt_plot

- Changed the build type to ament_python and fixed package to run with ros2 run (#58)
- Fix plots of array items (#71)
- Update maintainers
- Contributors: Alejandro Hernández Cordero, Ivan Santiago Paunovic, Mabel Zhang

rqt_publisher

- Changed the build type to ament_python and fixed package to run with ros2 run (#18)
- Drop numpy.float128 references (#26)
- Use rosidl_runtime_py instead of rqt_py_common where possible (#24)
- Add now() to evaluation (#22)
- fix setting expressions on sequence items (#16)
- Contributors: Alejandro Hernández Cordero, Ivan Santiago Paunovic, Michel Hidalgo, Yossi Ovcharik

rqt_py_common

- Avoid installing test interfaces (#228)
- Contributors: Dirk Thomas

rqt_py_console

- Changed the build type to ament_python and fixed package to run with ros2 run (#8)
- Contributors: Alejandro Hernández Cordero
rqt_reconfigure

- Cleanups to the install scripts. (#103)
- Fix a flake8 warning. (#99)
- Use timeouts in service calls to avoid hangs (#98)
- Add maintainer to package.xml (#95)
- Save instance state in rqt settings (#90)
- Use safe YAML loader (#89)
- Don’t process scroll events unless specifically focused (#88)
- Fix node selection from command line (#87)
- Add pytest.ini so local tests don’t display warning (#91)
- Support PEP 338 invocation of rqt_reconfigure (#85)
- Fixed package to run with ros2 run (#81)
- Contributors: Alejandro Hernández Cordero, Chris Lalancette, Michael Jeronimo, Scott K Logan

rqt_service_caller

- Changed the build type to ament_python and fixed package to run with ros2 run (#13)
- ignore services that don’t use the SRV_MODE (‘srv’) (#20)
- Contributors: Alejandro Hernández Cordero, William Woodall

rqt_shell

- Changed the build type to ament_python and fixed package to run with ros2 run (#11)
- Contributors: Alejandro Hernández Cordero

rqt_srv

- Changed the build type to ament_python and fixed package to run with ros2 run (#4)
- Contributors: Alejandro Hernández Cordero

rqt_top

- Changed the build type to ament_python and fixed package to run with ros2 run (#8)
- Contributors: Alejandro Hernández Cordero
rqt_topic

- Add pytest.ini to silence warnings when running locally.
- Fix warnings pointed out by flake8.
- Created an entry-point for rqt_topic in setup.py (#16)
- Fix flake8 errors and add linter tests (#28)
- Update Open Robotics Maintainer (#26)
- Use raw / non-string value for ordering (#23)
- Support order fields as defined in message (#22)
- Fix the type cell value for sequence items (#21)
- Updated version package and license in setup.py (#17)
- Contributors: Alejandro Hernández Cordero, Chris Lalancette, Dirk Thomas, Scott K Logan

rti_connext_dds_cmake_module

- Pass -Wl,--no-as-needed for system dependencies of Connext 5.3.1.
- Set IMPORTED_NO_SONAME true for Connext 5.3.1 imported library target.
- Add <buildtool_export_depend> for ament_cmake.
- Add <depend> for rti-connext-dds-5.3.1
- Add dependency from rti-connext-dds-5.3.1.
- Initial release.

rttest

- Fix up nonsensical handling of NULL in rttest_get_{params,statistics} (#107)
- Remove “struct” from rttest_sample_buffer variable declaration. (#105)
- Convert the sample buffer to a vector. (#104)
- Use strdup instead of strlen/strcpy dance. (#100)
- Enable basic warnings in rttest (#99)
- Only copy an rttest_sample_buffer if it is not nullptr. (#98)
- Convert timespec to uint64 not long and vice versa (#94) (#96)
- Fix standard deviation overflow(#95) (#97)
- Contributors: Audrow Nash, Chris Lalancette, y-okumura-isp
rviz2

- Use “%s” as format string literal in logging macros (#633)
- Add linters and use ament_lint_auto (#608)
- Update maintainers (#607)
- Move and update documentation for ROS 2 (#600)
- Contributors: Audrow Nash, Chris Lalancette, Jacob Perron

rviz_assimp_vendor

- Always preserve source permissions in vendor packages (#647)
- Add an override flag to force vendored build (#642)
- Add linters and use ament_lint_auto (#608)
- Update maintainers (#607)
- Updated a hack to avoid CMake warning with assimp 5.0.1 and older, applying it cross platforms (#565)
- Contributors: Dirk Thomas, Jacob Perron, Scott K Logan

rviz_common

- Add visualization_frame to the public API (#660)
- Add ViewPicker::get3DPatch to the public API (#657)
- Fix byte indexing for depth patch pixels (#661)
- fix toolbar vanishing when pressing escape (#656)
- Expose VisualizationManager and YamlConfigReader to the public API (#649)
- Use the stack for the classes in the property test. (#644)
- Check that the views_man_ and views_man_->getCurrent() are not nullptr. (#634)
- Fix for mousewheel to zoom in/out (#623)
- Ensure rviz_common::MessageFilterDisplay processes messages in the main thread (#620)
- Fix render window disppearing after saving image (#611)
- Add linters and use ament_lint_auto (#608)
- Update maintainers (#607)
- TimePanel port (#599)
- Upgrade to tinyxml2 for rviz (#418)
- Fix segfault on changing filter size for non-existent topic (#597)
- improve color support for themes (#590)
- Fix topic IntProperty number ranges (#596)
- Switch to nullptr everywhere. (#592)
- Expose MessageFilterDisplay’s queue size (#593)
• Filter topics in drop down menu (#591)
• rviz_common: Remove variadic macro warning check (#421)
• Use retriever.hpp (#589)
• Fix the order of destructors (#572)
• Changed to not install test header files in rviz_rendering. (#564)
• Fixed alphabetical include order (#563)
• Changed to avoid trying to moc generate env_config.hpp file. (#550)
• Contributors: Audrow Nash, Chen Lihui, Chris Lalancette, Jacob Perron, Jafar Abdi, Joseph Schornak, Karsten Knese, Martin Idel, Michael Ferguson, Michael Jeronimo, Michel Hidalgo, Nico Neumann, Rich Mattes, Shane Loretz, ipa-fez, spiralray

rviz_default_plugins

• Add ViewPicker::get3DPatch to the public API (#657)
• Allow to zoom more with orbit controller (#654)
• Fix possible nullptr access in robot_joint.cpp. (#636)
• Fix for mousewheel to zoom in/out (#623)
• Make the types explicit in quaternion_helper.hpp. (#625)
• Update status message by removing colon or adjust colon position (#624)
• Do not use assume every RenderPanel has a ViewController. (#613)
• Add linters and use ament_lint_auto (#608)
• Update maintainers (#607)
• TimePanel port (#599)
• Upgrade to tinyxml2 for rviz (#418)
• Use retriever.hpp (#589)
• Added covariance settings to set pose estimate (#569)
• use reference in range loops to avoid copy (#577)
• Changed to not install test header files in rviz_rendering. (#564)
• Changed to use a dedicated TransformListener thread. (#551)
• Suppressed warnings when building with older Qt versions. (#562)
• Restored compatibility with older Qt versions (#561)
• Contributors: Chen Lihui, Chris Lalancette, Dirk Thomas, Jacob Perron, Joseph Schornak, Martin Idel, Matthijs den Toom, Michel Hidalgo, Nico Neumann, Shane Loretz, Victor Lamoine, ymd-stella
rviz_ogre_vendor

- Always preserve source permissions in vendor packages (#647)
- Add linters and use ament_lint_auto (#608)
- Update maintainers (#607)
- Pass through CMAKE_{C,CXX}_FLAGS to OGRE build (#587)
- Contributors: Jacob Perron, Scott K Logan

rviz_rendering

- Reset current line width when calculating text width (#655)
- Silence a dead store warning. (#643)
- Fix a memory leak when using the ResourceIOSystem. (#641)
- Revert “Support loading meshes other than .mesh and .stl with package URIs (#610)” (#638)
- Prevent rviz_rendering::AssimpLoader from loading materials twice. (#622)
- Support loading meshes other than .mesh and .stl with package URIs (#610)
- Add linters and use ament_lint_auto (#608)
- Update maintainers (#607)
- Switch to nullptr everywhere. (#592)
- Use retriever.hpp (#589)
- Avoid hiding base class getRenderOperation in PointCloudRenderable (#586)
- Changed to not install test header files in rviz_rendering. (#564)
- Contributors: Chris Lalancette, Ivan Santiago Paunovic, Jacob Perron, Michel Hidalgo, Shane Loretz, ipa-fez

rviz_rendering_tests

- Add linters and use ament_lint_auto (#608)
- Update maintainers (#607)
- Use retriever.hpp (#589)
- Changed to not install test header files in rviz_rendering. (#564)
- Contributors: Chris Lalancette, Jacob Perron, Shane Loretz
rviz_visual_testing_framework

- Quiet a clang warning about a Qt memory leak. (#651)
- use rcutils_get_env. (#609)
- Add linters and use ament_lint_auto (#608)
- Update maintainers (#607)
- Contributors: Chris Lalancette, Jacob Perron, tomoya

sensor_msgs

- Change index.ros.org -> docs.ros.org. (#149)
- updating quality declaration links (re: ros2/docs.ros2.org#52) (#145)
- Fix PointCloud2Iterator namespace in docs (#139)
- Add coverage/performance to qd for sensor_msgs (#137)
- Update QDs to QL 1 (#135)
- Update package maintainers. (#132)
- Updated Quality Level to 2 (#131)
- Missing cstring header for memcpy in fill_image.hpp (#126)
- Update Quality levels to level 3 (#124)
- Add Security Vulnerability Policy pointing to REP-2006. (#120)
- Contributors: Alejandro Hernández Cordero, Andre Nguyen, Chris Lalancette, Jose Luis Rivero, Michel Hidalgo, Stephen Brawner, brawner, shonigmann

sensor_msgs_py

- Use underscores instead of dashes in setup.cfg (#150)
- Port of point_cloud2.py from ROS1 to ROS2. As separate pkg. (#128)
- Contributors: Ivan Santiago Paunovic, Sebastian Grans

shape_msgs

- Change index.ros.org -> docs.ros.org. (#149)
- updating quality declaration links (re: ros2/docs.ros2.org#52) (#145)
- Update QDs to QL 1 (#135)
- Update package maintainers. (#132)
- Updated Quality Level to 2 (#131)
- Update Quality levels to level 3 (#124)
- Add Security Vulnerability Policy pointing to REP-2006. (#120)
- Contributors: Alejandro Hernández Cordero, Chris Lalancette, Michel Hidalgo, Stephen Brawner, brawner, shonigmann

**shared_queues_vendor**
- Explicitly add emersonknapp as maintainer (#692)
- Update the package.xml files with the latest Open Robotics maintainers (#535)
- Contributors: Emerson Knapp, Michael Jeronimo

**spdlog_vendor**
- updating quality declaration links (re: ros2/docs.ros2.org#52) (#24)
- Update to spdlog 1.8.2 (#23)
- Remove a stale TODO (#22)
- Always preserve source permissions in vendor packages (#20)
- Remove unnecessary call to find_package(PATCH) (#18)
- Updated QD to 1 (#16)
- bump spdlog version to 1.6.1 (#15)
- Bump QD to level 3 and updated QD (#14)
- Add Security Vulnerability Policy pointing to REP-2006. (#13)
- Contributors: Alejandro Hernández Cordero, Michael Jeronimo

**sqlite3_vendor**
- Explicitly add emersonknapp as maintainer (#692)
- Always preserve source permissions in vendor packages (#645)
- Update the package.xml files with the latest Open Robotics maintainers (#535)
- use interface_include_directories (#426)
- Contributors: Emerson Knapp, Karsten Knese, Michael Jeronimo, Scott K Logan

**sros2**
- Skip mypy test on platforms using importlib_resources (#258)
- Enable topic “ros_discovery_info” for rmw_connextdds (#253)
- Declare missing dependency on python3-importlib-resources (#249) Co-authored-by: <mikael.arguedas@gmail.com>
- Fix namedtuple names. (#250)
- parameter_events topic is now absolute (#233) Signed-off-by: Mikael Arguedas <mikael.arguedas@gmail.com>
- Expose keystore operations in public API (#241)
- add cyclonedds to the list of rmw using graph info topics (#231)
- Add scope parameter (#230)
- Fix name of argument passed to NodeStrategy (#227)
- Remove the use of pkg_resources. (#225)
- Make use of ros_testing to test policy generation. (#214)
- Add pytest.ini so local tests don’t display warning (#224)
- Fix list keys verb (#219)
- Contributors: Andrea Sorbini, Chris Lalancette, Jacob Perron, Jose Luis Rivero, Kyle Fazzari, Michel Hidalgo, Mikael Arguedas, Scott K Logan

**statistics_msgs**

- Updating Quality Declaration (#120)
- Update quality declaration to QL 1. (#116)
- Update package maintainers. (#112)
- Increase Quality level of packages to 3 (#108)
- Add Security Vulnerability Policy pointing to REP-2006. (#106)
- Updating QD to reflect package versions (#107)
- Contributors: Chris Lalancette, Michel Hidalgo, Stephen Brawner, brawner, shonigmann

**std_msgs**

- Change index.ros.org -> docs.ros.org. (#149)
- updating quality declaration links (re: ros2/docs.ros2.org#52) (#145) Co-authored-by: Simon Honigmann <shonigmann@blueorigin.com>
- Update QDs to QL 1 (#135)
- Update package maintainers. (#132)
- Updated Quality Level to 2 (#131)
- Update Quality levels to level 3 (#124)
- Add Security Vulnerability Policy pointing to REP-2006. (#120)
- Contributors: Alejandro Hernández Cordero, Chris Lalancette, Michel Hidalgo, Stephen Brawner, brawner, shonigmann
std_srvs

- Change index.ros.org -> docs.ros.org. (#149)
- updating quality declaration links (re: ros2/docs.ros2.org#52) (#145)
- Update QDs to QL 1 (#135)
- Update package maintainers. (#132)
- Updated Quality Level to 2 (#131)
- Update Quality levels to level 3 (#124)
- Add Security Vulnerability Policy pointing to REP-2006. (#120)
- Contributors: Alejandro Hernández Cordero, Chris Lalancette, Michel Hidalgo, Stephen Brawner, brawner, shonigmann

stereo_msgs

- Change index.ros.org -> docs.ros.org. (#149)
- updating quality declaration links (re: ros2/docs.ros2.org#52) (#145)
- Update QDs to QL 1 (#135)
- Update package maintainers. (#132)
- Updated Quality Level to 2 (#131)
- Update Quality levels to level 3 (#124)
- Add Security Vulnerability Policy pointing to REP-2006. (#120)
- Contributors: Alejandro Hernández Cordero, Chris Lalancette, Michel Hidalgo, Stephen Brawner, brawner, shonigmann

tango_icons_vendor

- Add exec_depend on tango-icon-theme system package (#8)
- Added common linters (#7) * Added common linters * Fixed license in package.xml
- Remaned package qt_gui_icons -> tango_icons_vendor (#4) * Remaned package qt_gui_icons -> tango_icons_vendor * Updated CMake var to install tango icons: INSTALL_TANGO_ICONS * Added cmake option INSTALL_TANGO_ICONS * Fixed logic * set INSTALL_TANGO_ICONS_DEFAULT_VALUE to option * Make linters happy
- Updated link on the description (#6)
- Updated the maintainer (#5)
- Version 0.0.0 this package was never released (#3)
- Install icons by default on macOS too (#1)
- Updating package.xml
- fixup! Install tango icons
- Adding icons
- Install tango icons

450 Chapter 4. Contributing to the documentation
• Contributors: Alejandro Hernández Cordero, Scott K Logan, Stephen, Stephen Brawner

test_cli

• Update maintainers. (#450)
• Enable -Wall, -Wextra, and -Wpedantic. (#447)
• Contributors: Audrow Nash, Jacob Perron

test_cli_remapping

• Fix test_cli_remapping flaky test. (#470)
• Update maintainers. (#450)
• Enable -Wall, -Wextra, and -Wpedantic. (#448)
• Contributors: Audrow Nash, Jacob Perron, Shane Loretz

test_communication

• Add support for rmw_connextdds. (#463)
• Kill off the ros2 daemon before running tests. (#460)
• Remove Opensplice from test_communication. (#460)
• Make TestMessageSerialization robust to missed messages. (#456)
• Add corresponding rclcpp::shutdown. (#455)
• Update maintainers. (#450)
• Contributors: Andrea Sorbini, Chris Lalancette, Jacob Perron, Stephen Brawner

test_interface_files

• Update maintainer (#13)
• Contributors: Jacob Perron

test_launch_ros

• Add a package marker to test_launch_ros. (#226)
• Re-order shutdown vs node destruction (#213)
• Increase test_composable_node_container timeout (#195)
• Remove constructors arguments deprecated since Foxy (#190)
• Merge pull request #183 from ros2/update-maintainers
• Move previous maintainer to <author>
• Update the package.xml files with the latest Open Robotics maintainers
• Handle any substitution types for SetParameter name argument (#182)
• Address security bug in yaml loading (#175)
• Resolve TODO in test (#172)
• Fix case where list of composable nodes is zero (#173)
• Do not use event handler for loading composable nodes (#170)
• Fix race with launch context changes when loading composable nodes (#166)
• Substitutions in parameter files (#168)
• Fix problems when parsing a Command Substitution as a parameter value (#137)
• Drop double single-quoted params. (#164)
• Add a way to set remapping rules for all nodes in the same scope (#163)
• Fix ComposableNode ignoring PushRosNamespace actions (#162)
• Add a SetParameter action that sets a parameter to all nodes in the same scope (#158)
• Make namespace parameter mandatory in LifecycleNode constructor (#157)
• Avoid using a wildcard to specify parameters if possible (#154)
• Remove the loop parameter from async.sleep. (#155)
• Fix no specified namespace (#153)
• Fix test_node_frontend (#146)
• Add pytest.ini so local tests don’t display warning (#152)
• Contributors: Chris Lalancette, Dan Rose, Ivan Santiago Paunovic, Jacob Perron, Michael Jeronimo, Michel Hidalgo, Scott K Logan, Víctor Mayoral Vilches

test_launch_testing

• Update package maintainers (#465)
• Add pytest.ini to test_launch_testing so tests succeed locally. (#431)
• Contributors: Chris Lalancette, Michel Hidalgo

test_msgs

• Update package maintainers. (#112)
• Contributors: Chris Lalancette
test_quality_of_service

- Add support for rmw_connextdds. (#463)
- Run QoS tests. (#441)
- Update maintainers. (#450)
  - Contributors: Andrea Sorbini, Jacob Perron, Michel Hidalgo

**test_rclcpp**

- Reenable test that used to be flaky. (#467)
- Get_parameters_service_ should return empty if allow_undeclared_ is false. (#466)
- Make test pass after rclcpp#1532. (#465)
- Adapt tests to statically typed parameters. (#462)
- Guard against TOCTTOU with rclcpp::ok and rclcpp::spin_some. (#459)
- Update parameter client test with timeout. (#457)
- Call rclcpp::init and rclcpp::shutdown in each test for test_rclcpp. (#454)
- Set cppcheck timeout to 400 seconds. (#453)
- Modify to match Waitable interface adding take_data. (#444)
- Update maintainers. (#450)
- Show numbers of nanoseconds in EXPECT with durations. (#438) * Show numbers of nanoseconds in expect with durations * Fix syntax
- Remove ament_pytest dependency from test_rclcpp. (#437) It is not used in test_rclcpp anywhere.
- Contributors: Audrow Nash, Chris Lalancette, Dirk Thomas, Ivan Santiago Paunovic, Jacob Perron, Michel Hidalgo, Shane Loretz, Stephen Brawner, Tomoya Fujita, tomoya

**test_rmw_implementation**

- Implement test for subscription loaned messages (#186)
- Remove rmw_connext_cpp. (#183)
- Add support for rmw_connextdds (#182)
- Add function for checking QoS profile compatibility (#180)
- Make sure to initialize the rmw_message_sequence after init. (#175)
- Set the value of is_available before entering the loop (#173)
- Set the return value of rmw_ret_t before entering the loop. (#171)
- Add some additional checking that cleanup happens. (#168)
- Add test to check rmw_send_response when the client is gone (#162)
- Update maintainers (#154)
- Add fault injection tests to construction/destroy APIs. (#144)
• Add tests bad type_support implementation (#152)
• Add tests for localhost-only node creation (#150)
• Added rmw_service_server_is_available tests (#140)
• Use 10x the intraprocess delay to wait for sent requests. (#148)
• Added rmw_wait, rmw_create_wait_set, and rmw_destroy_wait_set tests (#139)
• Add tests service/client request/response with bad arguments (#141)
• Added test for rmw_get_serialized_message_size (#142)
• Add service/client construction/destruction API test coverage. (#138)
• Added rmw_publisher_allocation and rmw_subscription_allocation related tests (#137)
• Add tests take serialized with info bad arguments (#130)
• Add gid API test coverage. (#134)
• Add tests take bad arguments (#125)
• Bump graph API test coverage. (#132)
• Add tests take sequence serialized with bad arguments (#129)
• Add tests take sequence + take sequence with bad arguments (#128)
• Add tests take with info bad arguments (#126)
• Add tests for non-implemented rmw_take_* functions (#131)
• Add tests publish serialized bad arguments (#124)
• Add tests publish bad arguments (#123)
• Add tests non-implemented functions + loan bad arguments (#122)
• Add missing empty topic name tests. (#136)
• Add rmw_get_serialization_format() smoke test. (#133)
• Complete publisher/subscription QoS query API test coverage. (#120)
• Remove duplicate assertions (#121)
• Add publisher/subscription matched count API test coverage. (#119)
• Add serialize/deserialize API test coverage. (#118)
• Add subscription API test coverage. (#117)
• Extend publisher API test coverage (#115)
• Add node construction/destruction API test coverage. (#112)
• Check that rmw_init() fails if no enclave is given. (#113)
• Add init options API test coverage. (#108)
• Complete init/shutdown API test coverage. (#107)
• Add dependency on ament_cmake_gtest (#109)
• Add test_rmw_implementation package. (#106)

Contributors: Alejandro Hernández Cordero, Andrea Sorbini, Chris Lalancette, Geoffrey Biggs, Ivan Santiago Paunovic, Jacob Perron, Jose Tomas Lorente, José Luis Bueno López, Michel Hidalgo, Miguel Company, Shane Loretz
test_security

- Add support for rmw_connextdds. (#463)
- Update deprecated gtest macros. (#449)
- Update maintainers. (#450)
- Run test_security on CycloneDDS as well. (#408)
- Remove invalid cert folder to force regeneration of certificates. (#434)
- Contributors: Andrea Sorbini, Audrow Nash, Jacob Perron, Mikael Arguedas

test_tf2

- Update maintainers of the ros2/geometry2 fork. (#328)
- Activate usual compiler warnings and fix errors (#270)
- Fix a TOCTTOU race in tf2. (#307)
- Fixed memory leak in Buffer::waitForTransform (#281)
- relax test timings to pass with Connext (#304)
- Explicitly initialize instances of tf2::Duration (#291)
- Generate callbacks after updating message_ (#274)
- fix test_static_publisher in macos (#284)
- Fix up the dependencies in test_tf2. (#277)
- Split tf2_ros in tf2_ros and tf2_ros_py (#210)
- Contributors: Alejandro Hernández Cordero, Chris Lalancette, Dirk Thomas, Ivan Santiago Paunovic, Martin Ganeff, Michael Carroll, ymd-stella

tf2

- Change index.ros.org -> docs.ros.org. (#394)
- Update maintainers of the ros2/geometry2 fork. (#328)
- Active usual compiler warnings in tf2 (#322)
- Cleanups in buffer_core.cpp. (#301)
- Add PoseWithCovarianceStamped transform support (#312)
- Fix a TOCTTOU race in tf2. (#307)
- Fixed memory leak in Buffer::waitForTransform (#281)
- Add common linters to tf2. (#258)
- Provide more available error messaging for nonexistent and invalid frames in canTransform (ros2 #187)
- Contributors: Alejandro Hernández Cordero, Chris Lalancette, Ivan Santiago Paunovic, Joshua Whitley, Martin Ganeff
tf2_bullet

• Update maintainers of the ros2/geometry2 fork. (#328)
• Activate usual compiler warnings and fix errors (#270)
• Suppress compiler warning on Centos (#290)
• Contributors: Chris Lalancette, Ivan Santiago Paunovic, Michael Carroll

tf2_eigen

• Fix linter errors (#385)
• Fix up the style in tf2_eigen. (#378)
• Fix doTransform with Eigen Quaternion (#369)
• Update maintainers of the ros2/geometry2 fork. (#328)
• Activate usual compiler warnings and fix errors (#270)
• Contributors: Audrow Nash, Bjar Ne, Chris Lalancette, Ivan Santiago Paunovic

tf2_eigen_kdl

• fix order of find eigen3 cmake_module & find eigen3 (#344)
• Update package.xml (#333)
• Port eigen_kdl.h/cpp to ROS2 (#311)
• Contributors: Ahmed Sobhy, Jafar Abdi

tf2_geometry_msgs

• Fix doTransform with Eigen Quaternion (#369)
• Update maintainers of the ros2/geometry2 fork. (#328)
• Activate usual compiler warnings and fix errors (#270)
• Add PoseWithCovarianceStamped transform support (#312)
• Don’t install python tf2_geometry_msgs (#299) It hasn’t been ported yet. Closes https://github.com/ros2/geometry2/issues/285
• Split tf2_ros in tf2_ros and tf2_ros_py (#210) * Split tf2_ros in tf2_ros and tf2_ros_py
• Contributors: Alejandro Hernández Cordero, Bjar Ne, Chris Lalancette, Ivan Santiago Paunovic, Joshua Whitley, Shane Loretz
tf2_kdl

- Update maintainers of the ros2/geometry2 fork. (#328)
- Activate usual compiler warnings and fix errors (#270)
- Split tf2_ros in tf2_ros and tf2_ros_py (#210)
- Contributors: Alejandro Hernández Cordero, Chris Lalancette, Ivan Santiago Paunovic

tf2_msgs

- Update maintainers of the ros2/geometry2 fork. (#328)
- Activate usual compiler warnings and fix errors (#270)
- Contributors: Chris Lalancette, Ivan Santiago Paunovic

tf2_py

- Adapt to Python 3.9 (#362)
- Update maintainers of the ros2/geometry2 fork. (#328)
- Add in pytest.ini so tests succeed locally. (#280)
- Contributors: Chris Lalancette, Homalozoa X

tf2_ros

- Guard against access to null node pointer (#393)
- Allow to reconfigure durability for /tf topic broadcaster/listener (#383)
- Fix the rcl type used in the time jump. (#391)
- Fix linter errors (#385)
- fix accessing freed resources (#386)
- Allow reconfiguring qos of tf and tf_static topics through parameters (#381)
- Replace ROS_* logging macros and use RCLCPP_* instead (#380)
- Improve message filters error messages (#364)
- Clarify the role of child_frame_id and header.frame_id in the documentation. (#345)
- Remove usage of deprecated rclcpp::Duration constructor (#340)
- Remove messages_count member from tf2_ros::MessageFilter. (#335)
- Style fixup in tf2_ros. (#325)
- Update maintainers of the ros2/geometry2 fork. (#328)
- Update goal response callback signature (#323)
- Activate usual compiler warnings and fix errors (#270)
- Fixed memory leak in Buffer::waitForTransform (#281)
- fix time-reset test with Connext (#306)
• reenable FrameGraph server (#198)
• Use the usual style of parameters for static_transform_program (#300)
• Make static_transform_broadcaster consistent with its command line description (#294)
• Avoid using invalid std::list iterators (#293)
• Generate callbacks after updating message_ (#274)
• Moved unique_lock of messages_mutex_ to guarantee pointer (#279)
• Fix dependencies in tf2_ros. (#269)
• Split tf2_ros in tf2_ros and tf2_ros_py (#210)
• Contributors: Alejandro Hernández Cordero, Audrow Nash, Chris Lalancette, Dirk Thomas, Hunter L. Allen, Ivan Santiago Paunovic, Jacob Perron, Kazunari Tanaka, Martin Ganeff, Michael Carroll, Vikas Dhiman, ymd-
stella

**tf2_ros_py**

• Use underscores instead of dashes in setup.cfg. (#403) (#404)
• Use global namespace for TransformListener topics (#390)
• Fix indentation of a comment in buffer.py (#371)
• Update rclpy.Rate TODO with url to issue (#324)
• Update maintainers of the ros2/geometry2 fork. (#328)
• Add deprecation warnings to lookup_transform to handle the passing of the incorrect Time object. (#319)
• change signature to show true arguments (#321)
• Handle when None passed to qos argument in the constructor of TransformBroadcaster. (#320)
• Add type hints to tf2_ros_py code(#275) (#315)
• Clear callbacks_to_remove variable after removing (#303)
• Fix cache_time None check in buffer.py (#297)
• Split tf2_ros in tf2_ros and tf2_ros_py (#210)
• Contributors: Alejandro Hernández Cordero, Chris Lalancette, Jacob Perron, Matthijs den Toom, ScottMcMichael, surfertas

**tf2_sensor_msgs**

• Update maintainers of the ros2/geometry2 fork. (#328)
• Activate usual compiler warnings and fix errors (#270)
• Split tf2_ros in tf2_ros and tf2_ros_py (#210)
• Contributors: Alejandro Hernández Cordero, Chris Lalancette, Ivan Santiago Paunovic

Chapter 4. Contributing to the documentation
tf2_tools

- Use underscores instead of dashes in setup.cfg. (#403) (#404)
- Add wait time option to view_frames (#374)
- Cleanup tf2_tools to be more modern. (#351)
- Update maintainers of the ros2/geometry2 fork. (#328)
- Address security bug in yaml loading (#313)
- Split tf2_ros in tf2_ros and tf2_ros_py (#210)
- Contributors: Alejandro Hernández Cordero, Chris Lalancette, Jacob Perron, Víctor Mayoral Vilches

tlsf

- Switch to standard __VA_ARGS__. (#9)
- Enable basic warnings (#8)
- Contributors: Audrow Nash, Chris Lalancette

tlsf_cpp

- Add in the Apache license to tlsf_cpp. (#108)
- Contributors: Chris Lalancette

topic_monitor

- Use is_alive for threads. (#510) (#513)
- Use underscores instead of dashes in setup.cfg (#502)
- Change index.ros.org -> docs.ros.org. (#496)
- Update deprecated qos policy value names (#468)
- Update the package.xml files with the latest Open Robotics maintainers (#466)
- Contributors: Chris Lalancette, Ivan Santiago Paunovic, Michael Jeronimo

topic_statistics_demo

- Change index.ros.org -> docs.ros.org. (#496)
- Update logging macros (#476)
- Update the package.xml files with the latest Open Robotics maintainers (#466)
- Create new topic statistics demo package (#454)
- Contributors: Audrow Nash, Chris Lalancette, Michael Jeronimo, Prajakta Gokhale
tracetools

- Update QD to be more specific about public API
- Namespace tracetools C++ functions and macros and deprecate current ones
- Add support for rcl_publish and rclcpp_publish tracepoints
- Add instrumentation support for linking a timer to a node
- Bring tracetools up to quality level 1
- Add lifecycle node state transition instrumentation
- Do not export tracetools if empty
- Allow disabling tracetools status app
- Contributors: Christophe Bedard, Ingo Lütkebohle, José Antonio Moral

tracetools_launch

- Allow configuring tracing directory through environment variables
- Contributors: Christophe Bedard

tracetools_test

- Update after namespacing C++ tracetools functions and macros
- Add tests for rcl_publish and rclcpp_publish tracepoints
- Allow asserting order of list of events
- Allow skipping test trace cleanup by setting an environment variable
- Add test for timer-node linking instrumentation
- Increased code coverage > 94% as part of QL1
- Add lifecycle node state transition instrumentation test
- Contributors: Alejandro Hernández Cordero, Christophe Bedard, Ingo Lütkebohle

tracetools_trace

- Add support for rcl_publish and rclcpp_publish tracepoints
- Fix flake8 blind except error by using more concrete types
- Allow configuring tracing directory through environment variables
- Cleanly stop ros2trace/tracetools_trace tracing on SIGINT
- Add instrumentation support for linking a timer to a node
- Add lifecycle node state transition instrumentation
- Contributors: Christophe Bedard, Ingo Lütkebohle
trajectory_msgs

- Change index.ros.org -> docs.ros.org. (#149)
- updating quality declaration links (re: ros2/docs.ros2.org#52) (#145)
- Update QDs to QL 1 (#135)
- Update package maintainers. (#132)
- Updated Quality Level to 2 (#131)
- Update Quality levels to level 3 (#124)
- Finish up API documentation (#123)
- Add Security Vulnerability Policy pointing to REP-2006. (#120)
- Contributors: Alejandro Hernández Cordero, Chris Lalancette, Michel Hidalgo, Stephen Brawner, brawner, shonigmann

turtlesim

- Ignore key up events in teleop_turtle_key on Windows (#118)
- Update maintainers (#106)
- Update goal response callback signature (#100)
- add holonomic motion for turtlesim (#98)
- add step value to turtlesim color parameters (#91)
- update Foxy turtle (#90)
- Contributors: Jacob Perron, Michel Hidalgo, Shane Loretz

unique_identifier_msgs

- Change index.ros.org -> docs.ros.org (#21)
- Update QD to QL 1 (#17)
- Update Quality Declaration to QL2. (#15)
- Update Quality level to level 3 (#13)
- Add Security Vulnerability Policy pointing to REP-2006. (#11)
- Contributors: Chris Lalancette, Michel Hidalgo, Stephen Brawner, brawner
urdf

• Work around Windows min/max bug. (#21)
• Enable -Wall -Wextra -Wpedantic (#20)
• Add dependency on TinyXML2 (#19)
• Remove TinyXML dependency from urdf. (#17)
• Make urdf plugable and revive urdf_parser_plugin (#13)
• Contributors: Audrow Nash, Chris Lalancette, Shane Loretz

urdf_parser_plugin

• Export urdfdom_headers as urdf_parser_plugin dependency. (#25)
• Make urdf plugable and revive urdf_parser_plugin (#13)
• Contributors: Michel Hidalgo, Shane Loretz

visualization_msgs

• Change index.ros.org -> docs.ros.org. (#149)
• updating quality declaration links (re: ros2/docs.ros2.org#52) (#145)
• Update QDs to QL 1 (#135)
• Update package maintainers. (#132)
• Updated Quality Level to 2 (#131)
• Update Quality levels to level 3 (#124)
• Add Security Vulnerability Policy pointing to REP-2006. (#120)
• Contributors: Alejandro Hernández Cordero, Chris Lalancette, Michel Hidalgo, Stephen Brawner, brawner, shonigmann

yaml_cpp_vendor

• Always preserve source permissions in vendor packages (#22)
• Add an override flag to force vendored build (#21)
• Reapply “Use system installed yaml-cpp 0.6 if available (#8)” (#16)
• Revert “Use system installed yaml-cpp 0.6 if available (#8)” (#15)
• Use system installed yaml-cpp 0.6 if available (#8)
• Contributors: Ivan Santiago Paunovic, Scott K Logan, Sean Yen
**zstd_vendor**

- Explicitly add emersonknapp as maintainer (#692)
- Always preserve source permissions in vendor packages (#645)
- Zstd should not install internal headers - some of them try include others that aren’t installed. We don’t use them. Avoid the situation (#631)
- Patch zstd 1.4.4 to include cmake_minimum_version bump to 2.8.12 (#579)
- Update the package.xml files with the latest Open Robotics maintainers (#535)
- Contributors: Emerson Knapp, Michael Jeronimo, Scott K Logan

### Table of Contents

- Supported Platforms
- Installation
- New features in this ROS 2 release
  - Ability to specify per-logger log levels
  - Ability to configure logging directory through environment variables
  - Ability to invoke rosidl pipeline outside CMake
  - Externally configure QoS at start-up
  - Python point_cloud2 utilities available
  - RViz2 Time Panel
  - ros2 topic echo can print serialized data
  - Get the YAML representation of messages
  - Ability to load parameter files at runtime through the ros2 command
  - Tools to check for QoS incompatibilities
  - Use launch substitutions in parameter files
  - Support for unique network flows
  - Rosbag2 New Features
- Changes since the Foxy release
  - Default RMW changed to Eclipse Cyclone DDS
  - Connext RMW changed to rmw_connextdds
  - Large improvements in testing and overall quality
  - rmw
  - ament_cmake
  - nav2
  - tf2_ros Python split out of tf2_ros
  - tf2_ros Python TransformListener uses global namespace
Galactic Geochelone is the seventh release of ROS 2. What follows is highlights of the important changes and features in Galactic Geochelone since the last release. For a list of all of the changes since Foxy, see the long form changelog <Galactic-Geochelone-Complete-Changelog>.

Supported Platforms

Galactic Geochelone is primarily supported on the following platforms:

Tier 1 platforms:
- Ubuntu 20.04 (Focal): amd64 and arm64
- Windows 10 (Visual Studio 2019): amd64

Tier 2 platforms:
- RHEL 8: amd64

Tier 3 platforms:
- Ubuntu 20.04 (Focal): arm32
- Debian Bullseye (11): amd64, arm64 and arm32
- OpenEmbedded Thud (2.6) / webOS OSE: arm32 and arm64
- Mac macOS 10.14 (Mojave): amd64

For more information about RMW implementations, compiler / interpreter versions, and system dependency versions see REP 2000.
Installation

Install Galactic Geochelone

New features in this ROS 2 release

Ability to specify per-logger log levels

It is now possible to specify different logging levels for different loggers on the command line:

```
ros2 run demo_nodes_cpp talker --ros-args --log-level WARN --log-level talker:=DEBUG
```

The above command sets a global log level of WARN, but sets the log level of the talker node messages to DEBUG. The --log-level command-line option can be passed an arbitrary number of times to set different log levels for each logger.

Ability to configure logging directory through environment variables

It is now possible to configure the logging directory through two environment variables: ROS_LOG_DIR and ROS_HOME. The logic is as follows:

* Use $ROS_LOG_DIR if ROS_LOG_DIR is set and not empty.
* Otherwise, use $ROS_HOME/log, using ~/.ros for ROS_HOME if not set or if empty.

Thus the default value stays the same: ~/.ros/log.

Related PRs: ros2/rcl_logging#53 and ros2/launch#460.

For example:

```
ROS_LOG_DIR=/tmp/foo ros2 run demo_nodes_cpp talker
```

Will place all logs in /tmp/foo.

```
ROS_HOME=/path/to/home ros2 run demo_nodes_cpp talker
```

Will place all logs in /path/to/home/log.

Ability to invoke rosidl pipeline outside CMake

It is now straightforward to invoke the rosidl interface generation pipeline outside CMake. Source code generators and interface definition translators are accessible through a unified command line interface.

For example, given a Demo message in some demo package like:

```
mkdir -p demo/msg
cd demo
cat << EOF > msg/Demo.msg
std_msgs/Header header
geometry_msgs/Twist twist
geometry_msgs/Accel accel
EOF
```
it is easy to generate C, C++, and Python support source code:

```bash
rosidl generate -o gen -t c -t cpp -t py -I$(ros2 pkg prefix --share std_msgs)/.. \  -I$(ros2 pkg prefix --share geometry_msgs)/.. demo msg/Demo.msg
```

Generated source code will be put in the `gen` directory.

One may also translate the message definition to a different format for a third-party code generation tool to consume:

```bash
rosidl translate -o gen --to idl -I$(ros2 pkg prefix --share std_msgs)/.. \  -I$(ros2 pkg prefix --share geometry_msgs)/.. demo msg/Demo.msg
```

The translated message definition will be put in the `gen` directory.

Note that these tools generate sources but do not build it – that responsibility is still on the caller. This is a first step towards enabling `rosidl` interface generation in build systems other than CMake. See the design document for further reference and next steps.

**Externally configure QoS at start-up**

It is now possible to externally configure the QoS settings for a node at start-up time. QoS settings are **not** configurable during runtime; they are only configurable at start-up. Node authors must opt-in to enable changing QoS settings at start-up. If the feature is enabled on a node, then QoS settings can be set with ROS parameters when a node first starts.

Demos in C++ and Python can be found here.

See the design document for more details.

Note, user code handling parameter changes with registered callbacks should avoid rejecting updates for unknown parameters. It was considered bad practice prior to Galactic, but with externally configurable QoS enabled it will result in a hard failure.

Related PRs: ros2/rclcpp#1408 and ros2/rclpy#635

**Python point_cloud2 utilities available**

Several utilities for interacting with `PointCloud2` messages in Python were ported to ROS 2. These utilities allow one to get a list of points from a `PointCloud2` message (`read_points` and `read_points_list`), and to create a `PointCloud2` message from a list of points (`create_cloud` and `create_cloud_xyz32`).

An example of creating `PointCloud 2` message, then reading it back:

```python
import sensor_msgs_py.point_cloud2
from std_msgs.msg import Header

pointlist = [[0.0, 0.1, 0.2]]

pointcloud = sensor_msgs_py.point_cloud2.create_cloud_xyz32(Header(frame_id='frame'), \  pointlist)

for point in sensor_msgs_py.point_cloud2.read_points(pointcloud):
  print(point)
```
RViz2 Time Panel

The RViz2 Time Panel, which shows the current Wall and ROS time, along with the elapsed Wall and ROS time, has been ported to RViz2. To enable the Time Panel, click on Panels -> Add New Panel, and select “Time”. A panel that looks like the following will appear:

![Time Panel Screenshot]

**ros2 topic echo can print serialized data**

When debugging middleware issues, it can be useful to see the raw serialized data that the RMW is sending. The `--raw` command-line flag was added to `ros2 topic echo` to show this data. To see this in action, run the following commands.

Terminal 1:

```
$ ros2 topic pub /chatter std_msgs/msg/String "data: 'hello'"
```

Terminal 2:

```
$ ros2 topic echo --raw /chatter
b'\x00\x01\x00\x00\x06\x00\x00\x00\x00\x00\x00hello\x00\x00\x00'
---
```

**Get the YAML representation of messages**

It is now possible to get a YAML representation of all messages in C++ using the `to_yaml` function. An example of code that prints out the YAML representation:

```c
#include <cstdio>
#include <std_msgs/msg/string.hpp>

int main()
{
    std_msgs::msg::String msg;
    msg.data = "hello world";
    printf("%s", rosidl_generator_traits::to_yaml(msg).c_str());
    return 0;
}
```
Ability to load parameter files at runtime through the ros2 command

ROS 2 has long had the ability to specify parameter values at startup (through command-line arguments or a YAML file), and to dump current parameters out to a file (through `ros2 param dump`). Galactic adds the ability to load parameter values at runtime from a YAML file using the `ros2 param load` verb. For example:

Terminal 1:

```bash
$ ros2 run demo_nodes_cpp parameter_blackboard
```

Terminal 2:

```bash
$ ros2 param set /parameter_blackboard foo bar   # sets 'foo' parameter to value 'bar'
$ ros2 param dump /parameter_blackboard         # dumps current value of parameters to ./
                         __parameter_blackboard.yaml
$ ros2 param set /parameter_blackboard foo different # sets 'foo' parameter to value
                         __'different'
$ ros2 param load /parameter_blackboard ./parameter_blackboard.yaml # reloads previous state of
                         parameters, 'foo' is back to 'bar'
```

Tools to check for QoS incompatibilities

Built on top of new QoS compatibility check APIs, `ros2doctor` and `rqt_graph` can now detect and report QoS incompatibilities between publishers and subscriptions.

Given a publisher and a subscription with incompatible QoS settings `<../../Concepts/Intermediate/About-Quality-of-Service-Settings>`:

Terminal 1:

```bash
$ ros2 run demo_nodes_py talker_qos -n 1000   # i.e. best_effort publisher
```

Terminal 2:

```bash
$ ros2 run demo_nodes_py listener_qos --reliable -n 1000   # i.e. reliable subscription
```

`ros2doctor` reports:

```bash
$ ros2 doctor --report
# ...
  QOS COMPATIBILITY LIST
topic [type] : /chatter [std_msgs/msg/String]
publisher node : talker_qos
subscriber node : listener_qos
compatibility status : ERROR: Best effort publisher and reliable subscription;
# ...
```

while `rqt_graph` shows:
Use launch substitutions in parameter files

Just like `rosparam` tags in ROS 1 `roslaunch`, `launch_ros` can now evaluate substitutions in parameter files. For example, given some `parameter_file_with_substitutions.yaml` like the following:

```yaml
/**:
    ros__parameters:
      launch_date: $(command date)
```

Set `allow_substs` to `True` to get substitutions evaluated upon Node launch:

```python
import launch
import launch_ros.parameter_descriptions
import launch_ros.actions

def generate_launch_description():
    return launch.LaunchDescription([
        launch_ros.actions.Node(
            package='demo_nodes_cpp',
            executable='parameter_blackboard',
            parameters=[
                launch_ros.parameter_descriptions.ParameterFile(
                    param_file='parameter_file_with_substitutions.yaml',
                    allow_substs=True)
            ]
        )
    ])```
XML launch files also support this.

```xml
<launch>
  <node pkg="demo_nodes_cpp" exec="parameter_blackboard">
    <param from="parameter_file_with_substitutions.yaml" allow_substs="true"/>
  </node>
</launch>

Related PR: ros2/launch_ros#168

**Support for unique network flows**

Applications may now require UDP/TCP and IP-based RMW implementations to provide unique network flows (i.e. unique Differentiated Services Code Points and/or unique IPv6 Flow Labels and/or unique ports in IP packet headers) for publishers and subscriptions, enabling QoS specifications for these IP streams in network architectures that support such a feature, like 5G networks.

To see this in action, you may run these C++ examples (to be found in the ros2/examples repository):

Terminal 1:

```
ros2 run examples_rclcpp_minimal_publisher publisher_member_function_with_unique_network_flow_endpoints
```

Terminal 2:

```
ros2 run examples_rclcpp_minimal_subscriber subscriber_member_function_with_unique_network_flow_endpoints
```

See the [Unique Network Flows design document](https://github.com/ros2/launch_ros/blob/master/design/unique-network-flows.md) for further reference.

**Rosbag2 New Features**

**Split recording by time**

In Foxy, you could only split bags as they were recording by the size of the bag, now you can also split by the elapsed time. The following command will split bagfiles into 100-second chunks.

```
ros2 bag record --all --max-bag-duration 100
```

**ros2 bag list**

This new command lists installed plugins of various types that rosbag2 uses.

```
$ ros2 bag list storage
rosbag2_v2
sqlite3

$ ros2 bag list converter
rosbag_v2_converter
```
Compression implementation is a plugin

In Foxy, rosbag2 compression was hardcoded with a Zstd library implementation. This has been rearchitected so that compression implementations are a plugin, and can be swapped out without modifying the core rosbag2 codebase. The default plugin that ships with rosc--galactic-rosbag2 is still the Zstd plugin - but now more can be released and used, and by selectively installing packages Zstd could be excluded from an installation.

Compress per-message

In Foxy, you could automatically compress each rosbag file as it was split (per-file compression), but now you can also specify per-message compression.

```
ros2 bag record --all --compression-format zstd --compression-mode message
```

Rosbag2 Python API

A new package rosbag2_py has been released in Galactic, which provides a Python API. This package is a pybind11 binding around the C++ API. As of the initial Galactic release, it does not yet expose all functionality available via the rosbag2_cpp API, but it is the sole connection for the ros2 bag CLI tool, so a good deal of functionality is available.

Performance testing package and performance improvements

A thorough performance analysis project was performed on rosbag2 since the Foxy release. The full initial report is available at https://github.com/ros2/rosbag2/blob/galactic/rosbag2_performance/rosbag2_performance_benchmarking/docs/rosbag2_performance_improvements.pdf. The package rosbag2_performance_benchmarking provides tools to run performance analyses, especially on recording, which helps us maintain and improve the performance of rosbag2.

Following this report, key work was done do improve the performance to a much more usable state for actual robot workflows. To highlight a key metric - in a high bandwidth stress test (200Mbps), the Foxy release dropped up to 70% of messages, whereas the Galactic version was approximately 100% retention. Please see the linked report for more details.

--regex and --exclude options for topic selection

The new recording options --regex and --exclude allow for fine-tuning the topics recorded in a bag, without having to explicitly list all topics. These options may be used together or separately, and in conjunction with --all.

The following command will record only topics with “scan” in the name.

```
ros2 bag record --regex "*scan*"
```

The following command will record all topics except for ones in /my_namespace/

```
ros2 bag record --all --exclude "/my_namespace/*"
```
**ros2 bag reindex**

ROS 2 bags are represented by a directory, instead of a single file. This directory contains a `metadata.yaml` file, and one or more bag files. When the `metadata.yaml` file is lost or missing, `ros2 bag reindex $bag_dir` will attempt to reconstruct it by reading all the bag files in the directory.

**Playback time control**

New controls have been added for rosbag2 playback - pause & resume, change rate, and play-next. As of the Galactic release, these controls are exposed only as services on the rosbag2 player node. Development is in progress to expose them to keyboard controls as well in `ros2 bag play`, but until then a user application with buttons or keyboard controls may be trivially implemented to call these services.

```bash
# In one shell
$ ros2 bag play my_bag

# In another shell
$ ros2 service list -t /rosbag2_player/get_rate [rosbag2_interfaces/srv/GetRate]
/rosbag2_player/is_paused [rosbag2_interfaces/srv/IsPaused]
/rosbag2_player/pause [rosbag2_interfaces/srv/Pause]
/rosbag2_player/play_next [rosbag2_interfaces/srv/PlayNext]
/rosbag2_player/resume [rosbag2_interfaces/srv/Resume]
/rosbag2_player/set_rate [rosbag2_interfaces/srv/SetRate]
/rosbag2_player/toggle_paused [rosbag2_interfaces/srv/TogglePaused]

# Check if playback is paused
$ ros2 service call /rosbag2_player/is_paused rosbag2_interfaces/IsPaused

# Pause playback
$ ros2 service call /rosbag2_player/pause rosbag2_interfaces/Pause

# Resume playback
$ ros2 service call /rosbag2_player/resume rosbag2_interfaces/Resume

# Change the paused state of playback to its opposite. If playing, pauses. If paused, resumes.
$ ros2 service call /rosbag2_player/toggle_paused rosbag2_interfaces/TogglePaused

# Get the current playback rate
$ ros2 service call /rosbag2_player/get_rate

# Set the current playback rate (must be > 0)
$ ros2 service call /rosbag2_player/set_rate rosbag2_interfaces/SetRate "rate: 0.1"

# Play a single next message (only works while paused)
$ ros2 service call /rosbag2_player/play_next rosbag2_interfaces/PlayNext
```
Playback publishes `/clock`

Rosbag2 can also dictate “simulation time” by publishing to the `/clock` topic during playback. The following commands will publish the clock message at a regular interval.

```
# Publish at default rate - 40Hz
ros2 bag play my_bag --clock

# Publish at specific rate - 100Hz
ros2 bag play my_bag --clock 100
```

Changes since the Foxy release

Default RMW changed to Eclipse Cyclone DDS

During the Galactic development process, the ROS 2 Technical Steering Committee voted to change the default ROS middleware (RMW) to Eclipse Cyclone DDS project of Eclipse Foundation. Without any configuration changes, users will get Eclipse Cyclone DDS by default. Fast DDS and Connext are still Tier-1 supported RMW vendors, and users can opt-in to use one of these RMWs at their discretion by using the `RMW_IMPLEMENTATION` environment variable. See the Working with multiple RMW implementations guide <../../How-To-Guides/Working-with-multiple-RMW-implementations> for more information.

Connext RMW changed to `rmw_connextdss`

A new RMW for Connext called `rmw_connextdss` was merged for Galactic. This RMW has better performance and fixes many of the issues with the older RMW `rmw_connext_cpp`.

Large improvements in testing and overall quality

Galactic contains many changes that fix race conditions, plug memory leaks, and fix user reported problems. Besides these changes, there was a concerted effort during Galactic development to improve overall quality of the system by implementing REP 2004. The `rclcpp` package and all of its dependencies (which include most of the ROS 2 non-Python core packages) were brought up to Quality Level 1 by:

- Having a version policy (QL1 requirement 1)
- Having a documented change control process (QL1 requirement 2)
- Documenting all features and public APIs (QL1 requirement 3)
- Adding many additional tests (QL1 requirement 4):
  - System tests for all features
  - Unit tests for all public APIs
  - Nightly performance tests
  - Code coverage at 95%
- Having all runtime dependencies of packages be at least as high as the package (QL1 requirement 5)
- Supporting all of the REP-2000 platforms (QL1 requirement 6)
- Having a vulnerability disclosure policy (QL1 requirement 7)
rmw

New API for checking QoS profile compatibility

`rmw_qos_profile_check_compatible` is a new function for checking the compatibility of two QoS profiles. RMW vendors should implement this API for QoS debugging and introspection features in tools such as `rqt_graph` to work correctly.

Related PR: ros2/rmw#299

ament_cmake

`ament_install_python_package()` now installs a Python egg

By installing a flat Python egg, Python packages installed using `ament_install_python_package()` can be discovered using modules such as `pkg_resources` and `importlib.metadata`. Also, additional metadata can be provided in a `setup.cfg` file (including entry points).

Related PR: ament/ament_cmake#326

ament_target_dependencies() handles SYSTEM dependencies

Some package dependencies can now be marked as SYSTEM dependencies, helping to cope with warnings in external code. Typically, SYSTEM dependencies are also excluded from dependency calculations – use them with care.

Related PR: ament/ament_cmake#297

nav2

Changes include, but are not limited to, a number of stability improvements, new plugins, interface changes, costmap filters. See Migration Guides for full list

tf2_ros Python split out of tf2_ros

The Python code that used to live in tf2_ros has been moved into its own package named tf2_ros_py. Any existing Python code that depends on tf2_ros will continue to work, but the `package.xml` of those packages should be amended to `exec_depend` on tf2_ros_py.

tf2_ros Python TransformListener uses global namespace

The Python TransformListener now subscribes to `/tf` and `/tf_static` in the global namespace. Previously, it was subscribing in the node’s namespace. This means that the node’s namespace will no longer have an effect on the `/tf` and `/tf_static` subscriptions.

For example:

```
ros2 run tf2_ros tf2_echo --ros-args -r __ns:=/test -- odom base_link
```

will subscribe to `/tf` and `/tf_static`, as `ros2 topic list` will show.

Related PR: ros2/geometry2#390
rclcpp

Change in spin_until_future_complete template parameters

The first template parameter of Executor::spin_until_future_complete was the future result type ResultT, and the method only accepted a std::shared_future<ResultT>. In order to accept other types of futures (e.g.: std::future), that parameter was changed to the future type itself.

In places where a spin_until_future_complete call was relying on template argument deduction, no change is needed. If not, this is an example diff:

```cpp
std::shared_future<MyResultT> future;
...
-executor.spin_until_future_complete<MyResultT>(future);
+executor.spin_until_future_complete<std::shared_future<MyResultT>>(future);
```

For more details, see ros2/rclcpp#1160. For an example of the needed changes in user code, see ros-visualization/interactive_markers#72.

Change in default /clock subscription QoS profile

The default was changed from a reliable communication with history depth 10 to a best effort communication with history depth 1. See ros2/rclcpp#1312.

Waitable API

Waitable API was modified to avoid issues with the MultiThreadedExecutor. This only affects users implementing a custom waitable. See ros2/rclcpp#1241 for more details.

Change in rclcpp's logging macros

Previously, the logging macros were vulnerable to a format string attack, where the format string is evaluated and can potentially execute code, read the stack, or cause a segmentation fault in the running program. To address this security issue, the logging macro now accepts only string literals for it's format string argument.

If you previously had code like:

```cpp
const char *my_const_char_string format = "Foo";
RCLCPP_DEBUG(get_logger(), my_const_char_string);
```

you should now replace it with:

```cpp
const char *my_const_char_string format = "Foo";
RCLCPP_DEBUG(get_logger(), "%s", my_const_char_string);
```

or:

```cpp
RCLCPP_DEBUG(get_logger(), "Foo");
```

This change removes some convenience from the logging macros, as std::strings are no longer accepted as the format argument.

If you previously had code with no format arguments like:
std::string my_std_string = "Foo";
RCLCPP_DEBUG(get_logger(), my_std_string);

you should now replace it with:

std::string my_std_string = "Foo";
RCLCPP_DEBUG(get_logger(), "%s", my_std_string.c_str());

Note: If you are using a std::string as a format string with format arguments, converting that string to a char * and using it as the format string will yield a format security warning. That’s because the compiler has no way at compile to introspect into the std::string to verify the arguments. To avoid the security warning, we recommend you build the string manually and pass it in with no format arguments like the previous example.

std::stringstream types are still accepted as arguments to the stream logging macros. See ros2/rclcpp#1442 for more details.

Parameter types are now static by default

Previously, the type of a parameter could be changed when a parameter was set. For example, if a parameter was declared as an integer, a later call to set the parameter could change that type to a string. This behavior could lead to bugs, and is rarely what the user wants. As of Galactic parameter types are static by default, and attempts to change the type will fail. If the previous dynamic behavior is desired, there is an mechanism to opt it in (see the code below).

// declare integer parameter with default value, trying to set it to a different type will fail.
node->declare_parameter("my_int", 5);
// declare string parameter with no default and mandatory user provided override.
// i.e. the user must pass a parameter file setting it or a command line rule -p <param_name>:=<value>
node->declare_parameter("string_mandatory_override", rclcpp::PARAMETER_STRING);
// Conditionally declare a floating point parameter with a mandatory override.
// Useful when the parameter is only needed depending on other conditions and no default is reasonable.
if (mode == "modeA") {
    node->declare_parameter("conditionally_declare_double_parameter", rclcpp::PARAMETER_DOUBLE);
}
// You can also get the old dynamic typing behavior if you want:
rcl_interfaces::msg::ParameterDescriptor descriptor;
descriptor.dynamic_typing = true;
node->declare_parameter("dynamically_typed_param", rclcpp::ParameterValue{}, descriptor);

New API for checking QoS profile compatibility

qos_check_compatible is a new function for checking the compatibility of two QoS profiles.
Related PR: ros2/rclcpp#1554

rclpy

Removal of deprecated Node.set_parameters_callback

The method Node.set_parameters_callback was deprecated in ROS Foxy and has been removed in ROS Galactic. Use Node.add_on_set_parameters_callback() instead. Here is some example code using it.

```python
import rclpy
import rclpy.node
from rcl_interfaces.msg import ParameterType
from rcl_interfaces.msg import SetParametersResult

rclpy.init()
node = rclpy.node.Node('callback_example')
node.declare_parameter('my_param', 'initial value')

def on_parameter_event(parameter_list):
    for parameter in parameter_list:
        node.get_logger().info(f'Got {parameter.name}={parameter.value}')
    return SetParametersResult(successful=True)

node.add_on_set_parameters_callback(on_parameter_event)
rclpy.spin(node)
```

Run this command to see the parameter callback in action.

```
ros2 param set /callback_example my_param "Hello World"
```

Parameter types are now static by default

In Foxy and earlier a call to set a parameter could change its type. As of Galactic parameter types are static and cannot be changed by default. If the previous behavior is desired, then set dynamic_typing to true in the parameter descriptor. Here is an example.

```python
import rclpy
import rclpy.node
from rcl_interfaces.msg import ParameterDescriptor

rclpy.init()
node = rclpy.node.Node('static_param_example')
node.declare_parameter('static_param', 'initial value')
```

(continues on next page)
Run these commands to see how statically and dynamically typed parameters are different.

```bash
$ ros2 param set /static_param_example dynamic_param 42
Set parameter successful
$ ros2 param set /static_param_example static_param 42
Setting parameter failed: Wrong parameter type, expected 'Type.STRING' got 'Type.INTEGER'
```


**New API for checking QoS profile compatibility**

`rclpy.qos.qos_check_compatible` is a new function for checking the compatibility of two QoS profiles. If the profiles are compatible, then a publisher and subscriber using them will be able to talk to each other.

```python
import rclpy.qos

publisher_profile = rclpy.qos.qos_profile_sensor_data
subscription_profile = rclpy.qos.qos_profile_parameter_events

print(rclpy.qos.qos_check_compatible(publisher_profile, subscription_profile))
```

```bash
$ python3 qos_check_compatible_example.py
(QoSCompatibility.ERROR, 'ERROR: Best effort publisher and reliable subscription;')
```

**rclcpp_action**

**Action client goal response callback signature changed**

The goal response callback should now take a shared pointer to a goal handle, instead of a future.

For example, old signature:

```cpp
void goal_response_callback(std::shared_future<GoalHandleFibonacci::SharedPtr> future)
```

New signature:

```cpp
void goal_response_callback(GoalHandleFibonacci::SharedPtr goal_handle)
```

Related PR: ros2/rclcpp#1311
rosidl_typesupport_introspection_c

API break in function that gets an element from an array

The signature of the function was changed because it was semantically different to all the other functions used to get an element from an array or sequence. This only affects authors of rmw implementations using the introspection typesupport.

For further details, see ros2/rosidl#531.

rcl_lifecycle and rclcpp_lifecycle

RCL's lifecycle state machine gets new init API

The lifecycle state machine in rcl_lifecycle was modified to expect a newly introduced options struct, combining general configurations for the state machine. The option struct allows to indicate whether the state machine shall be initialized with default values, whether its attached services are active and which allocator to be used.

```c
rcl_ret_t
rcl_lifecycle_state_machine_init(
    rcl_lifecycle_state_machine_t * state_machine,
    rcl_node_t * node_handle,
    const rosidl_message_type_support_t * ts_pub_notify,
    const rosidl_service_type_support_t * ts_srv_change_state,
    const rosidl_service_type_support_t * ts_srv_get_state,
    const rosidl_service_type_support_t * ts_srv_get_available_states,
    const rosidl_service_type_support_t * ts_srv_get_available_transitions,
    const rosidl_service_type_support_t * ts_srv_get_transition_graph,
    const rcl_lifecycle_state_machine_options_t * state_machine_options);
```

RCL's lifecycle state machine stores allocator instance

The options struct (discussed above) entails an instance of the allocator being used for initializing the state machine. This options struct and there the embodied allocator are being stored within the lifecycle state machine. As a direct consequence, the rcl_lifecycle_fini function no longer expects an allocator in its fini function but rather uses the allocator set in the options struct for deallocating its internal data structures.

```c
rcl_ret_t
rcl_lifecycle_state_machine_fini(
    rcl_lifecycle_state_machine_t * state_machine,
    rcl_node_t * node_handle);
```
RCLCPP’s lifecycle node exposes option to not instantiate services

In order to use rclcpp’s lifecycle nodes without exposing its internal services such as change_state, get_state et al., the constructor of a lifecycle node has a newly introduced parameter indicating whether or not the services shall be available. This boolean flag is set to true by default, not requiring any changes to existing API if not wished.

```cpp
explicit LifecycleNode(
    const std::string & node_name,
    const rclcpp::NodeOptions & options = rclcpp::NodeOptions(),
    bool enable_communication_interface = true);
```

Related PRs: ros2/rcl#882 and ros2/rclcpp#1507

rclifecycle and rclcpp_lifecycle

Recording - Split by time

Known Issues

ros2cli

Daemon slows down CLI on Windows

As a workaround, CLI commands may be used without a daemon e.g.:

```
ros2 topic list --no-daemon
```

Issue is tracked by ros2/ros2cli#637.

rqt

Some rqt_bag icons are missing

The icons to “Zoom In”, “Zoom Out”, “Zoom Home”, and “Toggle Thumbnails” are missing in rqt_bag. The issue is tracked in ros-visualization/rqt_bag#102

Most rqt utilities don’t work standalone on Windows

Launching the rqt utilities “standalone” on Windows (like ros2 run rqt_graph rqt_graph) generally doesn’t work. The workaround is to launch the rqt container process (rqt), and then insert the plugins to be used.
rviz2

**RViz2 panel close buttons are blank**

The upper right-hand corner of every RViz2 panel should contain an “X” to allow one to close the panel. Those buttons are there, but the “X” inside of them is missing on all platforms. The issue is being tracked in ros2/rviz2#692.

**Timeline before the release**

- **Mon. March 22, 2021 - Alpha** Preliminary testing and stabilization of ROS Core\(^1\) packages.
- **Mon. April 5, 2021 - Freeze** API and feature freeze for ROS Core\(^1\) packages in Rolling Ridley. Note that this includes rmw, which is a recursive dependency of ros_core. Only bug fix releases should be made after this point. New packages can be released independently.
- **Mon. April 19, 2021 - Branch** Branch from Rolling Ridley. rosdistro is reopened for Rolling PRs for ROS Core\(^1\) packages. Galactic development shifts from ros-rolling-* packages to ros-galactic-* packages.
- **Mon. April 26, 2021 - Beta** Updated releases of ROS Desktop\(^2\) packages available. Call for general testing.
- **Mon. May 17, 2021 - RC** Release Candidate packages are built. Updated releases of ROS Desktop\(^2\) packages available.
- **Thu. May 20, 2021 - Distro Freeze** Freeze rosdistro. No PRs for Galactic on the rosdistro repo will be merged (reopens after the release announcement).
- **Sun. May 23, 2021 - General Availability** Release announcement. rosdistro is reopened for Galactic PRs.

**Foxy Fitzroy (foxy)**

---

\(^1\) The ros_core variant is described in REP 2001 (ros-core).
\(^2\) The desktop variant is described in REP 2001 (desktop-variants).
Supported Platforms

Foxy Fitzroy is primarily supported on the following platforms:

Tier 1 platforms:
- Ubuntu 20.04 (Focal): amd64 and arm64
- Mac macOS 10.14 (Mojave)
- Windows 10 (Visual Studio 2019)

Tier 3 platforms:
- Ubuntu 20.04 (Focal): arm32
- Debian Buster (10): amd64, arm64 and arm32
- OpenEmbedded Thud (2.6) / webOS OSE: arm32 and x86

For more information about RMW implementations, compiler / interpreter versions, and system dependency versions see REP 2000.
Installation

Install Foxy Fitzroy

New features in this ROS 2 release

During the development the Foxy meta-ticket on GitHub contains an up-to-date state of the ongoing high-level tasks as well as references specific tickets with more details.

Changes in Patch Release 8 (2022-09-28)

Launch GroupAction scopes environment

The SetEnvironmentVariable action is now scoped to any GroupAction it is returned from.

For example, consider the following launch files,

Python

```python
import launch
from launch.actions import SetEnvironmentVariable
from launch.actions import GroupAction
from launch_ros.actions import Node
def generate_launch_description():
    return launch.LaunchDescription([
        SetEnvironmentVariable(name='my_env_var', value='1'),
        Node(package='foo', executable='foo', output='screen'),
        GroupAction([  
            SetEnvironmentVariable(name='my_env_var', value='2'),
        ]),
    ])
```

XML

```xml
<launch>
    <set_env name="my_env_var" value="1"/>
    <node pkg="foo" exec="foo" output="screen"/>
    <group>
        <set_env name="my_env_var" value="2"/>
    </group>
</launch>
```

Before patch release 8, the node `foo` will start with `my_env_var=2`, but now it will start with `my_env_var=1`.

To opt-out of the new behavior, you can set the argument `scoped=False` on the GroupAction.

Related tickets:

- ros2#1244
- launch#630
Changes in Patch Release 7 (2022-02-08)

Launch set_env frontend behavior change

launch#468 inadvertently changed behavior to the scope of the set_env action in frontend launch files. Changes to environment variables using the set_env action are no longer scoped to parent group actions, and instead apply globally. Since it was backported, the change affects this release.

We consider this change a regression and intend to fix the behavior in the next patch release and in future ROS distributions. We also plan to fix the behavior in Python launch files, which have never scoped setting environment variables properly.

Related issues:
- ros2#1244
- launch#597

Fix launch frontend parser

A refactor of the launch frontend parser fixed some issues parsing special characters. As a result, there has been a small behavior change when it comes to parsing strings. For example, previously to pass a number as a string you would have to add extra quotation marks (two sets of quotation marks were needed if using a substitution):

```xml
<!-- results in the string value "'3'" -->
<param name="foo" value="''3'''"/>
```

After the refactor, the above will result in the the string "''3'''" (note the extra set of quotation marks). Now, users should use the type attribute to signal that the value should be interpreted as a string:

```xml
<param name="foo" value="3" type="str"/>
```

Related pull requests:
- launch#530
- launch_ros#265

Fix memory leaks and undefined behavior in rmw_fastrtps_dynamic_cpp

API was changed in the following header files:
- rmw_fastrtps_dynamic_cpp/TypeSupport.hpp
- rmw_fastrtps_dynamic_cpp/TypeSupport_impl.hpp

Though technically they are publically accessible, it is unlikely people are using them directly. Therefore, we decided to break API in order to fix memory leaks and undefined behavior.

The fix was originally submitted in rmw_fastrtps#429 and later backported to Foxy in rmw_fastrtps#577.
Changes in Patch Release 2 (2020-08-07)

Bug in static_transform_publisher

During the development of Foxy, a bug was introduced into the tf2_ros static_transform_publisher program. The implementation of the order of the Euler angles passed to static_transform_publisher disagrees with the documentation. Foxy patch release 2 fixes the order so that the implementation agrees with the documentation (yaw, pitch, roll). For users who have started using the initial Foxy release or patch release 1, this means that any launch files that use static_transform_publisher will have to have the command-line order swapped according to the new order. For users who are coming from ROS 2 Dashing, ROS 2 Eloquent, or ROS 1, no changes need to be made to port to Foxy patch release 2.

Changes since the Eloquent release

Classic CMake vs. modern CMake

In “classic” CMake a package provides CMake variables like `<pkgname>_INCLUDE_DIRS` and `<pkgname>_LIBRARIES` when being find_package()-ed. With ament_cmake that is achieved by calling ament_export_include_directories and ament_export_libraries. In combination with ament_export_dependencies, ament_cmake ensures that all include directories and libraries of recursive dependencies are concatenated and included in these variables.

In “modern” CMake a package provides an interface target instead (commonly named `<pkgname>::<pkgname>`) which in itself encapsulates all recursive dependencies. In order to export a library target to use modern CMake ament_export_targets needs to be called with an export name which is also used when installing the libraries using install(TARGETS `<libA>` `<libB>` EXPORT `<export_name>` ...). The exported interface targets are available through the CMake variable `<pkgname>_TARGETS`. For library targets to be exportable like this they must not rely on classic functions affecting global state like include_directories() but set the include directories on the target itself - for the build as well as install environment - using generator expressions, e.g. target_include_directories(`<target>` PUBLIC "$<BUILD_INTERFACE:${CMAKE_CURRENT_BINARY_DIR}/include>" "$<INSTALL_INTERFACE:include>").

When ament_target_dependencies is used to add dependencies to a library target the function uses modern CMake targets when they are available. Otherwise it falls back to using classic CMake variables. As a consequence you should only export modern CMake targets if all dependencies are also providing modern CMake targets. Otherwise the exported interface target will contain the absolute paths to include directories / libraries in the generated CMake logic which makes the package non-relocatable.

For examples how packages have been updated to modern CMake in Foxy see ros2/ros2#904.

ament_export_interfaces replaced by ament_export_targets

The CMake function ament_export_interfaces from the package ament_cmake_export_interfaces has been deprecated in favor of the function ament_export_targets in the new package ament_cmake_export_targets. See the GitHub ticket ament/ament_cmake#237 for more context.
rosidl_generator_c|cpp namespace / API changes

The packages `rosidl_generator_c` and `rosidl_generator_cpp` have been refactored with many headers and sources moved into the new packages `rosidl-runtime_c` and `rosidl-runtime_cpp`. The intention is to remove run dependencies on the generator packages and therefore the code generation tools using Python. While moving the headers the include paths / namespaces were updated accordingly so in many cases changing include directives from the generator package to the runtime package is sufficient.

The generated C / C++ code has also been refactored. The files ending in `__struct.h|hpp`, `__functions.h`, `__traits.hpp`, etc. have been moved into a subdirectory `detail` but most code only includes the header named after the interface without any of these suffixes.

Some types regarding string and sequence bounds have also been renamed to match the naming conventions but they aren’t expected to be used in user code (above RMW implementation and type support packages)

For more information see ros2/rosidl#446 (for C) and ros2/rosidl#447 (for C++).

Default working directory for ament_add_test

The default working directory for tests added with `ament_add_test` has been changed to `CMAKE_CURRENT_BINARY_DIR` to match the behavior of CMake `add_test`. Either update the tests to work with the new default or pass `WORKING_DIRECTORY ${CMAKE_SOURCE_DIR}` to restore the previous value.

Default Console Logging Format

The default console logging output format was changed to include the timestamp by default, see:

- https://github.com/ros2/rcutils/pull/190
- https://discourse.ros.org/t/ros2-logging-format/11549

Default Console Logging Output Stream

As of Foxy, all logging messages at all severity levels get logged to stderr by default. This ensures that logging messages come out immediately, and brings the ROS 2 logging system into alignment with most other logging systems. It is possible to change the stream to stdout at runtime via the `RCUTILS_LOGGING_USE_STDOUT` environment variable, but all logging messages will still go to the same stream. See https://github.com/ros2/rcutils/pull/196 for more details.

launch_ros

Node name and namespace parameters changed

The Node action parameters related to naming have been changed:

- `node_name` has been renamed to `name`
- `node_namespace` has been renamed to `namespace`
- `node_executable` has been renamed to `executable`
- `exec_name` has been added for naming the process associated with the node. Previously, users would have used the `name` keyword argument.
The old parameters have been deprecated.
These changes were made to make the launch frontend more idiomatic. For example, instead of

```xml
<node pkg="demo_nodes_cpp" exec="talker" node-name="foo" />
```

we can now write

```xml
<node pkg="demo_nodes_cpp" exec="talker" name="foo" />
```

This change also applies to ComposableNodeContainer, ComposableNode, and LifecycleNode. For examples, see the relevant changes to the demos.
Related pull request in launch_ros.

rclcpp

**Change in Advanced Subscription Callback Signature**

With the pull request https://github.com/ros2/rclcpp/pull/1047 the signature of callbacks which receive the message info with the message has changed. Previously it used the rmw type rmw_message_info_t, but now uses the rclcpp type rclcpp::MessageInfo. The required changes are straightforward, and can be seen demonstrated in these pull requests:

- https://github.com/ros2/system_tests/pull/423/files
- https://github.com/ros2/rosbag2/pull/375/files
- https://github.com/ros2/ros1_bridge/pull/253/files

**Change in Serialized Message Callback Signature**

The pull request ros2/rclcpp#1081 introduces a new signature of the callbacks for retrieving ROS messages in serialized form. The previously used C-Struct rcl_serialized_message_t is being superseded by a C++ data type rclcpp::SerializedMessage.

The example nodes in demo_nodes_cpp, namely talker_serialized_message as well as listener_serialized_message reflect these changes.

**Breaking change in Node Interface getters’ signature**

With pull request ros2/rclcpp#1069, the signature of node interface getters has been modified to return shared ownership of node interfaces (i.e. an std::shared_ptr) instead of a non-owning raw pointer. Required changes in downstream packages that relied on the previous signature are simple and straightforward: use the std::shared_ptr::get() method.
Deprecate set_on_parameters_set_callback

Instead, use the rclcpp::Node methods add_on_set_parameters_callback and remove_on_set_parameters_callback for adding and removing functions that are called when parameters are set.

Related pull request: https://github.com/ros2/rclcpp/pull/1123

Breaking change in Publisher getter signature

With pull request ros2/rclcpp#1119, the signature of publisher handle getter has been modified to return shared ownership of the underlying rcl structure (i.e. an std::shared_ptr) instead of a non-owning raw pointer. This was necessary to fix a segfault in certain circumstances. Required changes in downstream packages that relied on the previous signature are simple and straightforward: use the std::shared_ptr::get() method.

rclcpp_action

Deprecate ClientGoalHandle::async_result()

Using this API, it is possible to run into a race condition causing an exception to be thrown. Instead, prefer to use Client::async_get_result(), which is safer.

See ros2/rclcpp#1120 and the connected issue for more info.

rclpy

Support for multiple on parameter set callbacks

Use the Node methods add_on_set_parameters_callback and remove_on_set_parameters_callback for adding and removing functions that are called when parameters are set.

The method set_parameters_callback has been deprecated.


rmw_connext_cpp

Connext 5.1 locator kinds compatibility mode

Up to and including Eloquent, rmw_connext_cpp was setting dds.transport.use_510_compatible_locator_kinds property to true. This property is not being forced anymore, and shared transport communication between Foxy and previous releases will stop working. Logs similar to:

```plaintext
PRESParticipant_checkTransportInfoMatching:Warning: discovered remote participant 'RTI→Administration Console' using the 'shmem' transport with class ID 16777216. This class ID does not match the class ID 2 of the same transport in the local→participant 'talker'. These two participants will not communicate over the 'shmem' transport.
```

(continues on next page)
Check the value of the property `dds.transport.use_510_compatible_locator_kinds` in the local participant. See https://community.rti.com/kb/what-causes-error-discovered-remote-participant for additional info.

will be observed when this incompatibility happens.

If compatibility is needed, it can be set up in an external QoS profiles files containing:

```xml
<participant_qos>
  <property>
    <element>
      <name>dds.transport.use_510_compatible_locator_kinds</name>
      <value>1</value>
    </element>
  </property>
</participant_qos>
```

Remember to set the `NDDS_QOS_PROFILES` environment variable to the QoS profiles file path. For more information, see How to Change Transport Settings in 5.2.0 Applications for Compatibility with 5.1.0 section of Transport Compatibility.

**rviz**

**Tools timestamp messages using ROS time**

‘2D Pose Estimate’, ‘2D Nav Goal’, and ‘Publish Point’ tools now timestamp their messages using ROS time instead of system time, in order for the `use_sim_time` parameter to have an effect on them.

Related pull request: https://github.com/ros2/rviz/pull/519

**std_msgs**

**Deprecation of messages**

Although discouraged for a long time we have officially deprecated the following messages in std_msgs. There are copies in example_interfaces

- std_msgs/msg/Bool
- std_msgs/msg/Byte
- std_msgs/msg/ByteMultiArray
- std_msgs/msg/Char
- std_msgs/msg/Float32
- std_msgs/msg/Float32MultiArray
- std_msgs/msg/Float64
• std_msgs/msg/Float64MultiArray
• std_msgs/msg/Int16
• std_msgs/msg/Int16MultiArray
• std_msgs/msg/Int32
• std_msgs/msg/Int32MultiArray
• std_msgs/msg/Int64
• std_msgs/msg/Int64MultiArray
• std_msgs/msg/Int8
• std_msgs/msg/Int8MultiArray
• std_msgs/msg/MultiArrayDimension
• std_msgs/msg/MultiArrayLayout
• std_msgs/msg/String
• std_msgs/msg/UInt16
• std_msgs/msg/UInt16MultiArray
• std_msgs/msg/UInt32
• std_msgs/msg/UInt32MultiArray
• std_msgs/msg/UInt64
• std_msgs/msg/UInt64MultiArray
• std_msgs/msg/UInt8
• std_msgs/msg/UInt8MultiArray

Security features

Use of security enclaves

As of Foxy, domain participants are no longer mapped directly to ROS nodes. As a result, ROS 2 security features (which are specific to domain participants) are also no longer mapped directly to ROS nodes. Instead, Foxy introduces the concept of a security “enclave”, where an “enclave” is a process or group of processes that will share the same identity and access control rules.

This means that security artifacts are not retrieved based on the node name anymore but based on the Security enclave name. A node enclave name can be set by using the ROS argument --enclave, e.g. ros2 run demo_nodes_py talker --ros-args --enclave /my_enclave

Related design document: https://github.com/ros2/design/pull/274

Note that permissions files are limited by the underlying transport packet size, so grouping many permissions under the same enclave will not work if the resulting permissions file exceed 64kB. Related issue [ros2/sros2#228]
Renaming of the environment variables

Table 2: Environment variables renaming

<table>
<thead>
<tr>
<th>Name in Eloquent</th>
<th>Name in Foxy</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROS_SECURITY_ROOT_DIRECTORY</td>
<td>ROS_SECURITY_KEYSTORE</td>
</tr>
<tr>
<td>ROS_SECURITY_NODE_DIRECTORY</td>
<td>ROS_SECURITY_ENCLAVE_OVERRIDE</td>
</tr>
</tbody>
</table>

Known Issues

- [ros2/ros2#922] Services’ performance is flaky for rclcpp nodes using eProsima Fast-RTPS or ADLINK CycloneDDS as RMW implementation. Specifically, service clients sometimes do not receive the response from servers.
- [ros2/rclcpp#1212] Ready reentrant Waitable objects can attempt to execute multiple times.

Timeline before the release

A few milestones leading up to the release:

**Note:** The dates below reflect an extension by roughly two weeks due to the coronavirus pandemic.

**Wed. April 22nd, 2020** API and feature freeze for ros_core¹ packages. Note that this includes rmw, which is a recursive dependency of ros_core. Only bug fix releases should be made after this point. New packages can be released independently.

**Mon. April 29th, 2020 (beta)** Updated releases of desktop² packages available. Testing of the new features.

**Wed. May 27th, 2020 (release candidate)** Updated releases of desktop² packages available.

**Wed. June 3rd, 2020** Freeze rosdistro. No PRs for Foxy on the rosdistro repo will be merged (reopens after the release announcement).

Eloquent Elusor (eloquent)

---

1 The ros_core variant described in the variants repository.
2 The desktop variant described in the variants repository.
Eloquent Elusor is the fifth release of ROS 2.

## Supported Platforms

Eloquent Elusor is primarily supported on the following platforms:

### Tier 1 platforms:
- Ubuntu 18.04 (Bionic): `amd64` and `arm64`
- Mac macOS 10.14 (Mojave)
- Windows 10 (Visual Studio 2019)

### Tier 2 platforms:
- Ubuntu 18.04 (Bionic): `arm32`

### Tier 3 platforms:
- Debian Stretch (9): `amd64`, `arm64` and `arm32`
- OpenEmbedded Thud (2.6) / webOS OSE: `arm32` and `x86`

For more information about RMW implementations, compiler / interpreter versions, and system dependency versions see REP 2000.

## Installation

Install Eloquent Elusor

## New features in this ROS 2 release

A few features and improvements we would like to highlight:
- Support for markup-based launch files (XML/YAML)
- Improved launch-based testing
- Passing key-value parameters on CLI
- Support stream logging macros
- Per-node logging - All stdout/stderr output from nodes are logged in `~/ros`
- `ros2doctor`
- Improved performance of sourcing setup files
- `rviz`: interactive markers, torque ring, tf message filters
- `rqt`: parameter plugin, tf tree plugin, robot steering plugin (also backported to Dashing)
- `turtlesim` (also backported to Dashing)
- RMW implementations:
  - API to loan message for zero copy, used by `rmw_iceoryx`
  - Fast RTPS 1.9.3
  - New Tier-2 implementation: `rmw_cyclonedds` (also backported to Dashing)
- Environment variable `ROS_LOCALHOST_ONLY` to limit communication to localhost
- MacOS Mojave Support
- Tracing instrumentation for `rcl` and `rclcpp`

During the development the Eloquent meta ticket on GitHub contains an up-to-date state of the ongoing high level tasks as well as references specific tickets with more details.

### Changes since the Dashing release

#### geometry_msgs

The `geometry_msgs/msg/Quaternion.msg` interface now default initializes to a valid quaternion, with the following values:

\[
\begin{align*}
x &= 0 \\
y &= 0 \\
z &= 0 \\
w &= 1
\end{align*}
\]

Here is the pull request for more detail: [https://github.com/ros2/common_interfaces/pull/74](https://github.com/ros2/common_interfaces/pull/74)

Static transform broadcasters and listeners now use QoS durability `transient_local` on the `/tf_static` topic. Similar to the latched setting in ROS 1, static transforms only need to be published once. New listeners will receive transforms from all static broadcasters that are alive and have published before. All publishers must be updated to use this durability setting or their messages won’t be received by transform listeners. See this pull request for more detail: [https://github.com/ros2/geometry2/pull/160](https://github.com/ros2/geometry2/pull/160)

#### rclcpp

**API Break with `get_actual_qos()`**

Introduced in Dashing, the `get_actual_qos()` method on the `PublisherBase` and `SubscriptionBase` previously returned an `rmw` type, `rmw_qos_profile_t`, but that made it awkward to reuse with the creation of other entities. Therefore it was updated to return a `rclcpp::QoS` instead.

Existing code will need to use the `rclcpp::QoS::get_rmw_qos_profile()` method if the `rmw` profile is still required. For example:
The rationale for breaking this directly rather than doing a tick-tock is that it is a new function and is expected to be used infrequently by users. Also, since only the return type is changing, adding a new function with a different would betoonlywaytodoadeprecationcycleand get_actual_qos() is the most appropriate name, so we would be forced to pick a less obvious name for the method.

API Break with Publisher and Subscription Classes

In an effort to streamline the construction of Publishers and Subscriptions, the API of the constructors were changed. It would be impossible to support a deprecation cycle, because the old signature takes an rcl type and the new one takes the NodeBaseInterface type so that it can get additional information it now needs, and there’s no way to get the additional information needed from just the rcl type. The new signature could possibly be backported if that would help contributors, but since the publishers and subscriptions are almost always created using the factory functions or some other higher level API, we do not expect this to be a problem for most users.

Please see the original pr for more detail and comment there if this causes issues:

https://github.com/ros2/rclcpp/pull/867

Compiler warning about unused result of add_on_set_parameters_callback

Since Eloquent Patch Release 2 (2020-12-04)

Users should retain the handle returned by rclcpp::Node::add_on_set_parameters_callback, otherwise their callback may be unregistered. A warning has been added to help identify bugs where the returned handle is not used.

https://github.com/ros2/rclcpp/pull/1243

rmw

API Break Due to Addition of Publisher and Subscription Options

The rmw_create_publisher() method had a new argument added of type const rmw_publisher_options_t *. This new structure holds options (beyond the typesupport, topic name, and QoS) for new publishers.

The rmw_create_subscription() method had one argument removed, bool ignore_local_publications, and replaced by the new options of type const rmw_subscription_options_t *. The ignore_local_publications option was moved into the new rmw_subscription_options_t type.

In both cases the new argument, which are pointers, may never be null, and so the rmw implementations should check to make sure the options are not null. Additionally, the options should be copied into the corresponding rmw structure.

See this pull request, and the associated pull requests for more details:

https://github.com/ros2/rmw/pull/187
ros2cli

ros2msg and ros2srv deprecated

The CLI tools ros2msg and ros2srv are deprecated. They have been replaced by the tool ros2interface, which also supports action and IDL interfaces. You can run `ros2 interface --help` for usage.

ros2node

Service clients have been added to ros2node info. As part of that change the Python function `ros2node.api.get_service_info` has been renamed to `ros2node.api.get_service_server_info`.

rviz

Renamed ‘2D Nav Goal’ tool

The tool was renamed to ‘2D Goal Pose’ and the default topic was changed from `/move_base_simple/goal` to `/goal_pose`.

Here is the related pull request:
https://github.com/ros2/rviz/pull/455

TF2 Buffer

TF2 buffers now have to be given a timer interface.

If a timer interface is not given, an exception will be thrown.

For example:

```cpp
tf = std::make_shared<tf2_ros::Buffer>(get_clock());
// The next two lines are new in Eloquent
auto timer_interface = std::make_shared<tf2_ros::CreateTimerROS>(
    this->get_node_base_interface(),
    this->get_node_timers_interface());
tf->setCreateTimerInterface(timer_interface);
// Pass the Buffer to the TransformListener as before
transform_listener = std::make_shared<tf2_ros::TransformListener>(*tf);
```

rcl

ROS command line argument changes

To cope with an increasingly complex interface, with a now extended set of configuration options, ROS CLI syntax has been changed. As an example, a command line using Dashing syntax like:

```bash
ros2 run some_package some_node foo:=bar __params:=/path/to/params.yaml __log__level:=WARN --user-flag
```

is written using Eloquent (and onwards) syntax as:
This explicit syntax affords new features, like single parameter assignment `--param name:=value`. For further reference and rationale, check the ROS command line arguments design document.

**Warning:** Former syntax has been deprecated and is due for removal in the next release.

**Known Issues**

- [ros2/rclcpp#893] `rclcpp::Context` is not destroyed because of a reference cycle with `rclcpp::GraphListener`. This causes a memory leak. A fix has not been backported because of the risk of breaking ABI.

**Timeline before the release**

A few milestones leading up to the release:

- **Mon. Sep 30th (alpha)** First releases of core packages available. Testing can happen from now on (some features might not have landed yet).
- **Fri. Oct 18th** API and feature freeze for core packages. Only bug fix releases should be made after this point. New packages can be released independently.
- **Thu. Oct 24th (beta)** Updated releases of core packages available. Additional testing of the latest features.
- **Wed. Nov 13th (release candidate)** Updated releases of core packages available.
- **Tue. Nov 19th** Freeze rosdistro. No PRs for Eloquent on the rosdistro repo will be merged (reopens after the release announcement).

**Dashing Diademata (dashing)**

**Table of Contents**

- Supported Platforms
- Installation
- New features in this ROS 2 release
- Changes since the Crystal release
  - Declaring Parameters
  - `ament_cmake`
  - `rclcpp`
  - `rclcpp_components`
Dashing Diademata is the fourth release of ROS 2.

**Supported Platforms**

Dashing Diademata is primarily supported on the following platforms:

Tier 1 platforms:
- Ubuntu 18.04 (Bionic): `amd64` and `arm64`
- Mac macOS 10.12 (Sierra)
- Windows 10 (Visual Studio 2019)

Tier 2 platforms:
- Ubuntu 18.04 (Bionic): `arm32`

Tier 3 platforms:
- Debian Stretch (9): `amd64`, `arm64` and `arm32`
- OpenEmbedded Thud (2.6) / webOS OSE: `arm32` and x86

For more information about RMW implementations, compiler / interpreter versions, and system dependency versions see REP 2000.

**Installation**

Install Dashing Diademata
New features in this ROS 2 release

A few features and improvements we would like to highlight:

- **Components** are now the recommended way to write your node. They can be used standalone as well as being composed within a process and both ways are fully support from launch files.

- The **intra-process communication** (C++ only) has been improved - both in terms of latency as well as minimizing copies.

- The Python client library has been updated to match most of the C++ equivalent and some important bug fixes and improvements have landed related to memory usage and performance.

- Parameters are now a complete alternative to dynamic_reconfigure from ROS 1 including constraints like ranges or being read-only.

- By relying on (a subset of) IDL 4.2 for the message generation pipeline it is now possible to use .idl files (beside .msg/.srv/.action files). This change comes with support for optional UTF-8 encoding for ordinary strings as well as UTF-16 encoded multi-byte strings (see wide strings design article).

- Command line tools related to actions and components.

- Support for Deadline, Lifespan & Liveliness quality of service settings.

- **MoveIt 2 alpha release.**

Please see the Dashing meta ticket on GitHub, which contains more information as well as references to specific tickets with additional details.

Changes since the Crystal release

Declaring Parameters

There have been some changes to the behavior of parameters starting in Dashing, which have also lead to some new API’s and the deprecation of other API’s. See the rclcpp and rclpy sections below for more information about API changes.

Getting and Setting Undeclared Parameters

As of Dashing, parameters now need to be declared before being accessed or set.

Before Dashing, you could call get_parameter(name) and get either a value, if it had been previously set, or a parameter of type PARAMETER_NOT_SET. You could also call set_parameter(name, value) at any point, even if the parameter was previously unset.

Since Dashing, you need to first declare a parameter before getting or setting it. If you try to get or set an undeclared parameter you will either get an exception thrown, e.g. ParameterNotDeclaredException, or in certain cases you will get an unsuccessful result communicated in a variety of ways (see specific functions for more details).

However, you can get the old behavior (mostly, see the note in the next paragraph) by using the allow_undeclared_parameters option when creating your node. You might want to do this in order to avoid code changes for now, or in order to fulfill some uncommon use cases. For example, a “global parameter server” or “parameter blackboard” may want to allow external nodes to set new parameters on itself without first declaring them, so it may use the allow_undeclared_parameters option to accomplish that. In most cases, however, this option is not recommended because it makes the rest of the parameter API less safe to bugs like parameter name typos and “use before set” logical errors.
Note that using `allow_undeclared_parameters` will get you most of the old behavior specifically for “get” and “set” methods, but it will not revert all the behavior changes related to parameters back to how it was for ROS Crystal. For that you need to also set the `automatically_declare_parameters_from_overrides` option to `true`, which is described below in Parameter Configuration using a YAML File.

### Declaring a Parameter with a ParameterDescriptor

Another benefit to declaring your parameters before using them, is that it allows you to declare a parameter descriptor at the same time.

Now when declaring a parameter you may include a custom `ParameterDescriptor` as well as a name and default value. The `ParameterDescriptor` is defined as a message in `rcl_interfaces/msg/ParameterDescriptor` and contains meta data like `description` and constraints like `read_only` or `integer_range`. These constraints can be used to reject invalid values when setting parameters and/or as hints to external tools about what values are valid for a given parameter. The `read_only` constraint will prevent the parameter’s value from changing after being declared, as well as prevent if from being undeclared.

For reference, here’s a link to the `ParameterDescriptor` message as of the time of writing this:

https://github.com/ros2/rcl_interfaces/blob/0aba5a142878c2077d7a03977087e7d74d40ee68/rcl_interfaces/msg/ParameterDescriptor.msg#L1

### Parameter Configuration using a YAML File

As of Dashing, parameters in a YAML configuration file, e.g. passed to the node via the command line argument `__params:=`, are only used to override a parameter's default value when declaring the parameter.

Before Dashing, any parameters you passed via a YAML file would be implicitly set on the node. Since Dashing, this is no longer the case, as parameters need to be declared in order to appear on the node to external observers, like `ros2 param list`.

The old behavior may be achieved using the `automatically_declare_parameters_from_overrides` option when creating a node. This option, if set to `true`, will automatically declare all parameters in the input YAML file when the node is constructed. This may be used to avoid major changes to your existing code or to serve specific use cases. For example, a “global parameter server” may want to be seeded with arbitrary parameters on launch, which it could not have declared ahead of time. Most of the time, however, this option is not recommended, as it may lead to setting a parameter in a YAML file with the assumption that the node will use it, even if the node does not actually use it.

In the future we hope to have a checker that will warn you if you pass a parameter to a node that it was not expecting.

The parameters in the YAML file will continue to influence the value of parameters when they are first declared.

### ament_cmake

The CMake function `ament_index_has_resource` was returning either `TRUE` or `FALSE`. As of this release it returns either the prefix path in case the resource was found or `FALSE`.

If you are using the return value in a CMake condition like this:

```cmake
ament_index_has_resource(var ...)
if(${var})
```

you need to update the condition to ensure it considers a string value as `TRUE`:
Behavior Change for Node::get_node_names()

The function NodeGraph::get_node_names(), and therefore also Node::get_node_names(), now returns a `std::vector<std::string>` containing fully qualified node names with their namespaces included, instead of just the node names.

Changed the Way that Options are Passed to Nodes

Extended arguments (beyond name and namespace) to the rclcpp::Node() constructor have been replaced with a rclcpp::NodeOptions structure. See ros2/rclcpp#622 for details about the structure and default values of the options.

If you are using any of the extended arguments to rclcpp::Node() like this:

```cpp
auto context = rclcpp::contexts::default_context::get_global_default_context();
std::vector<std::string> args;
std::vector<rclcpp::Parameter> params = { rclcpp::Parameter("use_sim_time", true) };
auto node = std::make_shared<rclcpp::Node>("foo_node", "bar_namespace", context, args, ...
˓→params);
```

You need to update to use the NodeOptions structure

```cpp
std::vector<std::string> args;
std::vector<rclcpp::Parameter> params = { rclcpp::Parameter("use_sim_time", true) };
rclcpp::NodeOptions node_options;
node_options.arguments(args);
node_options.parameter_overrides(params);
auto node = std::make_shared<rclcpp::Node>("foo_node", "bar_namespace", node_options);
```

Changes to Creating Publishers and Subscriptions

There have been a few changes to creating publishers and subscriptions which are new in Dashing:

- QoS settings are now passed using the new rclcpp::QoS class, and the API encourages the user to specify at least the history depth.
- Options are now passed as an object, i.e. rclcpp::PublisherOptions and rclcpp::SubscriptionOptions.

All changes are backwards compatible (no code changes are required), but several existing call styles have been deprecated. Users are encouraged to update to the new signatures.

In the past, when creating a publisher or subscription, you could either not specify any QoS settings (e.g. just provide topic name for a publisher) or you could specify a “qos profile” data structure (of type `rmw_qos_profile_t`) with all the settings already set. Now you must use the new rclcpp::QoS object to specify your QoS and at least the history settings for your QoS. This encourages the user to specify a history depth when using KEEP_LAST, rather than defaulting it to a value that may or may not be appropriate.
In ROS 1, this was known as the *queue_size* and it was required in both C++ and Python. We’re changing the ROS 2 API to bring this requirement back.

Also, any options which could previously be passed during creation of a publisher or subscription have now been encapsulated in an `rclcpp::PublisherOptions` and `rclcpp::SubscriptionOptions` class respectively. This allows for shorter signatures, more convenient use, and for adding new future options without breaking API.

Some signatures for creating publishers and subscribers are now deprecated, and new signatures have been added to allow you to use the new `rclcpp::QoS` and publisher/subscription option classes.

These are the new and recommended APIs:

```cpp
template<
    typename MessageT,
    typename AllocatorT = std::allocator<void>,
    typename PublisherT = ::rclcpp::Publisher<MessageT, AllocatorT>>
std::shared_ptr<PublisherT>
create_publisher(
    const std::string & topic_name,
    const rclcpp::QoS & qos,
    const PublisherOptionsWithAllocator<AllocatorT> & options = PublisherOptionsWithAllocator<AllocatorT>()
);
```

```cpp
template<
    typename MessageT,
    typename CallbackT,
    typename AllocatorT = std::allocator<void>,
    typename SubscriptionT = rclcpp::Subscription<
        typename rclcpp::subscription_traits::has_message_type<CallbackT>::type, AllocatorT>>
std::shared_ptr<SubscriptionT>
create_subscription(
    const std::string & topic_name,
    const rclcpp::QoS & qos,
    CallbackT && callback,
    const SubscriptionOptionsWithAllocator<AllocatorT> & options = SubscriptionOptionsWithAllocator<AllocatorT>(),
    typename rclcpp::message_memory_strategy::MessageMemoryStrategy<
        typename rclcpp::subscription_traits::has_message_type<CallbackT>::type, AllocatorT>::SharedPtr
    msg_mem_strat = nullptr
);
```

And these are the deprecated ones:

```cpp
// [[deprecated("use create_publisher(const std::string &, const rclcpp::QoS &, ...) instead")]]
std::shared_ptr<PublisherT>
create_publisher(
```

(continues on next page)
\textbf{Vulcanexus Documentation, Release 1.0.0}

\begin{verbatim}
const std::string & topic_name,
size_t qos_history_depth,
std::shared_ptr<AllocatorT> allocator);

template<
    typename MessageT,
    typename AllocatorT = std::allocator<void>,
    typename PublisherT = ::rclcpp::Publisher<MessageT, AllocatorT>>
[[deprecated("use create_publisher(const std::string &, const rclcpp::QoS &, ...) instead")]]
std::shared_ptr<PublisherT>
create_publisher(
    const std::string & topic_name,
    const rmw_qos_profile_t & qos_profile = rmw_qos_profile_default,
    std::shared_ptr<AllocatorT> allocator = nullptr);

template<
    typename MessageT,
    typename CallbackT,
    typename Alloc = std::allocator<void>,
    typename SubscriptionT = rclcpp::Subscription<
        typename rclcpp::subscription_traits::has_message_type<CallbackT>::type, Alloc>>
[[deprecated("use create_subscription(const std::string &, const rclcpp::QoS &, CallbackT, ...) instead")]]
std::shared_ptr<SubscriptionT>
create_subscription(
    const std::string & topic_name,
    CallbackT && callback,
    const rmw_qos_profile_t & qos_profile = rmw_qos_profile_default,
    rclcpp::callback_group::CallbackGroup::SharedPtr group = nullptr,
    bool ignore_local_publications = false,
    typename rclcpp::message_memory_strategy::MessageMemoryStrategy<
        typename rclcpp::subscription_traits::has_message_type<CallbackT>::type, Alloc>::SharedPtr msg_mem_strat = nullptr,
    std::shared_ptr<Alloc> allocator = nullptr);

template<
    typename MessageT,
    typename CallbackT,
    typename Alloc = std::allocator<void>,
    typename SubscriptionT = rclcpp::Subscription<
        typename rclcpp::subscription_traits::has_message_type<CallbackT>::type, Alloc>>
[[deprecated("use create_subscription(const std::string &, const rclcpp::QoS &, CallbackT, ...) instead")]]
std::shared_ptr<SubscriptionT>
create_subscription(
    const std::string & topic_name,
    ...)
\end{verbatim}
Vulcanexus Documentation, Release 1.0.0

(continued from previous page)

CallbackT && callback,
size_t qos_history_depth,
rclcpp::callback_group::CallbackGroup::SharedPtr group = nullptr,
bool ignore_local_publications = false,
typename rclcpp::message_memory_strategy::MessageMemoryStrategy<
typename rclcpp::subscription_traits::has_message_type<CallbackT>::type, Alloc>
˓→::SharedPtr
msg_mem_strat = nullptr,
std::shared_ptr<Alloc> allocator = nullptr);

The change to how QoS is passed is most likely to impact users.
A typical change for a publisher looks like this:
- pub_ = create_publisher<std_msgs::msg::String>("chatter");
+ pub_ = create_publisher<std_msgs::msg::String>("chatter", 10);
And for a subscription:
- sub_ = create_subscription<std_msgs::msg::String>("chatter", callback);
+ sub_ = create_subscription<std_msgs::msg::String>("chatter", 10, callback);
If you have no idea what depth to use and don’t care right now (maybe just prototyping), then we recommend using 10,
as that was the default before and should preserve existing behavior.
More in depth documentation about how to select an appropriate depth is forthcoming.
This is an example of a slightly more involved change to avoid the newly deprecated API’s:
-

+
+
+

// Creates a latched topic
rmw_qos_profile_t qos = rmw_qos_profile_default;
qos.depth = 1;
qos.durability = RMW_QOS_POLICY_DURABILITY_TRANSIENT_LOCAL;
model_xml_.data = model_xml;
node_handle->declare_parameter("robot_description", model_xml);
description_pub_ = node_handle->create_publisher<std_msgs::msg::String>(
"robot_description", qos);
"robot_description",
// Transient local is similar to latching in ROS 1.
rclcpp::QoS(1).transient_local());

See the pull request (and connected pull requests) that introduced the QoS change for more examples and details:
• https://github.com/ros2/rclcpp/pull/713
– https://github.com/ros2/demos/pull/332
– https://github.com/ros2/robot_state_publisher/pull/19
– and others. . .

4.8. ROS 2 Documentation

503


Changes Due to Declare Parameter Change

For details about the actual behavior change, see Declaring Parameters above.

There are several new API calls in the rclcpp::Node’s interface:

- Methods that declare parameters given a name, optional default value, optional descriptor, and return the value actually set:

```cpp
const rclcpp::ParameterValue &
rclcpp::Node::declare_parameter(
  const std::string & name,
  const rclcpp::ParameterValue & default_value = rclcpp::ParameterValue(),
  const rcl_interfaces::msg::ParameterDescriptor & parameter_descriptor =
  rcl_interfaces::msg::ParameterDescriptor());
```

```cpp
template<typename ParameterT>
auto
rclcpp::Node::declare_parameter(
  const std::string & name,
  const ParameterT & default_value,
  const rcl_interfaces::msg::ParameterDescriptor & parameter_descriptor =
  rcl_interfaces::msg::ParameterDescriptor());
```

```cpp
template<typename ParameterT>
std::vector<ParameterT>
 rclcpp::Node::declare_parameters(
  const std::string & namespace_,
  const std::map<std::string, ParameterT> & parameters);
```

```cpp
template<typename ParameterT>
std::vector<ParameterT>
 rclcpp::Node::declare_parameters(
  const std::string & namespace_,
  const std::map<std::string,
    std::pair<ParameterT, rcl_interfaces::msg::ParameterDescriptor> > & parameters);
```

- A method to undeclare parameters and to check if a parameter has been declared:

```cpp
void
rclcpp::Node::undeclare_parameter(const std::string & name);
```

```cpp
bool
rclcpp::Node::has_parameter(const std::string & name) const;
```

- Some convenience methods that did not previously exist:

```cpp
rcl_interfaces::msg::SetParametersResult
rclcpp::Node::set_parameter(const rclcpp::Parameter & parameter);
```

```cpp
std::vector<rclcpp::Parameter>
 rclcpp::Node::get_parameters(const std::vector<std::string> & names) const;
```

(continues on next page)
rcl_interfaces::msg::ParameterDescriptor
rclcpp::Node::describe_parameter(const std::string & name) const;

• A new method to set the callback which is called anytime a parameter will be changed, giving you the opportunity to reject it:

```cpp
using OnParametersSetCallbackType =
  rclcpp::node_interfaces::NodeParametersInterface::OnParametersSetCallbackType;

OnParametersSetCallbackType
rclcpp::Node::set_on_parameters_set_callback(
  OnParametersSetCallbackType callback);
```

There were also several deprecated methods:

```cpp
template<typename ParameterT>
[[deprecated("use declare_parameter() instead")]]
void
rclcpp::Node::set_parameter_if_not_set(
  const std::string & name,
  const ParameterT & value);

template<typename ParameterT>
[[deprecated("use declare_parameters() instead")]]
void
rclcpp::Node::set_parameters_if_not_set(
  const std::string & name,
  const std::map<std::string, ParameterT> & values);

template<typename ParameterT>
[[deprecated("use declare_parameter() and it's return value instead")]]
void
rclcpp::Node::get_parameter_or_set(
  const std::string & name,
  ParameterT & value,
  const ParameterT & alternative_value);

template<typename CallbackT>
[[deprecated("use set_on_parameters_set_callback() instead")]]
void
rclcpp::Node::register_param_change_callback(CallbackT && callback);
```
**Memory Strategy**

The interface `rclcpp::memory_strategy::MemoryStrategy` was using the typedef `WeakNodeVector` in various method signatures. As of Dashing the typedef has been changed to `WeakNodeList` and with it the type of the parameter in various methods. Any custom memory strategy needs to be updated to match the modified interface.

The relevant API change can be found in `ros2/rclcpp#741`.

**rclcpp_components**

The correct way to implement composition in Dashing is by utilizing the `rclcpp_components` package.

The following changes must be made to nodes in order to correctly implement runtime composition:

The Node must have a constructor that takes `rclcpp::NodeOptions`:

```cpp
class Listener : public rclcpp::Node {
  Listener(const rclcpp::NodeOptions & options)
  : Node("listener", options)
  {
  }
};
```

C++ registration macros (if present) need to be updated to use the `rclcpp_components` equivalent. If not present, registration macros must be added in one translation unit.

```cpp
#include "rclcpp_components/register_node_macro.hpp"
// Use fully-qualified name in registration
RCLCPP_COMPONENTS_REGISTER_NODE(composition::Listener);
```

CMake registration macros (if present) need to be updated. If not present, registration macros must be added to the project's CMake.

```cpp
add_library(listener src/listener.cpp)
rclcpp_components_register_nodes(listener "composition::Listener")
```

For more information on composition, see the tutorial

**rclpy**

**Changes to Creating Publishers, Subscriptions, and QoS Profiles**

Prior to Dashing, you could optionally provide a `QoSProfile` object when creating a publisher or subscription. In an effort to encourage users to specify a history depth for message queues, we now **require** that a depth value or `QoSProfile` object is given when creating publishers or subscriptions.

To create a publisher, previously you would have written:

```cpp
node.create_publisher(Empty, 'chatter')
# Or using a keyword argument for QoSProfile
node.create_publisher(Empty, 'chatter', qos_profile=qos_profile_sensor_data)
```

In Dashing, prefer the following API that provides a depth value or `QoSProfile` object as a third positional argument:
Likewise for subscriptions, previously you would have written:

```python
node.create_subscription(BasicTypes, 'chatter', lambda msg: print(msg))
# Or using a keyword argument for QoSProfile
node.create_subscription(BasicTypes, 'chatter', lambda msg: print(msg), qos_profile=qos_profile_sensor_data)
```

In Dashing:

```python
# Assume a history setting of KEEP_LAST with depth 10
node.create_subscription(BasicTypes, 'chatter', lambda msg: print(msg), 10)
# Or pass a QoSProfile object directly
node.create_subscription(BasicTypes, 'chatter', lambda msg: print(msg), qos_profile=qos_profile_sensor_data)
```

To ease the transition, users who do not use the new API will see deprecation warnings.

Furthermore, we also require that when constructing QoSProfile objects that a history policy and/or depth is set. If a history policy of KEEP_LAST is provided, then a depth argument is also required. For example, these calls are valid:

```python
QoSProfile(history=QoSHistoryPolicy.RMW_QOS_POLICY_HISTORY_KEEP_ALL)
QoSProfile(history=QoSHistoryPolicy.RMW_QOS_POLICY_HISTORY_KEEP_LAST, depth=10)
QoSProfile(depth=10)  # equivalent to the previous line
```

And these calls will cause a deprecation warning:

```python
QoSProfile()
QoSProfile(reliability=QoSReliabilityPolicy.RMW_QOS_POLICY_RELIABILITY_BEST_EFFORT)
# KEEP_LAST but no depth
QoSProfile(history=QoSHistoryPolicy.RMW_QOS_POLICY_HISTORY_KEEP_LAST)
```

See the issue and pull request related to introducing this change for more details:

- https://github.com/ros2/rclpy/issues/342
- https://github.com/ros2/rclpy/pull/344

### Changes Due to Declare Parameter Change

For details about the actual behavior change, see Declaring Parameters above. The changes are analogous to the ones in rclcpp.

These are the new API methods available in rclpy.node.Node interface:

- To declare parameters given a name, an optional default value (supported by rcl_interfaces.msg. ParameterValue) and an optional descriptor, returning the value actually set:

```python
def declare_parameter(
    name: str,
    value: Any = None,
) -> ParameterValue:
  pass
```

(continues on next page)
To undeclare previously declared parameters and to check if a parameter has been declared beforehand:

```python
def undeclare_parameter(name: str) -> None
def has_parameter(name: str) -> bool
```

• To get and set parameter descriptors:

```python
def describe_parameter(name: str) -> ParameterDescriptor
def describe_parameters(names: List[str]) -> List[ParameterDescriptor]
def set_descriptor(
    name: str,
    descriptor: ParameterDescriptor,
    alternative_value: Optional[ParameterValue] = None
) -> ParameterValue
```

• A convenience method to get parameters that may not have been declared:

```python
def get_parameter_or(name: str, alternative_value: Optional[Parameter] = None) -> Parameter
```

**Other changes**

`rclpy.parameter.Parameter` can now guess its type without explicitly setting it (as long as it’s one of the supported ones by `rcl_interfaces.msg.ParameterValue`). For example, this code:

```python
p = Parameter('myparam', Parameter.Type.DOUBLE, 2.41)
```

Is equivalent to this code:

```python
p = Parameter('myparam', value=2.41)
```

This change does not break existing API.
rosidl

Until Crystal each message generator package registered itself using the `ament_cmake` extension point `rosidl_generate_interfaces` and was passed a set of `.msg` / `.srv` / `.action` files. As of Dashing the message generation pipeline is based on `.idl` files instead.

Any message generator package needs to change and register itself using the new extension point `rosidl_generate_idl_interfaces` which passes only `.idl` files instead. The message generators for the commonly supported languages C, C++, and Python as well as the typesupport packages for introspection, Fast RTPS, Connext and OpenSplice have already been updated (see ros2/rosidl#334). The CMake code calling `rosidl_generate_interfaces()` can either pass `.idl` files directly or pass `.msg` / `.srv` / `.action` which will then internally be converted into `.idl` files before being passed to each message generator.

The format of `.msg` / `.srv` / `.action` files is not being evolved in the future. The mapping between `.msg` / `.srv` / `.action` files and `.idl` files is described in this design article. A second design article describes the supported features in `.idl` files. In order to leverage any of the new features existing interfaces need to be converted (e.g. using the command line tools `msg2idl` / `srv2idl` / `action2idl`).

To distinguish same type names, but with different namespaces, the introspection structs now contain a namespace field that replaces the package name (see ros2/rosidl#335).

Mapping of char in `.msg` files

In ROS 1 `char` has been deprecated for a long time and is being mapped to `uint8`. In ROS 2 until Crystal `char` was mapped to a single character (`char` in C / C++, `str` with length 1 in Python) in an effort to provide a more natural mapping. As of Dashing the ROS 1 semantic has been restored and `char` maps to `uint8` again.

rosidl_generator_cpp

The C++ data structures generated for messages, services and actions provide setter methods for each field. Until Crystal each setter returned a pointer to the data structure itself to enable the named parameter idiom. As of Dashing these setters return a reference instead since that seems to be the more common signature as well as it clarifies that the returned value can’t be a `nullptr`.

rosidl_generator_py

Until Crystal an array (fixed size) or sequence (dynamic size, optionally with an upper boundary) field in a message was stored as a `list` in Python. As of Dashing the Python type for arrays / sequences of numeric values has been changed:

- an array of numeric values is stored as a `numpy.ndarray` (the `dtype` is chosen to match the type of the numeric value)
- a sequence of numeric values is stored as an `array.array` (the `typename` is chosen to match the type of the numeric value)

As before an array / sequence of non-numeric types is still represented as a `list` in Python.

This change brings a number of benefits:

- The new data structures ensure that each item in the array / sequence complies with the value range restrictions of the numeric type.
- The numeric values can be stored more efficiently in memory which avoid the overhead of Python objects for each item.
• The memory layout of both data structures allows to read and write all items of the array / sequence in a single operation which makes the conversion from and to Python significantly faster / more efficient.

launch

The launch_testing package caught up with the launch package redesign done in Bouncy Bolson. The legacy Python API, already moved into the launch.legacy submodule, has thus been deprecated and removed. See launch examples and documentation for reference on how to use its new API. See demos tests for reference on how to use the new launch_testing API.

rmw

Changes since the Crystal Clemmys <Release-Crystal-Clemmys> release:

• New API in rmw, a fini function for rmw_context_t:
  • rmw_context_fini

• Modification of rmw, now passes rmw_context_t to rmw_create_wait_set:
  • rmw_create_wait_set

• New APIs in rmw for preallocating space for published and subscribed messages:
  • rmw_init_publisher_allocation
  • rmw_fini_publisher_allocation
  • rmw_init_subscription_allocation
  • rmw_fini_subscription_allocation
  • rmw_serialized_message_size

• Modification of rmw, now passes rmw_publisher_allocation_t or rmw_subscription_allocation_t to rmw_publish and rmw_take, respectively. Note that this argument can be NULL or nullptr, which keeps existing Crystal behavior.
  • rmw_publish
  • rmw_take

• Type names returned by rmw_get_*_names_and_types* functions should have a fully-qualified namespace. For example, instead of rcl_interfaces/Parameter and rcl_interfaces/GetParameters, the returned type names should be rcl_interface/msg/Parameter and rcl_interfaces/srv/GetParameters.

actions

• Changes to rclcpp_action::Client signatures:
  The signature of rclcpp_action::Client::async_send_goal has changed. Now users can optionally provide callback functions for the goal response and the result using the new SendGoalOptions struct. The goal response callback is called when an action server accepts or rejects the goal and the result callback is called when the result for the goal is received. Optional callbacks were also added to rclcpp_action::Client::async_cancel_goal and rclcpp_action::Client::async_get_result.
• Changes to goal transition names:

The names of goal state transitions have been refactored to reflect the design documentation. This affects rcl_action, rclcpp_action, and rclpy. Here is a list of the event name changes (Old name -> New name):

- GOAL_EVENT_CANCEL -> GOAL_EVENT_CANCEL_GOAL
- GOAL_EVENT_SET_SUCCEEDED -> GOAL_EVENT_SUCCEED
- GOAL_EVENT_SET_ABORTED -> GOAL_EVENT_ABORT
- GOAL_EVENT_SET_CANCELED -> GOAL_EVENT_CANCELED

• Changes to CancelGoal.srv:

A return_code field was added to the response message of the CancelGoal service. This is to better communicate a reason for a failed service call. See the pull request and connected issue for details.

rviz

• Plugins should use fully qualified type names otherwise a warning will be logged. For example, use the type sensor_msgs/msg/Image instead of sensor_msgs/Image. See PR introducing this change for more details.

Known Issues

• [ros2/rclcpp#715] There is an inconsistency in the way that parameter YAML files are loaded between standalone ROS 2 nodes and composed ROS 2 nodes. Currently available workarounds are noted in an issue comment

• [ros2/rclpy#360] rclpy nodes ignore ctrl-c when using OpenSplice on Windows.

• [ros2/rosidl_typesupport_opensplice#30] There is a bug preventing nesting a message inside of a service or action definition with the same name when using OpenSplice.

• [ros2/rclcpp#781] Calling get_parameter/list_parameter from within on_set_parameter_callback causes a deadlock on Dashing. This is fixed for Eloquent, but is an ABI break so has not been backported to Dashing.

• [ros2/rclcpp#912] Inter-process communication forces a message copy when intra-process communication takes place between an std::unique_ptr publisher and a single std::unique_ptr subscription (published std::unique_ptr is internally being promoted to an std::shared_ptr).

• [ros2/rosbag2#125] Topics with unreliable QOS are not recorded.

• [ros2/rclcpp#715] Composable nodes cannot receive parameters via remapping. Supplying parameters to composable nodes can be accomplished using the methods described in [this comment].

• [ros2/rclcpp#893] rclcpp::Context is not destroyed because of a reference cycle with rclcpp::GraphListener. This causes a memory leak. A fix has not been backported because of the risk of breaking ABI.
Timeline before the release

A few milestones leading up to the release:

**Mon. Apr 8th (alpha)** First releases of core packages available. Testing can happen from now on (some features might not have landed yet).

**Thu. May 2nd** API freeze for core packages

**Mon. May 6th (beta)** Updated releases of core packages available. Additional testing of the latest features.

**Thu. May 16th** Feature freeze. Only bug fix releases should be made after this point. New packages can be released independently.

**Mon. May 20th (release candidate)** Updated releases of core packages available.

**Wed. May 29th** Freeze rosdistro. No PRs for Dashing on the rosdistro repo will be merged (reopens after the release announcement).

Crystal Clemmys (crystal)

Table of Contents

- Supported Platforms
- New features in this ROS 2 release
- Changes since the Bouncy release
- Known Issues

Crystal Clemmys is the third release of ROS 2.

**Supported Platforms**

Crystal Clemmys is primarily supported on the following platforms (see REP 2000 for full details):

Tier 1 platforms:
- Ubuntu 18.04 (Bionic)
- Mac macOS 10.12 (Sierra)
- Windows 10

Tier 2 platforms:
- Ubuntu 16.04 (Xenial)
New features in this ROS 2 release

- Actions in C / C++ (server / client examples)
- gazebo_ros_pkgs
- image_transport
- navigation2
- rosbag2
- rqt <../../Concepts/Intermediate/About-RQt>
- Improvement in memory management
- Introspection information about nodes
- Launch system improvements
  - Arguments
  - Nested launch files
  - Conditions
  - Pass params to Nodes
- Laid the groundwork for file-based logging and /rosout publishing
- Time and Duration API in Python
- Parameters work with Python nodes

Changes since the Bouncy release

Changes since the Bouncy Bolson <Release-Bouncy-Bolson> release:

- geometry2 - tf2::Buffer API Change

  tf2::Buffer now uses rclcpp::Time, with the constructor requiring a shared_ptr to a
  rclcpp::Clock instance. See https://github.com/ros2/geometry2/pull/67 for details, with example usage:

  ```
  #include <tf2_ros/transform_listener.h>
  #include <rclcpp/rclcpp.hpp>
  ...
  # Assuming you have a rclcpp::Node my_node
  tf2_ros::Buffer buffer(my_node.get_clock());
  tf2_ros::TransformListener tf_listener(buffer);
  ```

- All rclcpp and rcutils logging macros require semicolons.
  See https://github.com/ros2/rcutils/issues/113 for details.

- rcutils_get_error_string_safe() and rcl_get_error_string_safe() have been replaced with
  rcutils_get_error_string().str and rcl_get_error_string().str.
  See https://github.com/ros2/rcutils/pull/121 for details.

- rmw - rmw_init API Change

  There are two new structs, the rcl_context_t and the rcl_init_options_t, which are used with rmw_init.
  The init options struct is used to pass options down to the middleware and is an input to rmw_init. The context
is a handle which is an output of `rmw_init` function is used to identify which init-shutdown cycle each entity is associated with, where an “entity” is anything created like a node, guard condition, etc.

This is listed here because maintainers of alternative rmw implementations will need to implement these new functions to have their rmw implementation work in Crystal.

This is the function that had a signature change:

- `rmw_init`

Additionally, there are these new functions which need to be implemented by each rmw implementation:

- `rmw_shutdown`
- `rmw_init_options_init`
- `rmw_init_options_copy`
- `rmw_init_options_fini`

Here’s an example of what minimally needs to be changed in an rmw implementation to adhere to this API change:

- `rmw_fastrtps pr`

**rcl - rcl_init API Change**

Like the `rmw` change above, there’s two new structs in `rcl` called `rcl_context_t` and `rcl_init_options_t`. The init options are passed into `rcl_init` as an input and the context is passed in as an output. The context is used to associate all other rcl entities to a specific init-shutdown cycle, effectively making init and shutdown no longer global functions, or rather those functions no longer use an global state and instead encapsulate all state within the context type.

Any maintainers of a client library implementation (that also uses rcl under the hood) will need to make changes to work with Crystal.

These functions were removed:

- `rcl_get_global_arguments`
- `rcl_get_instance_id`
- `rcl_ok`

These functions had signature changes:

- `rcl_init`
- `rcl_shutdown`
- `rcl_guard_condition_init`
- `rcl_guard_condition_init_from_rmw`
- `rcl_node_init`
- `rcl_timer_init`

These are the new functions and types:

- `rcl_context_t`
- `rcl_get_zero_initialized_context`
- `rcl_context_fini`
- `rcl_context_get_init_options`
- `rcl_context_get_instance_id`
These new and changed functions will impact how you handle init and shutdown in your client library. For examples, look at the following rclcpp and rclpy PR’s:

- rclcpp
- rclpy

However, you may just continue to offer a single, global init and shutdown in your client library, and just store a single global context object.

**Known Issues**

- A race condition in Fast-RTPS 1.7.0 may cause messages to drop under stress (Issue).
- Using the TRANSIENT_LOCAL QoS setting with rmw_fastrtps_cpp can crash applications with large messages (Issue).
- Cross-vendor communication between rmw_fastrtps_cpp and other implementations is not functioning on Windows (Issue).
- When using OpenSplice (version < 6.9.190227) on macOS and Windows you might experience naming conflicts when when referencing field types with names from other packages if the same name also exist in the current package (Issue). By updating to a newer OpenSplice version as well as at least the third patch release of Crystal the problem should be resolved. On Linux updating to the latest Debian packages will include the newest OpenSplice version.

**Bouncy Bolson (bouncy)**

**Table of Contents**

- Supported Platforms
- Features
  - New features in this ROS 2 release
  - Changes since the Ardent release
- Known Issues

*Bouncy Bolson* is the second release of ROS 2.
Supported Platforms

This version of ROS 2 is supported on four platforms (see REP 2000 for full details):

- Ubuntu 18.04 (Bionic)
  - Debian packages for amd64 as well as arm64
- Ubuntu 16.04 (Xenial)
  - no Debian packages but building from source is supported
- Mac macOS 10.12 (Sierra)
- Windows 10 with Visual Studio 2017

Binary packages as well as instructions for how to compile from source are provided (see install instructions <../../Installation> as well as documentation).

Features

New features in this ROS 2 release

- New launch system <../Tutorials/Intermediate/Launch/Launch-system> featuring a much more capable and flexible Python API.
- Parameters can be passed as command line arguments <../How-To-Guides/Node-arguments> to C++ executables.
- Static remapping via command line arguments <../How-To-Guides/Node-arguments>.
- Various improvements to the Python client library.
- Support for publishing and subscribing to serialized data. This is the foundation for the upcoming work towards a native rosbag implementation.
- More command line tools <../../Concepts/Basic/About-Command-Line-Tools>, e.g. for working with parameters and lifecycle states.
- Binary packages / fat archives support three RMW implementations by default (without the need to build from source):
  - eProsima’s Fast RTPS (default)
  - RTI’s Connext
  - ADLINK’s OpenSplice

For an overview of all features available, including those from earlier releases, please see the Features <../../The-ROS2-Project/Features> page.

Changes since the Ardent release

Changes since the Ardent Apalone <Release-Ardent-Apalone> release:

- The Python package launch has been redesigned. The previous Python API has been moved into a submodule launch.legacy. You can update existing launch files to continue to use the legacy API if a transition to the new Python API is not desired.
- The ROS topic names containing namespaces are mapped to DDS topics including their namespaces. DDS partitions are not being used anymore for this.
• The recommended build tool is now colcon instead of ament_tools. This switch has no implications for the code in each ROS 2 package. The install instructions have been updated and the read-the-docs page describes how to map an existing ament_tools call to colcon.

• The argument order of this rclcpp::Node::create_subscription() signature has been modified.

Known Issues

• New-style launch files may hang on shutdown for some combinations of platform and RMW implementation.
• Static remapping of namespaces not working correctly when addressed to a particular node.
• Opensplice error messages may be printed when using ros2 param and ros2 lifecycle command-line tools.

Ardent Apalone (ardent)

Table of Contents

• Supported Platforms
• Features
  – New features in this ROS 2 release
  – Changes since Beta 3 release
• Known Issues

Welcome to the first non-beta release of ROS 2 software named Ardent Apalone!

Supported Platforms

This version of ROS 2 is supported on three platforms:

• Ubuntu 16.04 (Xenial)
• Mac macOS 10.12 (Sierra)
• Windows 10

Binary packages as well as instructions for how to compile from source are provided for all 3 platforms (see install instructions <../../Installation> as well as documentation).

Features

New features in this ROS 2 release

• Distributed discovery, publish / subscribe, request / response communication
  – Provided by a C API
  – Implemented using different vendors:
    ∗ eProsima’s Fast RTPS as well as ADLINK’s OpenSplice (from binary and source)
    ∗ RTI’s Connext (only from source)
– Numerous quality of service settings for handling non-ideal networks
– DDS Security support (with Connext and Fast RTPS)

• C++ and Python 3 client libraries
  – Sharing common code in C to unify the implementation
  – Execution model separated from the nodes, composable nodes
  – Node-specific parameters (only in C++ atm)
  – Life cycle (only in C++ atm)
  – Optionally intra-process communication using the same API (only in C++)

• Message definitions (with bounded arrays and strings as well as default values)
• Command line tools (e.g. `ros2 run`)
• `rviz` with a few display types (the Windows version will likely follow in a few weeks)
• File system-based resource index (querying information without recursive crawling)
• Realtime safe code paths for pub/sub (with compatible DDS implementations only)
• Bridge between ROS 1 and ROS 2
• HSR demo see Beta 3 <Beta3-Overview>
• Turtlebot demo see Beta 2 <Beta2-Overview>

For a more detailed description please see the `Features <../../The-ROS2-Project/Features>` page.

**Changes since Beta 3 release**

Improvements since the Beta 3 release:

• `rviz`
  • Different initialization options for message data structures in C++ (see design doc)
  • Logging API improvements, now also used in the demos
  • Time support in C++ with different clocks
  • `wait-for-service` support in the Python client library
  • Draft implementation of REP 149 specifying format 3 of the package manifest files

**Known Issues**

• Fast RTPS performance with larger data like the image demo
• Using Connext it is currently not allowed for two topics with the same base name but different namespaces to have a different type (see issue).
• Listing of node names (e.g. using `ros2 node list`) does not work across some rmw implementations.
• On Windows Python launch files might hang when trying to abort using Ctrl-C (see issue). In order to continue using the shell which is blocked by the hanging command you might want to end the hanging Python process using the process monitor.
Beta 3 (r2b3)

Table of Contents

- Supported Platforms
- Features
  - Improvements since Beta 2 release
  - New demo application
  - Selected features from previous Alpha/Beta releases
- Known issues

Supported Platforms

We support ROS 2 Beta 3 on three platforms: Ubuntu 16.04 (Xenial), macOS 10.12 (Sierra), and Windows 10. We provide both binary packages and instructions for how to compile from source for all 3 platforms (see install instructions <../../Installation> as well as documentation).

Features

Improvements since Beta 2 release

- Execution model in Python, many fixes to memory management in Python C extension
- Experimental rewrite of ros_control
- Exposure of DDS implementation-specific symbols to users (for Fast RTPS and Connext) (see example)
- Logging API in Python
- Fixed several memory leaks and race conditions in various packages
- Readded support for OpenSplice (on Linux and Windows atm) provided by PrismTech
- Use bloom (without patches) to make ROS 2 releases

New demo application

- HSR demo
  - Remote control a HSR robot using a ROS 2 joystick controller
  - Running the ros1.bridge in a Docker container on the HSR (since the robot is running ROS 1 on Ubuntu Trusty)
  - Run a ROS 2 development version of rviz to visualize sensor data from the robot etc. (see video)
Selected features from previous Alpha/Beta releases

For the complete list, see earlier release notes <../index>.

- C++ and Python implementations of ROS 2 client libraries including APIs for:
  - Publishing and subscribing to ROS topics
  - Requesting and replying ROS services (synchronous (C++ only) and asynchronous)
  - Getting and setting ROS parameters (C++ only, synchronous and asynchronous)
  - Timer callbacks
- Support for interoperability between multiple DDS/RTPS implementations
  - eProsima Fast RTPS is our default implementation, and is included in the binary packages
  - RTI Connext is supported: build from source to try it out
  - PrismTech OpenSplice: see limitations below
- A graph API for network events
- Distributed discovery
- Realtime safe code paths for publish and subscribe with compatible DDS implementation (only Connext at the moment)
  - Support for custom allocators
- ROS 1 <-> ROS 2 dynamic bridge node
- Executor threading model (C++ and Python)
- Component model to compose nodes at compile / link / runtime
- Managed component using a standard lifecycle
- Extended .msg format with new features:
  - Bounded arrays
  - Default values

Known issues

- On Windows Python launch files might hang when trying to abort using Ctrl-C (see issue). In order to continue using the shell which is blocked by the hanging command you might want to end the hanging Python process using the process monitor.
- OpenSplice support is currently not available for MacOS. Also access to native handles is not yet implemented.
- Using Connext it is currently not allowed for two topics with the same base name but different namespaces to have a different type (see issue).
Beta 2 (r2b2)

Table of Contents

• Supported Platforms
• Features
  – Improvements since Beta 1 release
  – New demo application
  – Selected features from previous Alpha/Beta releases
  – Known issues

Supported Platforms

We support ROS 2 Beta 2 on three platforms: Ubuntu 16.04 (Xenial), macOS 10.12 (Sierra), and Windows 10. We provide both binary packages and instructions for how to compile from source for all 3 platforms (see install instructions <../../Installation> as well as documentation).

Features

Improvements since Beta 1 release

• DDS_Security support (aka SROS2, see sros2)
• Debian packages for Ubuntu Xenial
• Typesupport has been redesigned so that you only build a single executable and can choose one of the available RMW implementations by setting an environment variable (see documentation <../../How-To-Guides/Working-with-multiple-RMW-implementations>).
• Namespace support for nodes and topics (see design article, see known issues below).
• A set of command-line tools using the extensible ros2 command (see conceptual article <../../Concepts/Basic/About-Command-Line-Tools>).
• A set of macros for logging messages in C / C++ (see API docs of rcutils).

New demo application

• Turtlebot 2 demos using the following repositories that have been (partially) converted to ROS 2 (Linux only):
  – ros_astra_camera
  – depthimage_to_laserscan
  – pcl_conversions
  – cartographer
  – cartographer_ros
  – ceres-solver
Selected features from previous Alpha/Beta releases

For the complete list, see earlier release notes <../index>.

• C++ and Python implementations of ROS 2 client libraries including APIs for:
  – Publishing and subscribing to ROS topics
  – Requesting and replying ROS services (synchronous (C++ only) and asynchronous)
  – Getting and setting ROS parameters (C++ only, synchronous and asynchronous)
  – Timer callbacks
• Support for interoperability between multiple DDS/RTPS implementations
  – eProsima Fast RTPS is our default implementation, and is included in the binary packages
  – RTI Connext is supported: build from source to try it out
  – We initially supported PrismTech OpenSplice but support for it is currently on hold
• A graph API for network events
• Distributed discovery
• Realtime safe code paths for publish and subscribe with compatible DDS implementation (only Connext at the moment)
  – Support for custom allocators
• ROS 1 <-> ROS 2 dynamic bridge node
• Executor threading model (C++ only)
• Component model to compose nodes at compile / link / runtime
• Managed component using a standard lifecycle
• Extended .msg format with new features:
  – Bounded arrays
  – Default values
Known issues

- We’re tracking issues in various repositories, but the main entry point is the ros2/ros2 issue tracker.
- We’d like to highlight a known issue that we are looking into which doesn’t allow two topics with the same base name but different namespaces to have a different type when using rmw_connext_cpp.
- Services with long responses are not working with Fast-RTPS. The fix, while not being part of beta2, is available upstream so you can work around this issue by building from source using Fast-RTPS master branch.

Beta 1 (Asphalt)

Table of Contents

- Supported Platforms
- Features
  - Improvements since Alpha 8 release
  - Selected features from previous Alpha releases
  - Known issues

Supported Platforms

We support ROS2 Beta 1 on three platforms: Ubuntu 16.04 (Xenial), Mac OS X 10.11 (El Capitan), and Windows 8.1 and 10. We provide both binary packages and instructions for how to compile from source for all 3 platforms.

Features

Improvements since Alpha 8 release

- Support for node composition at compile, link, or runtime.
- A standard lifecycle for managed nodes.
- Improved support for Quality of Service tuning and tests.
- New and updated design documents
- More tutorials <../../Tutorials> and examples
- Bridging services to / from ROS 1 (in addition to topics)
Selected features from previous Alpha releases

For the complete list, see earlier release notes <../index>.

- C++ and Python implementations of ROS 2 client libraries including APIs for:
  - Publishing and subscribing to ROS topics
  - Requesting and replying ROS services (synchronous (C++ only) and asynchronous)
  - Getting and setting ROS parameters (C++ only, synchronous and asynchronous)
  - Timer callbacks
  - Support for interoperability between multiple DDS/RTPS implementations
  - eProsima Fast RTPS is our default implementation, and is included in the binary packages
  - RTI Connext is supported: build from source to try it out
  - We initially supported PrismTech OpenSplice but eventually decided to drop it
- A graph API for network events
- Distributed discovery
- Realtime safe code paths for publish and subscribe with compatible DDS implementation (only Connext at the moment)
  - Support for custom allocators
- ROS 1 <-> ROS 2 dynamic bridge node
- Executor threading model in C++
- Extended .msg format with new features:
  - Bounded arrays
  - Default values

Known issues

- We’re tracking issues in various repositories, but the main entry point is the ros2/ros2 issue tracker
- We’d like to highlight a known issue that we are working with eProsima to fix that results in significantly degraded performance for large messages under FastRTPS. This will be observed when running some of the demos with larger image resolutions.

Alphas

Table of Contents

- ROS 2 alpha8 release (code name Hook-and-Loop; October 2016)
  - Changes to supported DDS vendors
  - Scope
- ROS 2 alpha7 release (code name Glue Gun; July 2016)
This is a merged version of the previously separated pages for the 8 alpha releases of ROS 2.

We hope that you try them out and provide feedback <../../Contact>.
ROS 2 alpha8 release (code name *Hook-and-Loop*; October 2016)

Changes to supported DDS vendors

ROS 2 supports multiple middleware implementations (see this page `<../../Concepts/Intermediate/About-Different-Middleware-Vendors>` for more details). Until Alpha 8, ROS 2 was supporting ROS middleware implementations for eProsima's Fast RTPS, RTI's Connext and PrismTech's OpenSplice. To streamline our efforts, as of Alpha 8, Fast RTPS and Connext (static) will be supported, with Fast RTPS (now Apache 2.0-licensed) shipped as the default.

Scope

As the “alpha” qualifier suggests, this release of ROS 2 is far from complete. You should not expect to switch from ROS 1 to ROS 2, nor should you expect to build a new robot control system with ROS 2. Rather, you should expect to try out some demos, explore the code, and perhaps write your own demos.

The improvements included in this release are:

- Several improvements to Fast RTPS and its rmw implementation
  - Support for large (image) messages in Fast RTPS
  - `wait_for_service` functionality in Fast RTPS
- Support for all ROS 2 message types in Python and C
- Added support for Quality of Service (QoS) settings in Python
- Fixed various bugs with the previous alpha release

Pretty much anything not listed above is not included in this release. The next steps are described in the Roadmap `<../../The-ROS2-Project/Roadmap>`.

ROS 2 alpha7 release (code name *Glue Gun*; July 2016)

Table of Contents

- New version of Ubuntu required
- Scope

New version of Ubuntu required

Until Alpha 6 ROS 2 was targeting Ubuntu Trusty Tahr (14.04). As of this Alpha ROS 2 is targeting Ubuntu Xenial Xerus (16.04) to benefit from newer versions of the compiler, CMake, Python, etc.
Scope

As the “alpha” qualifier suggests, this release of ROS 2 is far from complete. You should not expect to switch from ROS 1 to ROS 2, nor should you expect to build a new robot control system with ROS 2. Rather, you should expect to try out some demos, explore the code, and perhaps write your own demos.

The major features included in this release are:

• Graph API functionality: wait_for_service
  – Added interfaces in rclcpp and make use of them in examples, demos, and tests
• Improved support for large messages in both Connext and Fast-RTPS (partial for Fast-RTPS)
• Turtlebot demo using ported code from ROS 1
  – See: https://github.com/ros2/turtlebot2_demo

Pretty much anything not listed above is not included in this release. The next steps are described in the Roadmap <../../The-ROS2-Project/Roadmap>.

ROS 2 alpha6 release (code name Fastener; June 2016)

Table of Contents

• Scope

Scope

As the “alpha” qualifier suggests, this release of ROS 2 is far from complete. You should not expect to switch from ROS 1 to ROS 2, nor should you expect to build a new robot control system with ROS 2. Rather, you should expect to try out some demos, explore the code, and perhaps write your own demos.

The major features included in this release are:

• Graph API functionality: wait_for_service
  – Added graph guard condition to nodes for waiting on graph changes
  – Added rmw_service_server_is_available for verifying if a service is available
• Refactored rclcpp to use rcl
• Improved support for complex message types in Python
  – Nested messages
  – Arrays
  – Strings

Pretty much anything not listed above is not included in this release. The next steps are described in the Roadmap <../../The-ROS2-Project/Roadmap>.
ROS 2 alpha5 release (code name Epoxy; April 2016)

Table of Contents

- Scope

Scope

As the “alpha” qualifier suggests, this release of ROS 2 is far from complete. You should not expect to switch from
ROS 1 to ROS 2, nor should you expect to build a new robot control system with ROS 2. Rather, you should expect to
try out some demos, explore the code, and perhaps write your own demos.

The major features included in this release are:

- Support for C data structures in Fast RTPS and Connext Dynamic rmw implementations.
- Support services in C.
- Added 32-bit and 64-bit ARM as experimentally supported platforms.

Pretty much anything not listed above is not included in this release. The next steps are described in the Roadmap
<../../The-ROS2-Project/Roadmap>.

ROS 2 alpha4 release (code name Duct tape; February 2016)

Table of Contents

- Background
  - Status
  - Intended audience
  - Scope

Background

As explained in a design article, we are engaged in the development of a new major version of ROS, called “ROS 2.”
While the underlying concepts (e.g., publish / subscribe messaging) and goals (e.g., flexibility and reusability) are the
same as for ROS 1, we are taking this opportunity to make substantial changes to the system, including changing some
of the core APIs. For a deeper treatment of those changes and their rationale, consult the other ROS 2 design articles.
**Status**

On February 17, 2016, we are releasing ROS 2 alpha4, code-named **Duct tape**. Our primary goal with this release is to add more features, while also addressing the feedback we received for the previous releases. To that end, we built a set of demos <../../Tutorials> that show some of the key features of ROS 2. We encourage you to try out those demos, look at the code that implements them, and provide feedback <../../Contact>. We’re especially interested to know how well (or poorly) we’re addressing use cases that are important to you.

**Intended audience**

While everyone is welcome to try out the demos and look through the code, we’re aiming this release at people who are already experienced with ROS 1 development. At this point, the ROS 2 documentation is pretty sparse and much of the system is explained by way of how it compares to ROS 1.

**Scope**

As the “alpha” qualifier suggests, this release of ROS 2 is far from complete. You should not expect to switch from ROS 1 to ROS 2, nor should you expect to build a new robot control system with ROS 2. Rather, you should expect to try out some demos, explore the code, and perhaps write your own demos.

The major features included in this release are:

- Improved type support infrastructure, including support for C
- Preliminary Python client library, only publishers and subscriptions are supported. Beware, the API is subject to change and is far from complete!
- Added structures for ROS time in C API (still needs C++ API)
  - New concept of extensible “time sources” for ROS Time, the default time source will be like ROS 1 (implementation pending)

Pretty much anything not listed above is not included in this release. The next steps are described in the *Roadmap* <../../The-ROS2-Project/Roadmap>.

**ROS 2 alpha3 release (code name Cement; December 2015)**

<table>
<thead>
<tr>
<th>Table of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Background</em></td>
</tr>
<tr>
<td><em>Status</em></td>
</tr>
<tr>
<td><em>Intended audience</em></td>
</tr>
<tr>
<td><em>Scope</em></td>
</tr>
</tbody>
</table>
Background

As explained in a design article, we are engaged in the development of a new major version of ROS, called “ROS 2.” While the underlying concepts (e.g., publish / subscribe messaging) and goals (e.g., flexibility and reusability) are the same as for ROS 1, we are taking this opportunity to make substantial changes to the system, including changing some of the core APIs. For a deeper treatment of those changes and their rationale, consult the other ROS 2 design articles.

Status

On December 18, 2015, we are releasing ROS 2 alpha3, code-named Cement. Our primary goal with this release is to add more features, while also addressing the feedback we received for the previous releases. To that end, we built a set of demos </a> that show some of the key features of ROS 2. We encourage you to try out those demos, look at the code that implements them, and provide feedback </a>. We’re especially interested to know how well (or poorly) we’re addressing use cases that are important to you.

Intended audience

While everyone is welcome to try out the demos and look through the code, we’re aiming this release at people who are already experienced with ROS 1 development. At this point, the ROS 2 documentation is pretty sparse and much of the system is explained by way of how it compares to ROS 1.

Scope

As the “alpha” qualifier suggests, this release of ROS 2 is far from complete. You should not expect to switch from ROS 1 to ROS 2, nor should you expect to build a new robot control system with ROS 2. Rather, you should expect to try out some demos, explore the code, and perhaps write your own demos.

The major features included in this release are:

- Updated rcl interface.
  - This interface will be wrapped in order to create language bindings, e.g. rclpy.
  - This interface has improved documentation and test coverage over existing interfaces we currently have, e.g. rmw and rclcpp.
  - See rcl headers.
- Added support in rclcpp for using the TLSF (two-level segregate fit) allocator, a memory allocator design for embedded and real-time systems.
- Improved efficiency of MultiThreadedExecutor and fixed numerous bugs with multi-threaded execution, which is now test on CI.
- Added ability to cancel an Executor from within a callback called in spin.
- Added ability for a timer to cancel itself by supporting a Timer callback that accepts a reference to itself as a function parameter.
- Added checks for disallowing multiple threads to enter Executor::spin.
- Improved reliability of numerous tests that had been sporadically failing.
- Added support for using Fast RTPS (instead of, e.g., OpenSplice or Connext).
- A partial port of tf2 including the core libraries and core command line tools.
Pretty much anything not listed above is not included in this release. The next steps are described in the Roadmap.

ROS 2 alpha2 release (code name Baling wire; October 2015)

Table of Contents

- Background
- Status
- Intended audience
- Scope

Background

As explained in a design article, we are engaged in the development of a new major version of ROS, called “ROS 2.” While the underlying concepts (e.g., publish / subscribe messaging) and goals (e.g., flexibility and reusability) are the same as for ROS 1, we are taking this opportunity to make substantial changes to the system, including changing some of the core APIs. For a deeper treatment of those changes and their rationale, consult the other ROS 2 design articles.

Status

On November 3, 2015, we are releasing ROS 2 alpha2, code-named Baling wire. Our primary goal with this release is to add more features, while also addressing the feedback we received for the previous alpha 1 release. To that end, we built a set of demos that show some of the key features of ROS 2. We encourage you to try out those demos, look at the code that implements them, and provide feedback. We're especially interested to know how well (or poorly) we're addressing use cases that are important to you.

Intended audience

While everyone is welcome to try out the demos and look through the code, we're aiming this release at people who are already experienced with ROS 1 development. At this point, the ROS 2 documentation is pretty sparse and much of the system is explained by way of how it compares to ROS 1.

Scope

As the “alpha” qualifier suggests, this release of ROS 2 is far from complete. You should not expect to switch from ROS 1 to ROS 2, nor should you expect to build a new robot control system with ROS 2. Rather, you should expect to try out some demos, explore the code, and perhaps write your own demos.

The major features included in this release are:

- Support for custom allocators in rclcpp, useful for real-time messaging
- Feature parity of Windows with Linux/OSX, including workspace management, services and parameters
- rclcpp API improvements
- FreeRTPS improvements
Pretty much anything not listed above is not included in this release. The next steps are described in the Roadmap <../../The-ROS2-Project/Roadmap>.

ROS 2 alpha1 release (code name Anchor; August 2015)

Table of Contents

- Background
- Status
- Intended audience
- Scope

Background

As explained in a design article, we are engaged in the development of a new major version of ROS, called “ROS 2.” While the underlying concepts (e.g., publish / subscribe messaging) and goals (e.g., flexibility and reusability) are the same as for ROS 1, we are taking this opportunity to make substantial changes to the system, including changing some of the core APIs. For a deeper treatment of those changes and their rationale, consult the other ROS 2 design articles.

Status

On August 31, 2015, we are releasing ROS 2 alpha1, code-named Anchor. Our primary goal with this release is to give you the opportunity to understand how ROS 2 works, in particular how it differs from ROS 1. To that end, we built a set of demos <../../Tutorials> that show some of the key features of ROS 2. We encourage you to try out those demos, look at the code that implements them, and provide feedback <../../Contact>. We’re especially interested to know how well (or poorly) we’re addressing use cases that are important to you.

Intended audience

While everyone is welcome to try out the demos and look through the code, we’re aiming this release at people who are already experienced with ROS 1 development. At this point, the ROS 2 documentation is pretty sparse and much of the system is explained by way of how it compares to ROS 1.

Scope

As the “alpha” qualifier suggests, this release of ROS 2 is far from complete. You should not expect to switch from ROS 1 to ROS 2, nor should you expect to build a new robot control system with ROS 2. Rather, you should expect to try out some demos, explore the code, and perhaps write your own demos.

The major features included in this release are:

- Discovery, transport, and serialization use DDS
- Support multiple DDS vendors
- Support messaging primitives: topics (publish / subscribe), services (request / response), and parameters
- Support Linux (Ubuntu Trusty), OS X (Yosemite) and Windows (8)
• Use quality-of-service settings to handle lossy networks <../Tutorials/Demos/Quality-of-Service>
• Communicate inter-process or intra-process with the same API <../Tutorials/Demos/Intra-Process-Communication>
• Write real-time safe code that uses the ROS 2 APIs <../Tutorials/Demos/Real-Time-Programming>
• Run ROS 2 on “bare-metal” microcontrollers (no operating system)
• Bridge communication between ROS 1 and ROS 2

Pretty much anything not listed above is not included in this release. The next steps are described in the Roadmap <../../The-ROS2-Project/Roadmap>.

Development process for a release

Each ROS 2 distribution goes through a process of development more than a year long that begins prior to the release of the previous distribution. Below is a high-level view of this development process. There is no specific due date for the items in this process, but in general earlier items should be completed before later items can be completed.

For the progress through this process for a specific release, see that release’s documentation page.
### Vulcannexus Documentation, Release 1.0.0

<table>
<thead>
<tr>
<th>Item</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Find the ROS Boss</strong></td>
<td>The “ROS Boss” is the person in charge of shepherding a distribution through the development, release, update, and EOL'ing stages of its life. They are chosen from the internal ROS 2 team at Open Robotics.</td>
</tr>
<tr>
<td><strong>Run process to choose the distribution name</strong></td>
<td>The ROS Boss curates the process of choosing the distribution’s name, using input from sources such as the community and potential naming conflicts.</td>
</tr>
<tr>
<td><strong>Create distribution’s documentation page</strong></td>
<td>Every distribution has a documentation page that lists its vital statistics, such as planned release date, EOL date, and significant changes since the previous release.</td>
</tr>
<tr>
<td><strong>Set release timeline</strong></td>
<td>The final weeks leading up to release day (usually, World Turtle Day) are hectic and full of deadlines, such as when to freeze the default RMW implementation. These deadlines must be planned well in advance.</td>
</tr>
<tr>
<td><strong>Produce roadmap</strong></td>
<td>While every contributor to ROS has their own planned features for each distribution, we try to maintain an overall roadmap of the new features and significant changes we expect to see in the distribution. The ROS Boss and the leader of the ROS 2 development team at Open Robotics work together with the ROS 2 TSC and other interested parties to produce a roadmap that is achievable in the time available and meets the needs of the ROS community.</td>
</tr>
<tr>
<td><strong>Announce roadmap</strong></td>
<td>The list of planned features and significant changes is made public, via a GitHub issue that will track the progress on developing each item in the roadmap. Of course, this does not mean that the roadmap is fixed at this point, as development plans can change and we always (and frequently do) welcome new contributions even if they are not on the planned roadmap.</td>
</tr>
<tr>
<td><strong>Set target platforms and major dependencies</strong></td>
<td>The target platforms, in terms of operating system, distribution and version, must be set far enough in advance for development work on the infrastructure (such as support in the build farm) to proceed. Similarly, the versions of each major dependency (which Python version, which compiler(s), which version of Eigen, etc.) must also be fixed. This is done via an update to REP-2000.</td>
</tr>
<tr>
<td><strong>Add platform support to the build farm</strong></td>
<td>The build farm is a critical part of the infrastructure supporting a ROS 2 distribution. It provides continuous integration facilities that help us maintain quality, and it builds the binary packages the community relies on to avoid building ROS 2 and packages from source. If the target platforms differ from the previous ROS 2 distribution, then the necessary support must be added to the build farm.</td>
</tr>
<tr>
<td><strong>Commission logo and related artwork</strong></td>
<td>A well-loved part of every ROS 2 distribution (and ROS distribution!) is the logo. The logo is commissioned from a professional artist based on the chosen distribution name. Based on the logo, other artwork such as the turtlesim icon are also produced.</td>
</tr>
<tr>
<td><strong>Create mailing list for the distribution</strong></td>
<td>Vital for making critical announcements, a mailing list must be set up to contact people interested in knowing something about the distribution, such as that their package is failing to build into a binary on the build farm.</td>
</tr>
<tr>
<td><strong>Create test cases</strong></td>
<td>As the development process enters the final few months, testing begins in earnest. The integration test cases that will be used during the final stages of development must be produced and provided to the release team who will be responsible for executing them.</td>
</tr>
<tr>
<td><strong>Announce upcoming RMW freeze</strong></td>
<td>The RMW freeze is the point at which the default RMW implementation for the new distribution is feature-frozen. This gives developers a stable target to test their packages with, which is particularly important for the client library developers, who need to know what features of the RMW layer will be available for use by client libraries.</td>
</tr>
<tr>
<td><strong>Upgrade dependency packages</strong></td>
<td>Packages depended on by ROS but not ROS software and not available in the platform package manager (such as aptitude for Ubuntu), the so-called “vendor packages”, must be updated to the versions specified in REP-2000 (or an appropriate version, for those not listed in REP-2000). This is particularly important on Windows.</td>
</tr>
<tr>
<td><strong>Create a detailed release plan</strong></td>
<td>Planning for the final two months of the development process is performed. This produces a detailed test plan, timelines of when certain packages must be available, and so on. It enables the finding of dependencies between steps in the release process and finding people to perform each of those steps.</td>
</tr>
<tr>
<td><strong>Freeze RMW</strong></td>
<td>The RMW implementation is now feature-frozen. In theory, it can now be ex-</td>
</tr>
</tbody>
</table>

---

## Final release preparations

- **Announce beta**
- **Add distribution documentation**
- **Tarballs**
- **Begin building interim testing**
- **Add distribution to CI**
- **Announce upcoming beta**
- **Announce upcoming branch**
- **Freeze RMW**
- **Create a detailed release plan**
- **Upgrade dependency packages**
- **Create test cases**
- **Announce roadmap**
- **Set target platforms and major dependencies**
- **Add platform support to the build farm**
- **Commission logo and related artwork**
- **Create mailing list for the distribution**
- **Find the ROS Boss**
- **Run process to choose the distribution name**
- **Create distribution’s documentation page**
- **Set release timeline**
- **Produce roadmap**
- **Announce roadmap**
- **Set target platforms and major dependencies**
- **Add platform support to the build farm**
- **Commission logo and related artwork**
- **Create mailing list for the distribution**
- **Create test cases**
- **Announce upcoming RMW freeze**
- **Upgrade dependency packages**
- **Create a detailed release plan**
- **Freeze RMW**
<table>
<thead>
<tr>
<th>Distro</th>
<th>Release date</th>
<th>Logo</th>
<th>EOL date</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Iron Irwini</strong></td>
<td>May 23rd, 2023</td>
<td><img src="image" alt="Iron Irwini Logo" /></td>
<td>November 2024</td>
</tr>
<tr>
<td><strong>Humble Hawksbill</strong></td>
<td>May 23rd, 2022</td>
<td><img src="image" alt="Humble Hawksbill Logo" /></td>
<td>May 2027</td>
</tr>
<tr>
<td><strong>Galactic Geochelone</strong></td>
<td>May 23rd, 2021</td>
<td><img src="image" alt="Galactic Geochelone Logo" /></td>
<td>December 9th, 2022</td>
</tr>
<tr>
<td><strong>Foxy Fitzroy</strong></td>
<td>June 5th, 2020</td>
<td><img src="image" alt="Foxy Fitzroy Logo" /></td>
<td>June 20th, 2023</td>
</tr>
<tr>
<td><strong>Eloquent Elusor</strong></td>
<td>November 22nd, 2019</td>
<td><img src="image" alt="Eloquent Elusor Logo" /></td>
<td>November 2020</td>
</tr>
<tr>
<td><strong>Dashing Diademata</strong></td>
<td>May 31st, 2019</td>
<td><img src="image" alt="Dashing Diademata Logo" /></td>
<td>May 2021</td>
</tr>
<tr>
<td><strong>Crystal Clemmys</strong></td>
<td>December 14th, 2018</td>
<td><img src="image" alt="Crystal Clemmys Logo" /></td>
<td>December 2019</td>
</tr>
</tbody>
</table>
**Future Distributions**

For details on upcoming features see the *roadmap*.

There is a new ROS 2 distribution released yearly on May 23rd (World Turtle Day).

<table>
<thead>
<tr>
<th>Distro</th>
<th>Release date</th>
<th>Logo</th>
<th>EOL date</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Jazzy Jalisco</em></td>
<td>May 2024</td>
<td>TBD</td>
<td>May 2029</td>
</tr>
</tbody>
</table>

**Rolling Distribution**

*ROS 2 Rolling Ridley* is the rolling development distribution of ROS 2. It is described in REP 2002 and was first introduced in June 2020.

The Rolling distribution of ROS 2 serves two purposes:

1. it is a staging area for future stable distributions of ROS 2, and
2. it is a collection of the most recent development releases.

As the name implies, Rolling is continuously updated and **can have in-place updates that include breaking changes**. We recommend that most people use the most recent stable distribution instead (see *List of Distributions*).

Packages released into the Rolling distribution will be automatically released into future stable distributions of ROS 2. *Releasing a ROS 2 package* into the Rolling distribution follows the same procedures as all other ROS 2 distributions.

**4.8.3 Tutorials**

The tutorials are a collection of step-by-step instructions meant to steadily build skills in ROS 2.

The best way to approach the tutorials is to walk through them for the first time in order, as they build off of each other and are not meant to be comprehensive documentation.

For quick solutions to more specific questions, see the *How-to Guides*.

**Beginner: CLI tools**

**Configuring environment**

**Goal:** This tutorial will show you how to prepare your ROS 2 environment.

**Tutorial level:** Beginner

**Time:** 5 minutes

**Contents**

- **Background**
- **Prerequisites**
- **Tasks**
  - 1 Source the setup files
  - 2 Add sourcing to your shell startup script
Background

ROS 2 relies on the notion of combining workspaces using the shell environment. “Workspace” is a ROS term for the location on your system where you’re developing with ROS 2. The core ROS 2 workspace is called the underlay. Subsequent local workspaces are called overlays. When developing with ROS 2, you will typically have several workspaces active concurrently.

Combining workspaces makes developing against different versions of ROS 2, or against different sets of packages, easier. It also allows the installation of several ROS 2 distributions (or “distros”, e.g. Dashing and Eloquent) on the same computer and switching between them.

This is accomplished by sourcing setup files every time you open a new shell, or by adding the source command to your shell startup script once. Without sourcing the setup files, you won’t be able to access ROS 2 commands, or find or use ROS 2 packages. In other words, you won’t be able to use ROS 2.

Prerequisites

Before starting these tutorials, install ROS 2 by following the instructions on the ROS 2 Installation page.

The commands used in this tutorial assume you followed the binary packages installation guide for your operating system (Debian packages for Linux). You can still follow along if you built from source, but the path to your setup files will likely be different. You also won’t be able to use the sudo apt install ros-<distro>-<package> command (used frequently in the beginner level tutorials) if you install from source.

If you are using Linux or macOS, but are not already familiar with the shell, this tutorial will help.

Tasks

1 Source the setup files

You will need to run this command on every new shell you open to have access to the ROS 2 commands, like so:

Linux

```
# Replace ".bash" with your shell if you're not using bash
# Possible values are: setup.bash, setup.sh, setup.zsh
source /opt/ros/iron/setup.bash
```

macOS

```
. ~/ros2_install/ros2-osx/setup.bash
```

Windows

```
call C:\dev\ros2\local_setup.bat
```
Note: The exact command depends on where you installed ROS 2. If you’re having problems, ensure the file path leads to your installation.

2 Add sourcing to your shell startup script

If you don’t want to have to source the setup file every time you open a new shell (skipping task 1), then you can add the command to your shell startup script:

Linux

```bash
echo "source /opt/ros/iron/setup.bash" >> ~/.bashrc
```

To undo this, locate your system’s shell startup script and remove the appended source command.

macOS

```bash
echo "source ~/ros2_install/ros2-osx/setup.bash" >> ~/.bash_profile
```

To undo this, locate your system’s shell startup script and remove the appended source command.

Windows


```powershell
C:\dev\ros2_iron\local_setup.ps1
```

PowerShell will request permission to run this script everytime a new shell is opened. To avoid that issue you can run:

```powershell
Unblock-File C:\dev\ros2_iron\local_setup.ps1
```

To undo this, remove the new ‘Microsoft.PowerShell_profile.ps1’ file.

3 Check environment variables

Sourcing ROS 2 setup files will set several environment variables necessary for operating ROS 2. If you ever have problems finding or using your ROS 2 packages, make sure that your environment is properly set up using the following command:

Linux

```bash
printenv | grep -i ROS
```

macOS

```bash
printenv | grep -i ROS
```

Windows

```powershell
set | findstr -i ROS
```

Check that variables like ROS_DISTRO and ROS_VERSION are set.
If the environment variables are not set correctly, return to the ROS 2 package installation section of the installation guide you followed. If you need more specific help (because environment setup files can come from different places), you can get answers from the community.

### 3.1 The ROS_DOMAIN_ID variable

See the *domain ID* article for details on ROS domain IDs. Once you have determined a unique integer for your group of ROS 2 nodes, you can set the environment variable with the following command:

**Linux**

```bash
export ROS_DOMAIN_ID=<your_domain_id>
```

To maintain this setting between shell sessions, you can add the command to your shell startup script:

```bash
echo "export ROS_DOMAIN_ID=<your_domain_id>" >> ~/.bashrc
```

**macOS**

```bash
export ROS_DOMAIN_ID=<your_domain_id>
```

To maintain this setting between shell sessions, you can add the command to your shell startup script:

```bash
echo "export ROS_DOMAIN_ID=<your_domain_id>" >> ~/.bash_profile
```

**Windows**

```bash
set ROS_DOMAIN_ID=<your_domain_id>
```

If you want to make this permanent between shell sessions, also run:

```bash
setx ROS_DOMAIN_ID <your_domain_id>
```

### 3.2 The ROS_AUTOMATIC_DISCOVERY_RANGE variable

By default, ROS 2 communication is not limited to localhost. ROS_AUTOMATIC_DISCOVERY_RANGE environment variable allows you to limit ROS 2 discovery range. Using ROS_AUTOMATIC_DISCOVERY_RANGE is helpful in certain settings, such as classrooms, where multiple robots may publish to the same topic causing strange behaviors. See *Improved Dynamic Discovery* for more details.
Summary

The ROS 2 development environment needs to be correctly configured before use. This can be done in two ways: either sourcing the setup files in every new shell you open, or adding the source command to your startup script.

If you ever face any problems locating or using packages with ROS 2, the first thing you should do is check your environment variables and ensure they are set to the version and distro you intended.

Next steps

Now that you have a working ROS 2 installation and you know how to source its setup files, you can start learning the ins and outs of ROS 2 with the turtlesim tool.

Using turtlesim, ros2, and rqt

Goal: Install and use the turtlesim package and rqt tools to prepare for upcoming tutorials.

Tutorial level: Beginner

Time: 15 minutes

Contents

• Background
• Prerequisites
• Tasks
  – 1 Install turtlesim
  – 2 Start turtlesim
  – 3 Use turtlesim
  – 4 Install rqt
  – 5 Use rqt
  – 6 Remapping
  – 7 Close turtlesim
• Summary
• Next steps
• Related content
Background

Turtlesim is a lightweight simulator for learning ROS 2. It illustrates what ROS 2 does at the most basic level to give you an idea of what you will do with a real robot or a robot simulation later on.

The ros2 tool is how the user manages, introspects, and interacts with a ROS system. It supports multiple commands that target different aspects of the system and its operation. One might use it to start a node, set a parameter, listen to a topic, and many more. The ros2 tool is part of the core ROS 2 installation.

rqt is a graphical user interface (GUI) tool for ROS 2. Everything done in rqt can be done on the command line, but rqt provides a more user-friendly way to manipulate ROS 2 elements.

This tutorial touches upon core ROS 2 concepts, like nodes, topics, and services. All of these concepts will be elaborated on in later tutorials; for now, you will simply set up the tools and get a feel for them.

Prerequisites

The previous tutorial, Configuring environment, will show you how to set up your environment.

Tasks

1 Install turtlesim

As always, start by sourcing your setup files in a new terminal, as described in the previous tutorial.

Install the turtlesim package for your ROS 2 distro:

Linux

```
sudo apt update
sudo apt install ros-iron-turtlesim
```

macOS

As long as the archive you installed ROS 2 from contains the ros_tutorials repository, you should already have turtlesim installed.

Windows

As long as the archive you installed ROS 2 from contains the ros_tutorials repository, you should already have turtlesim installed.

Check that the package is installed:

```
ros2 pkg executables turtlesim
```

The above command should return a list of turtlesim’s executables:

```
turtlesim draw_square
turtlesim mimic
turtlesim turtle_teleop_key
turtlesim turtlesim_node
```
2 Start turtlesim

To start turtlesim, enter the following command in your terminal:

```
ros2 run turtlesim turtlesim_node
```

The simulator window should appear, with a random turtle in the center.

In the terminal, under the command, you will see messages from the node:

```
[INFO] [turtlesim]: Starting turtlesim with node name /turtlesim
[INFO] [turtlesim]: Spawning turtle [turtle1] at x=[5.544445], y=[5.544445], theta=[0.000000]
```

There you can see the default turtle's name and the coordinates where it spawns.
3 Use turtlesim

Open a new terminal and source ROS 2 again.

Now you will run a new node to control the turtle in the first node:

```
ros2 run turtlesim turtle_teleop_key
```

At this point you should have three windows open: a terminal running `turtlesim_node`, a terminal running `turtle_teleop_key` and the turtlesim window. Arrange these windows so that you can see the turtlesim window, but also have the terminal running `turtle_teleop_key` active so that you can control the turtle in turtlesim.

Use the arrow keys on your keyboard to control the turtle. It will move around the screen, using its attached “pen” to draw the path it followed so far.

**Note:** Pressing an arrow key will only cause the turtle to move a short distance and then stop. This is because, realistically, you wouldn’t want a robot to continue carrying on an instruction if, for example, the operator lost the connection to the robot.

You can see the nodes, and their associated topics, services, and actions, using the `list` subcommands of the respective commands:

```
ros2 node list
tros2 topic list
tros2 service list
tros2 action list
```

You will learn more about these concepts in the coming tutorials. Since the goal of this tutorial is only to get a general overview of turtlesim, you will use rqt to call some of the turtlesim services and interact with `turtlesim_node`.

4 Install rqt

Open a new terminal to install `rqt` and its plugins:

Ubuntu

```
sudo apt update
sudo apt install nros-iron-rqt
```

RHEL

```
sudo dnf install ros-iron-rqt
```

macOS

The standard archive for installing ROS 2 on macOS contains `rqt` and its plugins, so you should already have `rqt` installed.

Windows

The standard archive for installing ROS 2 on Windows contains `rqt` and its plugins, so you should already have `rqt` installed.

To run rqt:
5 Use rqt

When running rqt for the first time, the window will be blank. No worries; just select **Plugins > Services > Service Caller** from the menu bar at the top.

**Note:** It may take some time for rqt to locate all the plugins. If you click on **Plugins** but don’t see **Services** or any other options, you should close rqt and enter the command `rqt --force-discover` in your terminal.

Use the refresh button to the left of the **Service** dropdown list to ensure all the services of your turtlesim node are available.

Click on the **Service** dropdown list to see turtlesim’s services, and select the `/spawn` service.
5.1 Try the spawn service

Let's use rqt to call the /spawn service. You can guess from its name that /spawn will create another turtle in the turtlesim window.

Give the new turtle a unique name, like turtle2, by double-clicking between the empty single quotes in the Expression column. You can see that this expression corresponds to the value of name and is of type string.

Next enter some valid coordinates at which to spawn the new turtle, like \( x = 1.0 \) and \( y = 1.0 \).

![Image showing the rqt window with /spawn service call and expected values for x, y, and name]

Note: If you try to spawn a new turtle with the same name as an existing turtle, like the default turtle1, you will get an error message in the terminal running turtlesim_node:

```
[ERROR] [turtlesim]: A turtle named [turtle1] already exists
```

To spawn turtle2, you then need to call the service by clicking the Call button on the upper right side of the rqt window.

If the service call was successful, you should see a new turtle (again with a random design) spawn at the coordinates you input for \( x \) and \( y \).
If you refresh the service list in rqt, you will also see that now there are services related to the new turtle, /turtle2/..., in addition to /turtle1/....

5.2 Try the set_pen service

Now let's give turtle1 a unique pen using the /set_pen service:

![Screenshot of rqt showing the set_pen service call](image)

The values for \( r \), \( g \) and \( b \), which are between 0 and 255, set the color of the pen \( \text{turtle1} \) draws with, and \( \text{width} \) sets the thickness of the line.

To have \( \text{turtle1} \) draw with a distinct red line, change the value of \( r \) to 255, and the value of \( \text{width} \) to 5. Don’t forget to call the service after updating the values.

If you return to the terminal where \( \text{turtle_teleop_key} \) is running and press the arrow keys, you will see \( \text{turtle1} \)'s pen has changed.
You’ve probably also noticed that there’s no way to move turtle2. That’s because there is no teleop node for turtle2.

### 6 Remapping

You need a second teleop node in order to control turtle2. However, if you try to run the same command as before, you will notice that this one also controls turtle1. The way to change this behavior is by remapping the cmd_vel topic.

In a new terminal, source ROS 2, and run:

```
ros2 run turtlesim turtle_teleop_key --ros-args --remap turtle1/cmd_vel:=turtle2/cmd_vel
```

Now, you can move turtle2 when this terminal is active, and turtle1 when the other terminal running turtle_teleop_key is active.
7 Close turtlesim

To stop the simulation, you can enter Ctrl + C in the turtlesim_node terminal, and q in the turtle_teleop_key terminals.
Summary

Using turtlesim and rqt is a great way to learn the core concepts of ROS 2.

Next steps

Now that you have turtlesim and rqt up and running, and an idea of how they work, let’s dive into the first core ROS 2 concept with the next tutorial, *Understanding nodes*.

Related content

The turtlesim package can be found in the ros_tutorials repo.
This community contributed video demonstrates many of the items covered in this tutorial.

Understanding nodes

Goal: Learn about the function of nodes in ROS 2, and the tools to interact with them.

Tutorial level: Beginner

Time: 10 minutes

Contents

- Background
  - 1 The ROS 2 graph
  - 2 Nodes in ROS 2
- Prerequisites
- Tasks
  - 1 ros2 run
  - 2 ros2 node list
  - 3 ros2 node info
- Summary
- Next steps
- Related content
Background

1 The ROS 2 graph

Over the next few tutorials, you will learn about a series of core ROS 2 concepts that make up what is referred to as the “ROS (2) graph”.

The ROS graph is a network of ROS 2 elements processing data together at the same time. It encompasses all executables and the connections between them if you were to map them all out and visualize them.

2 Nodes in ROS 2

Each node in ROS should be responsible for a single, modular purpose, e.g. controlling the wheel motors or publishing the sensor data from a laser range-finder. Each node can send and receive data from other nodes via topics, services, actions, or parameters.

A full robotic system is comprised of many nodes working in concert. In ROS 2, a single executable (C++ program, Python program, etc.) can contain one or more nodes.

Prerequisites

The previous tutorial shows you how to install the turtlesim package used here.

As always, don’t forget to source ROS 2 in every new terminal you open.

Tasks

1 ros2 run

The command ros2 run launches an executable from a package.

```
ros2 run <package_name> <executable_name>
```

To run turtlesim, open a new terminal, and enter the following command:

```
ros2 run turtlesim turtlesim_node
```

The turtlesim window will open, as you saw in the previous tutorial.

Here, the package name is turtlesim and the executable name is turtlesim_node.

We still don't know the node name, however. You can find node names by using ros2 node list
2 ros2 node list

`ros2 node list` will show you the names of all running nodes. This is especially useful when you want to interact with a node, or when you have a system running many nodes and need to keep track of them.

Open a new terminal while turtlesim is still running in the other one, and enter the following command:

```
ros2 node list
```

The terminal will return the node name:

```
/turtlesim
```

Open another new terminal and start the teleop node with the command:

```
ros2 run turtlesim turtle_teleop_key
```

Here, we are referring to the `turtlesim` package again, but this time we target the executable named `turtle_teleop_key`.

Return to the terminal where you ran `ros2 node list` and run it again. You will now see the names of two active nodes:

```
/turtlesim
/teleop_turtle
```

2.1 Remapping

Remapping allows you to reassign default node properties, like node name, topic names, service names, etc., to custom values. In the last tutorial, you used remapping on `turtle_teleop_key` to change the cmd_vel topic and target `turtle2`.

Now, let’s reassign the name of our `/turtlesim` node. In a new terminal, run the following command:

```
ros2 run turtlesim turtlesim_node --ros-args --remap __node:=my_turtle
```

Since you’re calling `ros2 run` on turtlesim again, another turtlesim window will open. However, now if you return to the terminal where you ran `ros2 node list`, and run it again, you will see three node names:

```
/my_turtle
/turtlesim
/teleop_turtle
```

3 ros2 node info

Now that you know the names of your nodes, you can access more information about them with:

```
ros2 node info <node_name>
```

To examine your latest node, `my_turtle`, run the following command:

```
ros2 node info /my_turtle
```
ros2 node info returns a list of subscribers, publishers, services, and actions. i.e. the ROS graph connections that interact with that node. The output should look like this:

```
/my_turtle
Subscribers:
  /parameter_events: rcl_interfaces/msg/ParameterEvent
  /turtle1/cmd_vel: geometry_msgs/msg/Twist
Publishers:
  /parameter_events: rcl_interfaces/msg/ParameterEvent
  /rosout: rcl_interfaces/msg/Log
  /turtle1/color_sensor: turtlesim/msg/Color
  /turtle1/pose: turtlesim/msg/Pose
Service Servers:
  /clear: std_srvs/srv/Empty
  /kill: turtlesim_srv/Kill
  /my_turtle/describe_parameters: rcl_interfaces/srv/DescribeParameters
  /my_turtle/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
  /my_turtle/get_parameters: rcl_interfaces/srv/GetParameters
  /my_turtle/list_parameters: rcl_interfaces/srv/ListParameters
  /my_turtle/set_parameters: rcl_interfaces/srv/SetParameters
  /my_turtle/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
  /reset: std_srvs/srv/Empty
  /spawn: turtlesim/srv/Spawn
  /turtle1/set_pen: turtlesim/srv/SetPen
  /turtle1/teleport_absolute: turtlesim/srv/TeleportAbsolute
  /turtle1/teleport_relative: turtlesim/srv/TeleportRelative
Service Clients:
Action Servers:
  /turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
Action Clients:
```

Now try running the same command on the /teleop_turtle node, and see how its connections differ from my_turtle.

You will learn more about the ROS graph connection concepts including the message types in the upcoming tutorials.

**Summary**

A node is a fundamental ROS 2 element that serves a single, modular purpose in a robotics system.

In this tutorial, you utilized nodes created in the turtlesim package by running the executables turtlesim_node and turtle_teleop_key.

You learned how to use ros2 node list to discover active node names and ros2 node info to introspect a single node. These tools are vital to understanding the flow of data in a complex, real-world robot system.
Next steps

Now that you understand nodes in ROS 2, you can move on to the topics tutorial. Topics are one of the communication types that connects nodes.

Related content

The Concepts page adds some more detail to the concept of nodes.

Understanding topics

Goal: Use rqt_graph and command line tools to introspect ROS 2 topics.

Tutorial level: Beginner

Time: 20 minutes

Contents

- Background
- Prerequisites
- Tasks
  - 1 Setup
  - 2 rqt_graph
  - 3 ros2 topic list
  - 4 ros2 topic echo
  - 5 ros2 topic info
  - 6 ros2 interface show
  - 7 ros2 topic pub
  - 8 ros2 topic hz
  - 9 Clean up
- Summary
- Next steps

Background

ROS 2 breaks complex systems down into many modular nodes. Topics are a vital element of the ROS graph that act as a bus for nodes to exchange messages.

A node may publish data to any number of topics and simultaneously have subscriptions to any number of topics.

Topics are one of the main ways in which data is moved between nodes and therefore between different parts of the system.
Prerequisites

The previous tutorial provides some useful background information on nodes that is built upon here. As always, don’t forget to source ROS 2 in every new terminal you open.

Tasks

1 Setup

By now you should be comfortable starting up turtlesim.

Open a new terminal and run:

```
ros2 run turtlesim turtlesim_node
```

Open another terminal and run:

```
ros2 run turtlesim turtle_teleop_key
```

Recall from the previous tutorial that the names of these nodes are /turtlesim and /teleop_turtle by default.

2 rqt_graph

Throughout this tutorial, we will use rqt_graph to visualize the changing nodes and topics, as well as the connections between them.

The turtlesim tutorial tells you how to install rqt and all its plugins, including rqt_graph.

To run rqt_graph, open a new terminal and enter the command:

```
rqt_graph
```

You can also open rqt_graph by opening rqt and selecting Plugins > Introspection > Node Graph.
You should see the above nodes and topic, as well as two actions around the periphery of the graph (let’s ignore those for now). If you hover your mouse over the topic in the center, you’ll see the color highlighting like in the image above.

The graph is depicting how the /turtlesim node and the /teleop_turtle node are communicating with each other over a topic. The /teleop_turtle node is publishing data (the keystrokes you enter to move the turtle around) to the /turtle1/cmd_vel topic, and the /turtlesim node is subscribed to that topic to receive the data.

The highlighting feature of rqt_graph is very helpful when examining more complex systems with many nodes and topics connected in many different ways.

rqt_graph is a graphical introspection tool. Now we’ll look at some command line tools for introspecting topics.

3 ros2 topic list

Running the ros2 topic list command in a new terminal will return a list of all the topics currently active in the system:

```
/parameter_events
/rosout
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

ros2 topic list -t will return the same list of topics, this time with the topic type appended in brackets:

```
/parameter_events [rcl_interfaces/msg/ParameterEvent]
/rosout [rcl_interfaces/msg/Log]
/turtle1/cmd_vel [geometry_msgs/msg/Twist]
```
These attributes, particularly the type, are how nodes know they’re talking about the same information as it moves over topics.

If you’re wondering where all these topics are in rqt_graph, you can uncheck all the boxes under **Hide**:  

![Node Graph](image)

For now, though, leave those options checked to avoid confusion.

### 4 ros2 topic echo

To see the data being published on a topic, use:

```
ros2 topic echo <topic_name>
```

Since we know that `/teleop_turtle` publishes data to `/turtlesim` over the `/turtle1/cmd_vel` topic, let’s use `ros2 topic echo` to introspect that topic:

```
ros2 topic echo /turtle1/cmd_vel
```

At first, this command won’t return any data. That’s because it’s waiting for `/teleop_turtle` to publish something.
Return to the terminal where `turtle_teleop_key` is running and use the arrows to move the turtle around. Watch the terminal where your `echo` is running at the same time, and you'll see position data being published for every movement you make:

```
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
```

Now return to rqt_graph and uncheck the Debug box.

Now return to rqt_graph and uncheck the Debug box.

```
/robot_gui_node_26706
/roslaunch_node_26646
/turtle1/rotate_absolute/_action/status
/turtle1/rotate_absolute/_action/feedback
/teleop_turtle
```

`/ros2cli_26646` is the node created by the `echo` command we just ran (the number might be different). Now you can see that the publisher is publishing data over the `cmd_vel` topic, and two subscribers are subscribed to it.

### 5 ros2 topic info

Topics don't have to only be one-to-one communication; they can be one-to-many, many-to-one, or many-to-many. Another way to look at this is running:

```
ros2 topic info /turtle1/cmd_vel
```

Which will return:
Type: geometry_msgs/msg/Twist
Publisher count: 1
Subscription count: 2

6 ros2 interface show

Nodes send data over topics using messages. Publishers and subscribers must send and receive the same type of message to communicate.

The topic types we saw earlier after running `ros2 topic list -t` let us know what message type is used on each topic. Recall that the `cmd_vel` topic has the type:

```
geometry_msgs/msg/Twist
```

This means that in the package `geometry_msgs` there is a `msg` called `Twist`.

Now we can run `ros2 interface show <msg type>` on this type to learn its details. Specifically, what structure of data the message expects.

```
ros2 interface show geometry_msgs/msg/Twist
```

For the message type from above it yields:

```
# This expresses velocity in free space broken into its linear and angular parts.

Vector3 linear
  float64 x
  float64 y
  float64 z
Vector3 angular
  float64 x
  float64 y
  float64 z
```

This tells you that the `/turtlesim` node is expecting a message with two vectors, `linear` and `angular`, of three elements each. If you recall the data we saw `/teleop_turtle` passing to `/turtlesim` with the `echo` command, it’s in the same structure:

```
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
```
7 ros2 topic pub

Now that you have the message structure, you can publish data onto a topic directly from the command line using:

```
ros2 topic pub <topic_name> <msg_type> 'ARGS'
```

The 'ARGS' argument is the actual data you’ll pass to the topic, in the structure you just discovered in the previous section.

It's important to note that this argument needs to be input in YAML syntax. Input the full command like so:

```
ros2 topic pub --once /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}}"
```

--once is an optional argument meaning “publish one message then exit”.

You will see the following output in the terminal:

```
publisher: beginning loop
publishing #1: geometry_msgs.msg.Twist(linear=geometry_msgs.msg.Vector3(x=2.0, y=0.0, z=0.0), angular=geometry_msgs.msg.Vector3(x=0.0, y=0.0, z=1.8))
```

And you will see your turtle move like so:
The turtle (and commonly the real robots which it is meant to emulate) require a steady stream of commands to operate continuously. So, to get the turtle to keep moving, you can run:

```
ros2 topic pub --rate 1 /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0., z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}}"
```

The difference here is the removal of the `--once` option and the addition of the `--rate 1` option, which tells `ros2 topic pub` to publish the command in a steady stream at 1 Hz.
You can refresh rqt_graph to see what’s happening graphically. You will see that the `ros2 topic pub ... node (_ros2cli_30358)` is publishing over the `/turtle1/cmd_vel` topic, which is being received by both the `ros2 topic echo ... node (_ros2cli_26646)` and the `/turtlesim` node now.
Finally, you can run `echo` on the `pose` topic and recheck rqt_graph:

```
ros2 topic echo /turtle1/pose
```
You can see that the /turtlesim node is also publishing to the pose topic, which the new echo node has subscribed to.

8 ros2 topic hz

For one last introspection on this process, you can view the rate at which data is published using:

```
ros2 topic hz /turtle1/pose
```

It will return data on the rate at which the /turtlesim node is publishing data to the pose topic.

```
average rate: 59.354
min: 0.005s max: 0.027s std dev: 0.00284s window: 58
```

Recall that you set the rate of turtle1/cmd_vel to publish at a steady 1 Hz using `ros2 topic pub --rate 1`. If you run the above command with `turtle1/cmd_vel` instead of `turtle1/pose`, you will see an average reflecting that rate.
9 Clean up

At this point you’ll have a lot of nodes running. Don’t forget to stop them by entering Ctrl+C in each terminal.

Summary

Nodes publish information over topics, which allows any number of other nodes to subscribe to and access that information. In this tutorial you examined the connections between several nodes over topics using rqt_graph and command line tools. You should now have a good idea of how data moves around a ROS 2 system.

Next steps

Next you’ll learn about another communication type in the ROS graph with the tutorial Understanding services.

Understanding services

Goal: Learn about services in ROS 2 using command line tools.

Tutorial level: Beginner

Time: 10 minutes
Background

Services are another method of communication for nodes in the ROS graph. Services are based on a call-and-response model versus the publisher-subscriber model of topics. While topics allow nodes to subscribe to data streams and get continual updates, services only provide data when they are specifically called by a client.

Prerequisites

Some concepts mentioned in this tutorial, like Nodes and Topics, were covered in previous tutorials in the series. You will need the turtlesim package.

As always, don’t forget to source ROS 2 in every new terminal you open.

Tasks

1 Setup

Start up the two turtlesim nodes, /turtlesim and /teleop_turtle.

Open a new terminal and run:

```bash
ros2 run turtlesim turtlesim_node
```

Open another terminal and run:

```bash
ros2 run turtlesim turtle_teleop_key
```

2 ros2 service list

Running the ros2 service list command in a new terminal will return a list of all the services currently active in the system:

```
/clear
/kill
/reset
/spawn
/teleop_turtle/describe_parameters
/teleop_turtle/get_parameter_types
/teleop_turtle/get_parameters
/teleop_turtle/list_parameters
/teleop_turtle/set_parameters
/teleop_turtle/set_parameters_atomically
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/describe_parameters
/turtlesim/get_parameter_types
/turtlesim/get_parameters
/turtlesim/list_parameters
```

(continues on next page)
You will see that both nodes have the same six services with parameters in their names. Nearly every node in ROS 2 has these infrastructure services that parameters are built off of. There will be more about parameters in the next tutorial. In this tutorial, the parameter services will be omitted from the discussion.

For now, let’s focus on the turtlesim-specific services, /clear, /kill, /reset, /spawn, /turtle1/set_pen, /turtle1/teleport_absolute, and /turtle1/teleport_relative. You may recall interacting with some of these services using rqt in the *Use turtlesim, ros2, and rqt* tutorial.

### 3 ros2 service type

Services have types that describe how the request and response data of a service is structured. Service types are defined similarly to topic types, except service types have two parts: one message for the request and another for the response.

To find out the type of a service, use the command:

```
ros2 service type <service_name>
```

Let’s take a look at turtlesim’s /clear service. In a new terminal, enter the command:

```
ros2 service type /clear
```

Which should return:

```
std_srvs/srv/Empty
```

The Empty type means the service call sends no data when making a request and receives no data when receiving a response.

### 3.1 ros2 service list -t

To see the types of all the active services at the same time, you can append the --show-types option, abbreviated as -t, to the list command:

```
ros2 service list -t
```

Which will return:

```
/clear [std_srvs/srv/Empty]
/kill [turtlesim/srv/Kill]
/reset [std_srvs/srv/Empty]
/spawn [turtlesim/srv/Spawn]
...
/turtle1/set_pen [turtlesim/srv/SetPen]
/turtle1/teleport_absolute [turtlesim/srv/TeleportAbsolute]
/turtle1/teleport_relative [turtlesim/srv/TeleportRelative]
...
```
4 ros2 service find

If you want to find all the services of a specific type, you can use the command:

```
ros2 service find <type_name>
```

For example, you can find all the Empty typed services like this:

```
ros2 service find std_srvs/srv/Empty
```

Which will return:

```
/clear
/reset
```

5 ros2 interface show

You can call services from the command line, but first you need to know the structure of the input arguments.

```
ros2 interface show <type_name>
```

Try this on the /clear service's type, Empty:

```
ros2 interface show std_srvs/srv/Empty
```

Which will return:

```
---
```

The --- separates the request structure (above) from the response structure (below). But, as you learned earlier, the Empty type doesn't send or receive any data. So, naturally, its structure is blank.

Let's introspect a service with a type that sends and receives data, like /spawn. From the results of ros2 service list -t, we know /spawn's type is turtlesim/srv/Spawn.

To see the request and response arguments of the /spawn service, run the command:

```
ros2 interface show turtlesim/srv/Spawn
```

Which will return:

```
float32 x
float32 y
float32 theta
string name # Optional. A unique name will be created and returned if this is empty
---
string name
```

The information above the --- line tells us the arguments needed to call /spawn. x, y and theta determine the 2D pose of the spawned turtle, and name is clearly optional.

The information below the line isn't something you need to know in this case, but it can help you understand the data type of the response you get from the call.
6 ros2 service call

Now that you know what a service type is, how to find a service's type, and how to find the structure of that type's arguments, you can call a service using:

```
ros2 service call <service_name> <service_type> <arguments>
```

The `<arguments>` part is optional. For example, you know that `Empty` typed services don't have any arguments:

```
ros2 service call /clear std_srvs/srv/Empty
```

This command will clear the turtlesim window of any lines your turtle has drawn.

![Turtlesim window before and after clearing](image)

Now let's spawn a new turtle by calling `/spawn` and setting arguments. Input `<arguments>` in a service call from the command-line need to be in YAML syntax.

Enter the command:

```
ros2 service call /spawn turtlesim/srv/Spawn "{x: 2, y: 2, theta: 0.2, name: ''}"
```

You will get this method-style view of what's happening, and then the service response:

```
requester: making request: turtlesim.srv.Spawn_Request(x=2.0, y=2.0, theta=0.2, name='')
response:
turtlesim.srv.Spawn_Response(name='turtle2')
```

Your turtlesim window will update with the newly spawned turtle right away:
Summary

Nodes can communicate using services in ROS 2. Unlike a topic - a one way communication pattern where a node publishes information that can be consumed by one or more subscribers - a service is a request/response pattern where a client makes a request to a node providing the service and the service processes the request and generates a response.

You generally don’t want to use a service for continuous calls; topics or even actions would be better suited.

In this tutorial you used command line tools to identify, introspect, and call services.
Next steps

In the next tutorial, *Understanding parameters*, you will learn about configuring node settings.

Related content

Check out this tutorial; it’s an excellent realistic application of ROS services using a Robotis robot arm.

Understanding parameters

**Goal:** Learn how to get, set, save and reload parameters in ROS 2.

**Tutorial level:** Beginner

**Time:** 5 minutes

### Background

A parameter is a configuration value of a node. You can think of parameters as node settings. A node can store parameters as integers, floats, booleans, strings, and lists. In ROS 2, each node maintains its own parameters. For more background on parameters, please see the concept document.
Prerequisites

This tutorial uses the turtlesim package.
As always, don’t forget to source ROS 2 in every new terminal you open.

Tasks

1 Setup

Start up the two turtlesim nodes, /turtlesim and /teleop_turtle.
Open a new terminal and run:

```bash
ros2 run turtlesim turtlesim_node
```

Open another terminal and run:

```bash
ros2 run turtlesim turtle_teleop_key
```

2 ros2 param list

To see the parameters belonging to your nodes, open a new terminal and enter the command:

```bash
ros2 param list
```

You will see the node namespaces, /teleop_turtle and /turtlesim, followed by each node’s parameters:

```
/teleop_turtle:
qos_overrides./parameter_events.publisher.depth
qos_overrides./parameter_events.publisher.durability
qos_overrides./parameter_events.publisher.history
qos_overrides./parameter_events.publisher.reliability
scale_angular
scale_linear
use_sim_time
/turtlesim:
background_b
background_g
background_r
qos_overrides./parameter_events.publisher.depth
qos_overrides./parameter_events.publisher.durability
qos_overrides./parameter_events.publisher.history
qos_overrides./parameter_events.publisher.reliability
use_sim_time
```

Every node has the parameter use_sim_time; it's not unique to turtlesim.

Based on their names, it looks like /turtlesim’s parameters determine the background color of the turtlesim window using RGB color values.

To determine a parameter’s type, you can use ros2 param get.
3 ros2 param get

To display the type and current value of a parameter, use the command:

```
ros2 param get <node_name> <parameter_name>
```

Let's find out the current value of /turtlesim's parameter `background_g`:

```
ros2 param get /turtlesim background_g
```

Which will return the value:

```
Integer value is: 86
```

Now you know `background_g` holds an integer value.

If you run the same command on `background_r` and `background_b`, you will get the values 69 and 255, respectively.

4 ros2 param set

To change a parameter's value at runtime, use the command:

```
ros2 param set <node_name> <parameter_name> <value>
```

Let's change /turtlesim's background color:

```
ros2 param set /turtlesim background_r 150
```

Your terminal should return the message:

```
Set parameter successful
```

And the background of your turtlesim window should change colors:
Setting parameters with the `set` command will only change them in your current session, not permanently. However, you can save your settings and reload them the next time you start a node.

### 5 `ros2 param dump`

You can view all of a node’s current parameter values by using the command:

```
ros2 param dump <node_name>
```

The command prints to the standard output (stdout) by default but you can also redirect the parameter values into a file to save them for later. To save your current configuration of `/turtlesim`’s parameters into the file `turtlesim.yaml`, enter the command:

```
ros2 param dump /turtlesim > turtlesim.yaml
```

You will find a new file in the current working directory your shell is running in. If you open this file, you’ll see the following content:
Dumping parameters comes in handy if you want to reload the node with the same parameters in the future.

### 6 ros2 param load

You can load parameters from a file to a currently running node using the command:

```bash
ros2 param load <node_name> <parameter_file>
```

To load the `turtlesim.yaml` file generated with `ros2 param dump` into `/turtlesim` node’s parameters, enter the command:

```bash
ros2 param load /turtlesim turtlesim.yaml
```

Your terminal will return the message:

```
Set parameter background_b successful
Set parameter background_g successful
Set parameter background_r successful
Set parameter qos_overrides./parameter_events.publisher.depth failed: parameter 'qos_overrides./parameter_events.publisher.depth' cannot be set because it is read-only
Set parameter qos_overrides./parameter_events.publisher.durability failed: parameter 'qos_overrides./parameter_events.publisher.durability' cannot be set because it is read-only
Set parameter qos_overrides./parameter_events.publisher.history failed: parameter 'qos_overrides./parameter_events.publisher.history' cannot be set because it is read-only
Set parameter qos_overrides./parameter_events.publisher.reliability failed: parameter 'qos_overrides./parameter_events.publisher.reliability' cannot be set because it is read-only
Set parameter use_sim_time successful
```

**Note:** Read-only parameters can only be modified at startup and not afterwards, that is why there are some warnings for the “qos_overrides” parameters.
7 Load parameter file on node startup

To start the same node using your saved parameter values, use:

```
ros2 run <package_name> <executable_name> --ros-args --params-file <file_name>
```

This is the same command you always use to start turtlesim, with the added flags `--ros-args` and `--params-file`, followed by the file you want to load.

Stop your running turtlesim node, and try reloading it with your saved parameters, using:

```
ros2 run turtlesim turtlesim_node --ros-args --params-file turtlesim.yaml
```

The turtlesim window should appear as usual, but with the purple background you set earlier.

**Note:** When a parameter file is used at node startup, all parameters, including the read-only ones, will be updated.

**Summary**

Nodes have parameters to define their default configuration values. You can get and set parameter values from the command line. You can also save the parameter settings to a file to reload them in a future session.

**Next steps**

Jumping back to ROS 2 communication methods, in the next tutorial you’ll learn about *actions*.

**Understanding actions**

**Goal:** Introspect actions in ROS 2.

**Tutorial level:** Beginner

**Time:** 15 minutes

**Contents**

- Background
- Prerequisites
- Tasks
  - 1 Setup
  - 2 Use actions
  - 3 ros2 node info
  - 4 ros2 action list
  - 5 ros2 action info
  - 6 ros2 interface show
Background

Actions are one of the communication types in ROS 2 and are intended for long running tasks. They consist of three parts: a goal, feedback, and a result.

Actions are built on topics and services. Their functionality is similar to services, except actions can be canceled. They also provide steady feedback, as opposed to services which return a single response.

Actions use a client-server model, similar to the publisher-subscriber model (described in the topics tutorial). An “action client” node sends a goal to an “action server” node that acknowledges the goal and returns a stream of feedback and a result.

Prerequisites

This tutorial builds off concepts, like nodes and topics, covered in previous tutorials.

This tutorial uses the turtlesim package.

As always, don’t forget to source ROS 2 in every new terminal you open.

Tasks

1 Setup

Start up the two turtlesim nodes, /turtlesim and /teleop_turtle.

Open a new terminal and run:


ros2 run turtlesim turtlesim_node

Open another terminal and run:


ros2 run turtlesim turtle_teleop_key

2 Use actions

When you launch the /teleop_turtle node, you will see the following message in your terminal:

Use arrow keys to move the turtle.
Use G|B|V|C|D|E|R|T keys to rotate to absolute orientations. ‘F’ to cancel a rotation.

Let’s focus on the second line, which corresponds to an action. (The first instruction corresponds to the “cmd_vel” topic, discussed previously in the topics tutorial.)
Notice that the letter keys G|B|V|C|D|E|R|T form a “box” around the F key on a US QWERTY keyboard (if you are not using a QWERTY keyboard, see this link to follow along). Each key’s position around F corresponds to that orientation in turtlesim. For example, the E will rotate the turtle’s orientation to the upper left corner.

Pay attention to the terminal where the /turtlesim node is running. Each time you press one of these keys, you are sending a goal to an action server that is part of the /turtlesim node. The goal is to rotate the turtle to face a particular direction. A message relaying the result of the goal should display once the turtle completes its rotation:

```
[INFO] [turtlesim]: Rotation goal completed successfully
```

The F key will cancel a goal mid-execution.

Try pressing the C key, and then pressing the F key before the turtle can complete its rotation. In the terminal where the /turtlesim node is running, you will see the message:

```
[INFO] [turtlesim]: Rotation goal canceled
```

Not only can the client-side (your input in the teleop) stop a goal, but the server-side (the /turtlesim node) can as well. When the server-side chooses to stop processing a goal, it is said to “abort” the goal.

Try hitting the D key, then the G key before the first rotation can complete. In the terminal where the /turtlesim node is running, you will see the message:

```
[WARN] [turtlesim]: Rotation goal received before a previous goal finished. Aborting...
```

This action server chose to abort the first goal because it got a new one. It could have chosen something else, like reject the new goal or execute the second goal after the first one finished. Don’t assume every action server will choose to abort the current goal when it gets a new one.

### 3 ros2 node info

To see the list of actions a node provides, /turtlesim in this case, open a new terminal and run the command:

```
ros2 node info /turtlesim
```

Which will return a list of /turtlesim’s subscribers, publishers, services, action servers and action clients:

```
/turtlesim
  Subscribers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /turtle1/cmd_vel: geometry_msgs/msg/Twist
  Publishers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /rosout: rcl_interfaces/msg/Log
    /turtle1/color_sensor: turtlesim/msg/Color
    /turtle1/pose: turtlesim/msg/Pose
  Service Servers:
    /clear: std_srvs/srv/Empty
    /kill: turtlesim/srv/Kill
    /reset: std_srvs/srv/Empty
    /spawn: turtlesim/srv/Spawn
    /turtle1/set_pen: turtlesim/srv/SetPen
    /turtle1/teleport_absolute: turtlesim/srv/TeleportAbsolute
    /turtle1/teleport_relative: turtlesim/srv/TeleportRelative
```

(continues on next page)
Notice that the /turtle1/rotate_absolute action for /turtlesim is under Action Servers. This means /turtlesim responds to and provides feedback for the /turtle1/rotate_absolute action.

The /teleop_turtle node has the name /turtle1/rotate_absolute under Action Clients meaning that it sends goals for that action name. To see that, run:

```
ros2 node info /teleop_turtle
```

Which will return:

```
/teleop_turtle
Subscribers:
  /parameter_events: rcl_interfaces/msg/ParameterEvent
Publishers:
  /parameter_events: rcl_interfaces/msg/ParameterEvent
  /rosout: rcl_interfaces/msg/Log
  /turtle1/cmd_vel: geometry_msgs/msg/Twist
Service Servers:
  /teleop_turtle/describe_parameters: rcl_interfaces/srv/DescribeParameters
  /teleop_turtle/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
  /teleop_turtle/get_parameters: rcl_interfaces/srv/ListParameters
  /teleop_turtle/set_parameters: rcl_interfaces/srv/SetParameters
  /teleop_turtle/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
Service Clients:

Action Servers:

Action Clients:
  /turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
```
4 ros2 action list

To identify all the actions in the ROS graph, run the command:

```
ros2 action list
```

Which will return:

```
/turtle1/rotate_absolute
```

This is the only action in the ROS graph right now. It controls the turtle’s rotation, as you saw earlier. You also already know that there is one action client (part of /teleop_turtle) and one action server (part of /turtlesim) for this action from using the `ros2 node info <node_name>` command.

4.1 ros2 action list -t

Actions have types, similar to topics and services. To find /turtle1/rotate_absolute’s type, run the command:

```
ros2 action list -t
```

Which will return:

```
/turtle1/rotate_absolute [turtlesim/action/RotateAbsolute]
```

In brackets to the right of each action name (in this case only /turtle1/rotate_absolute) is the action type, turtlesim/action/RotateAbsolute. You will need this when you want to execute an action from the command line or from code.

5 ros2 action info

You can further introspect the /turtle1/rotate_absolute action with the command:

```
ros2 action info /turtle1/rotate_absolute
```

Which will return:

```
Action: /turtle1/rotate_absolute
Action clients: 1
   /teleop_turtle
Action servers: 1
   /turtlesim
```

This tells us what we learned earlier from running `ros2 node info` on each node: The /teleop_turtle node has an action client and the /turtlesim node has an action server for the /turtle1/rotate_absolute action.
6 ros2 interface show

One more piece of information you will need before sending or executing an action goal yourself is the structure of the action type.

Recall that you identified /turtle1/rotate_absolute's type when running the command `ros2 action list -t`. Enter the following command with the action type in your terminal:

```
ros2 interface show turtlesim/action/RotateAbsolute
```

Which will return:

```
# The desired heading in radians
float32 theta
---
# The angular displacement in radians to the starting position
float32 delta
---
# The remaining rotation in radians
float32 remaining
```

The section of this message above the first --- is the structure (data type and name) of the goal request. The next section is the structure of the result. The last section is the structure of the feedback.

7 ros2 action send_goal

Now let's send an action goal from the command line with the following syntax:

```
ros2 action send_goal <action_name> <action_type> <values>
```

<values> need to be in YAML format.

Keep an eye on the turtlesim window, and enter the following command into your terminal:

```
ros2 action send_goal /turtle1/rotate_absolute turtlesim/action/RotateAbsolute "{theta: 1.57}"
```

You should see the turtle rotating, as well as the following message in your terminal:

```
Waiting for an action server to become available...
Sending goal:
   theta: 1.57

Goal accepted with ID: f8db8f44410849eaa93d3feb747dd444

Result:
   delta: -1.568000316619873

Goal finished with status: SUCCEEDED
```

All goals have a unique ID, shown in the return message. You can also see the result, a field with the name delta, which is the displacement to the starting position.

To see the feedback of this goal, add --feedback to the `ros2 action send_goal` command:
ros2 action send_goal /turtle1/rotate_absolute turtlesim/action/RotateAbsolute "{theta: -1.57}" --feedback

Your terminal will return the message:

Sending goal:
  theta: -1.57

Goal accepted with ID: e6092c831f994afda92f0086f220da27

Feedback:
  remaining: -3.1268222332000732

Feedback:
  remaining: -3.1108222007751465

... 

Result:
  delta: 3.1200008392333984

Goal finished with status: SUCCEEDED

You will continue to receive feedback, the remaining radians, until the goal is complete.

**Summary**

Actions are like services that allow you to execute long running tasks, provide regular feedback, and are cancelable.

A robot system would likely use actions for navigation. An action goal could tell a robot to travel to a position. While the robot navigates to the position, it can send updates along the way (i.e. feedback), and then a final result message once it’s reached its destination.

Turtlesim has an action server that action clients can send goals to for rotating turtles. In this tutorial, you introspected that action, /turtle1/rotate_absolute, to get a better idea of what actions are and how they work.

**Next steps**

Now you’ve covered all of the core ROS 2 concepts. The last few tutorials in the “Users” set will introduce you to some tools and techniques that will make using ROS 2 easier, starting with *Using rqt_console to view logs*.

**Related content**

You can read more about the design decisions behind actions in ROS 2 [here](#).
Using rqt_console to view logs

Goal: Get to know rqt_console, a tool for introspecting log messages.

Tutorial level: Beginner

Time: 5 minutes

Background

rqt_console is a GUI tool used to introspect log messages in ROS 2. Typically, log messages show up in your terminal. With rqt_console, you can collect those messages over time, view them closely and in a more organized manner, filter them, save them and even reload the saved files to introspect at a different time.

Nodes use logs to output messages concerning events and status in a variety of ways. Their content is usually informational, for the sake of the user.

Prerequisites

You will need rqt_console and turtlesim installed.

As always, don’t forget to source ROS 2 in every new terminal you open.

Tasks

1 Setup

Start rqt_console in a new terminal with the following command:

```
ros2 run rqt_console rqt_console
```

The rqt_console window will open:
The first section of the console is where log messages from your system will display.

In the middle you have the option to filter messages by excluding severity levels. You can also add more exclusion filters using the plus-sign button to the right.

The bottom section is for highlighting messages that include a string you input. You can add more filters to this section as well.

Now start `turtlesim` in a new terminal with the following command:

```
ros2 run turtlesim turtlesim_node
```

### 2 Messages on `rqt_console`

To produce log messages for `rqt_console` to display, let’s have the turtle run into the wall. In a new terminal, enter the `ros2 topic pub` command (discussed in detail in the `topics tutorial`) below:

```
ros2 topic pub -r 1 /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0,y: 0.0,z: 0.0}}"
```

Since the above command is publishing the topic at a steady rate, the turtle is continuously running into the wall. In `rqt_console` you will see the same message with the `Warn` severity level displayed over and over, like so:
Press Ctrl+C in the terminal where you ran the `ros2 topic pub` command to stop your turtle from running into the wall.

### 3 Logger levels

ROS 2’s logger levels are ordered by severity:

- **Fatal**
- **Error**
- **Warn**
- **Info**
- **Debug**

There is no exact standard for what each level indicates, but it’s safe to assume that:

- **Fatal** messages indicate the system is going to terminate to try to protect itself from detriment.
- **Error** messages indicate significant issues that won’t necessarily damage the system, but are preventing it from functioning properly.
- **Warn** messages indicate unexpected activity or non-ideal results that might represent a deeper issue, but don’t harm functionality outright.
- **Info** messages indicate event and status updates that serve as a visual verification that the system is running as expected.
• Debug messages detail the entire step-by-step process of the system execution.

The default level is Info. You will only see messages of the default severity level and more-severe levels.

Normally, only Debug messages are hidden because they’re the only level less severe than Info. For example, if you set the default level to Warn, you would only see messages of severity Warn, Error, and Fatal.

3.1 Set the default logger level

You can set the default logger level when you first run the /turtlesim node using remapping. Enter the following command in your terminal:

ros2 run turtlesim turtlesim_node --ros-args --log-level WARN

Now you won’t see the initial Info level messages that came up in the console last time you started turtlesim. That’s because Info messages are lower priority than the new default severity, Warn.

Summary

rqt_console can be very helpful if you need to closely examine the log messages from your system. You might want to examine log messages for any number of reasons, usually to find out where something went wrong and the series of events leading up to that.

Next steps

The next tutorial will teach you about starting multiple nodes at once with ROS 2 Launch.

Launching nodes

Goal: Use a command line tool to launch multiple nodes at once.

Tutorial Level: Beginner

Time: 5 minutes

Contents

• Background
• Prerequisites
• Tasks
  – Running a Launch File
  – (Optional) Control the Turtlesim Nodes
• Summary
• Next steps
Background

In most of the introductory tutorials, you have been opening new terminals for every new node you run. As you create more complex systems with more and more nodes running simultaneously, opening terminals and reentering configuration details becomes tedious.

Launch files allow you to start up and configure a number of executables containing ROS 2 nodes simultaneously.

Running a single launch file with the `ros2 launch` command will start up your entire system - all nodes and their configurations - at once.

Prerequisites

Before starting these tutorials, install ROS 2 by following the instructions on the ROS 2 Installation page.

The commands used in this tutorial assume you followed the binary packages installation guide for your operating system (Debian packages for Linux). You can still follow along if you built from source, but the path to your setup files will likely be different. You also won't be able to use the `sudo apt install ros-<distro>-<package>` command (used frequently in the beginner level tutorials) if you install from source.

If you are using Linux and are not already familiar with the shell, this tutorial will help.

Tasks

Running a Launch File

Open a new terminal and run:

```
ros2 launch turtlesim_multisim.launch.py
```

This command will run the following launch file:

```
# turtlesim/launch/multisim.launch.py
from launch import LaunchDescription
import launch_ros.actions

def generate_launch_description():
    return LaunchDescription([
        launch_ros.actions.Node(
            namespace= "turtlesim1", package='turtlesim', executable='turtlesim_node',
            output='screen'),
        launch_ros.actions.Node(
            namespace= "turtlesim2", package='turtlesim', executable='turtlesim_node',
            output='screen'),
    ])  
```

Note: The launch file above is written in Python, but you can also use XML and YAML to create launch files. You can see a comparison of these different ROS 2 launch formats in Using Python, XML, and YAML for ROS 2 Launch Files.

This will run two turtlesim nodes:
For now, don’t worry about the contents of this launch file. You can find more information on ROS 2 launch in the ROS 2 launch tutorials.

**(Optional) Control the Turtlesim Nodes**

Now that these nodes are running, you can control them like any other ROS 2 nodes. For example, you can make the turtles drive in opposite directions by opening up two additional terminals and running the following commands:

In the second terminal:

```
ros2 topic pub /turtlesim1/turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}}"
```

In the third terminal:

```
ros2 topic pub /turtlesim2/turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: -1.8}}"
```

After running these commands, you should see something like the following:
Summary

The significance of what you’ve done so far is that you’ve run two turtlesim nodes with one command. Once you learn to write your own launch files, you’ll be able to run multiple nodes - and set up their configuration - in a similar way, with the rosv2 launch command.

For more tutorials on ROS 2 launch files, see the main launch file tutorial page.

Next steps

In the next tutorial, Recording and playing back data, you’ll learn about another helpful tool, rosv2 bag.

Recording and playing back data

Goal: Record data published on a topic so you can replay and examine it any time.

Tutorial level: Beginner

Time: 10 minutes
Background

**ros2 bag** is a command line tool for recording data published on topics in your system. It accumulates the data passed on any number of topics and saves it in a database. You can then replay the data to reproduce the results of your tests and experiments. Recording topics is also a great way to share your work and allow others to recreate it.

Prerequisites

You should have **ros2 bag** installed as a part of your regular ROS 2 setup.

If you installed ROS from Debian packages on Linux and your system doesn’t recognize the command, install it like so:

```bash
sudo apt-get install ros-iron-ros2bag \
ros-iron-rosbag2-storage-default-plugins
```

This tutorial talks about concepts covered in previous tutorials, like **nodes** and **topics**. It also uses the **turtlesim package**. As always, don’t forget to source ROS 2 in **every new terminal you open**.

Tasks

1 Setup

You’ll be recording your keyboard input in the **turtlesim** system to save and replay later on, so begin by starting up the **/turtlesim** and **/teleop_turtle** nodes.

Open a new terminal and run:

```bash
ros2 run turtlesim turtlesim_node
```

Open another terminal and run:

```bash
ros2 run turtlesim turtle_teleop_key
```

Let’s also make a new directory to store our saved recordings, just as good practice:

```bash
mkdir bag_files
cd bag_files
```
2 Choose a topic

`ros2 bag` can only record data from published messages in topics. To see the list of your system’s topics, open a new terminal and run the command:

```bash
ros2 topic list
```

Which will return:

```
/parameter_events
/rosout
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

In the topics tutorial, you learned that the `/turtle_teleop` node publishes commands on the `/turtle1/cmd_vel` topic to make the turtle move in turtlesim.

To see the data that `/turtle1/cmd_vel` is publishing, run the command:

```bash
ros2 topic echo /turtle1/cmd_vel
```

Nothing will show up at first because no data is being published by the teleop. Return to the terminal where you ran the teleop and select it so it’s active. Use the arrow keys to move the turtle around, and you will see data being published on the terminal running `ros2 topic echo`.

```
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
```

3 ros2 bag record

3.1 Recording formats

By default, `ros2 bag record` will record data files using the MCAP file format (.mcap).

To record files using the SQLite3 file format (.db3), add the `--storage sqlite3` flag (or `-s sqlite3`) to your `ros2 bag record` commands.

For more information on ROS 2 storage plugin options, check out the following resources:

- MCAP
- SQLite3
3.2 Record a single topic

To record the data published to a topic use the command syntax:

```
ros2 bag record <topic_name>
```

Before running this command on your chosen topic, open a new terminal and move into the `bag_files` directory you created earlier, because the rosbag file will save in the directory where you run it.

Run the command:

```
ros2 bag record /turtle1/cmd_vel
```

You will see the following messages in the terminal (the date and time will be different):

```
[INFO] [rosbag2_storage]: Opened database 'rosbag2_2019_10_11-05_18_45'.
[INFO] [rosbag2_transport]: Listening for topics...
[INFO] [rosbag2_transport]: Subscribed to topic '/turtle1/cmd_vel'
[INFO] [rosbag2_transport]: All requested topics are subscribed. Stopping discovery...
```

Now `ros2 bag` is recording the data published on the `/turtle1/cmd_vel` topic. Return to the teleop terminal and move the turtle around again. The movements don’t matter, but try to make a recognizable pattern to see when you replay the data later.
Press Ctrl+C to stop recording.

The data will be accumulated in a new bag directory with a name in the pattern of `rosbag2_year_month_day-hour_minute_second`. This directory will contain a `metadata.yaml` along with the bag file in the recorded format.

### 3.3 Record multiple topics

You can also record multiple topics, as well as change the name of the file `ros2_bag` saves to.

Run the following command:

```
ros2 bag record -o subset /turtle1/cmd_vel /turtle1/pose
```

The `-o` option allows you to choose a unique name for your bag file. The following string, in this case `subset`, is the file name.

To record more than one topic at a time, simply list each topic separated by a space.

You will see the following message, confirming that both topics are being recorded.
[INFO] [rosbag2_storage]: Opened database 'subset'.
[INFO] [rosbag2_transport]: Listening for topics...
[INFO] [rosbag2_transport]: Subscribed to topic '/turtle1/cmd_vel'
[INFO] [rosbag2_transport]: Subscribed to topic '/turtle1/pose'
[INFO] [rosbag2_transport]: All requested topics are subscribed. Stopping discovery...

You can move the turtle around and press Ctrl+C when you’re finished.

**Note:** There is another option you can add to the command, -a, which records all the topics on your system.

### 4 ros2 bag info

You can see details about your recording by running:

```
ros2 bag info <bag_file_name>
```

Running this command on the `subset` bag file will return a list of information on the file:

```
Files:          subset.mcap
Bag size:       228.5 KiB
Storage id:     mcap
Duration:       48.47s
Start:          Oct 11 2019 06:09:09.12 (1570799349.12)
End:            Oct 11 2019 06:09:57.60 (1570799397.60)
Messages:       3013
Topic information: Topic: /turtle1/cmd_vel | Type: geometry_msgs/msg/Twist | Count: 9 |
                  Serialization Format: cdr
                  Topic: /turtle1/pose | Type: turtlesim/msg/Pose | Count: 3004 |
                  Serialization Format: cdr
```

To view the individual messages, you would have to open up the database, in this case sqlite3, to examine it, which is beyond the scope of ROS 2.

### 5 ros2 bag play

Before replaying the bag file, enter Ctrl+C in the terminal where the teleop is running. Then make sure your turtlesim window is visible so you can see the bag file in action.

Enter the command:

```
ros2 bag play subset
```

The terminal will return the message:

```
[INFO] [rosbag2_storage]: Opened database 'subset'.
```

Your turtle will follow the same path you entered while recording (though not 100% exactly; turtlesim is sensitive to small changes in the system’s timing).
Because the subset file recorded the `/turtle1/pose` topic, the `ros2 bag play` command won’t quit for as long as you had turtlesim running, even if you weren’t moving.

This is because as long as the `/turtlesim` node is active, it publishes data on the `/turtle1/pose` topic at regular intervals. You may have noticed in the `ros2 bag info` example result above that the `/turtle1/cmd_vel` topic’s Count information was only 9; that’s how many times we pressed the arrow keys while recording.

Notice that `/turtle1/pose` has a Count value of over 3000; while we were recording, data was published on that topic 3000 times.

To get an idea of how often position data is published, you can run the command:

```
ros2 topic hz /turtle1/pose
```
Summary

You can record data passed on topics in your ROS 2 system using the `ros2 bag` command. Whether you’re sharing your work with others or introspecting your own experiments, it’s a great tool to know about.

Next steps

You’ve completed the “Beginner: CLI Tools” tutorials! The next step is tackling the “Beginner: Client Libraries” tutorials, starting with Creating a workspace.

Related content

A more thorough explanation of `ros2 bag` can be found in the README here. For more information on QoS compatibility and `ros2 bag`, see rosbag2: Overriding QoS Policies.

Beginner: Client libraries

Using colcon to build packages

Table of Contents

- Background
- Prerequisites
  - Install colcon
  - Install ROS 2
- Basics
  - Create a workspace
  - Add some sources
  - Source an underlay
  - Build the workspace
  - Run tests
  - Source the environment
  - Try a demo
- Create your own package
- Setup colcon_cd
- Setup colcon tab completion
- Tips

Goal: Build a ROS 2 workspace with colcon.

Tutorial level: Beginner

Time: 20 minutes
This is a brief tutorial on how to create and build a ROS 2 workspace with colcon. It is a practical tutorial and not designed to replace the core documentation.

**Background**

colcon is an iteration on the ROS build tools catkin_make, catkin_make_isolated, catkin_tools and ament_tools. For more information on the design of colcon see this document.

The source code can be found in the [colcon GitHub organization](https://github.com).  

**Prerequisites**

**Install colcon**

Linux

```bash
sudo apt install python3-colcon-common-extensions
```

macOS

```bash
python3 -m pip install colcon-common-extensions
```

Windows

```bash
pip install -U colcon-common-extensions
```

**Install ROS 2**

To build the samples, you will need to install ROS 2.

Follow the [installation instructions](https://www.ros.org/).  

**Attention:** If installing from Debian packages, this tutorial requires the *desktop installation.*

**Basics**

A ROS workspace is a directory with a particular structure. Commonly there is a `src` subdirectory. Inside that subdirectory is where the source code of ROS packages will be located. Typically the directory starts otherwise empty.

colcon does out of source builds. By default it will create the following directories as peers of the `src` directory:

- The `build` directory will be where intermediate files are stored. For each package a subfolder will be created in which e.g. CMake is being invoked.
- The `install` directory is where each package will be installed to. By default each package will be installed into a separate subdirectory.
- The `log` directory contains various logging information about each colcon invocation.

**Note:** Compared to catkin there is no `devel` directory.
Create a workspace

First, create a directory (ros2_ws) to contain our workspace:

Linux

```
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws
```

macOS

```
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws
```

Windows

```
md \dev\ros2_ws\src
cd \dev\ros2_ws
```

At this point the workspace contains a single empty directory src:

```
|-- src
`-- 1 directory, 0 files
```

Add some sources

Let’s clone the examples repository into the src directory of the workspace:

```
git clone https://github.com/ros2/examples src/examples -b iron
```

Now the workspace should have the source code to the ROS 2 examples:

```
|-- src
   |-- examples
      |-- CONTRIBUTING.md
      |-- LICENSE
      |-- rclcpp
      |-- rclpy
      `-- README.md

4 directories, 3 files
```
Source an underlay

It is important that we have sourced the environment for an existing ROS 2 installation that will provide our workspace with the necessary build dependencies for the example packages. This is achieved by sourcing the setup script provided by a binary installation or a source installation, i.e. another colcon workspace (see *Installation*). We call this environment an **underlay**.

Our workspace, ros2_ws, will be an **overlay** on top of the existing ROS 2 installation. In general, it is recommended to use an overlay when you plan to iterate on a small number of packages, rather than putting all of your packages into the same workspace.

Build the workspace

**Attention:** To build packages on Windows you need to be in a Visual Studio environment, see *Building the ROS 2 Code* for more details.

In the root of the workspace, run colcon build. Since build types such as ament_cmake do not support the concept of the *devel* space and require the package to be installed, colcon supports the option `--symlink-install`. This allows the installed files to be changed by changing the files in the *source* space (e.g. Python files or other non-compiled resources) for faster iteration.

Linux

```
colcon build --symlink-install
```

macOS

```
colcon build --symlink-install
```

Windows

```
colcon build --symlink-install --merge-install
```

Windows doesn’t allow long paths, so `merge-install` will combine all the paths into the *install* directory.

After the build is finished, we should see the **build**, **install**, and **log** directories:

```
build
├── install
├── log
└── src
```

4 directories, 0 files
Run tests

To run tests for the packages we just built, run the following:

Linux

```
colcon test
```

macOS

```
colcon test
```

Windows

Remember to use a x64 Native Tools Command Prompt for VS 2019 for executing the following command, as we are going to build a workspace.

```
colcon test --merge-install
```

You also need to specify `--merge-install` here since we used it for building above.

Source the environment

When colcon has completed building successfully, the output will be in the install directory. Before you can use any of the installed executables or libraries, you will need to add them to your path and library paths. colcon will have generated bash/bat files in the install directory to help set up the environment. These files will add all of the required elements to your path and library paths as well as provide any bash or shell commands exported by packages.

Linux

```
source install/setup.bash
```

macOS

```
. install/setup.bash
```

Windows

```
call install\setup.bat
```

Or with Powershell:

```
install\setup.ps1
```

Try a demo

With the environment sourced, we can run executables built by colcon. Let’s run a subscriber node from the examples:

```
ros2 run examples_rclcpp_minimal_subscriber subscriber_member_function
```

In another terminal, let’s run a publisher node (don’t forget to source the setup script):

```
ros2 run examples_rclcpp_minimal_publisher publisher_member_function
```

You should see messages from the publisher and subscriber with numbers incrementing.
Create your own package

colcon uses the `package.xml` specification defined in REP 149 (format 2 is also supported).
colcon supports multiple build types. The recommended build types are `ament_cmake` and `ament_python`. Also supported are pure `cmake` packages.
An example of an `ament_python` build is the `ament_index_python` package, where the setup.py is the primary entry point for building.
A package such as `demo_nodes_cpp` uses the `ament_cmake` build type, and uses CMake as the build tool.
For convenience, you can use the tool `ros2 pkg create` to create a new package based on a template.

Note: For catkin users, this is the equivalent of `catkin_create_package`.

Setup colcon_cd

The command `colcon_cd` allows you to quickly change the current working directory of your shell to the directory of a package. As an example `colcon_cd some_ros_package` would quickly bring you to the directory `~/ros2_ws/src/some_ros_package`.

Linux

```
echo "source /usr/share/colcon_cd/function/colcon_cd.sh" >> ~/.bashrc
echo "export _colcon_cd_root=/opt/ros/iron/" >> ~/.bashrc
```

macOS

```
echo "source /usr/local/share/colcon_cd/function/colcon_cd.sh" >> ~/.bashrc
echo "export _colcon_cd_root=~/.ros2_install" >> ~/.bashrc
```

Windows

Not yet available

Depending on the way you installed `colcon_cd` and where your workspace is, the instructions above may vary, please refer to the documentation for more details. To undo this in Linux and macOS, locate your system’s shell startup script and remove the appended source and export commands.

Setup colcon tab completion

The command `colcon` supports command completion for bash and bash-like shells if the `colcon-argcomplete` package is installed.

Linux

```
echo "source /usr/share/colcon_argcomplete/hook/colcon-argcomplete.bash" >> ~/.bashrc
```

macOS

```
echo "source $HOME/.local/share/colcon_argcomplete/hook/colcon-argcomplete.bash" >> ~/.bash_profile
```
Windows

Not yet available

Depending on the way you installed colcon and where your workspace is, the instructions above may vary, please refer to the documentation for more details. To undo this in Linux and macOS, locate your system’s shell startup script and remove the appended source command.

Tips

- If you do not want to build a specific package place an empty file named COLCON_IGNORE in the directory and it will not be indexed.
- If you want to avoid configuring and building tests in CMake packages you can pass: `--cmake-args -DBUILD_TESTING=0`.
- If you want to run a single particular test from a package:

  ```
  colcon test --packages-select YOUR_PKG_NAME --ctest-args -R YOUR_TEST_IN_PKG
  ```

Creating a workspace

**Goal:** Create a workspace and learn how to set up an overlay for development and testing.

**Tutorial level:** Beginner

**Time:** 20 minutes

<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Background</strong></td>
</tr>
<tr>
<td><strong>Prerequisites</strong></td>
</tr>
<tr>
<td><strong>Tasks</strong></td>
</tr>
<tr>
<td>- 1 Source ROS 2 environment</td>
</tr>
<tr>
<td>- 2 Create a new directory</td>
</tr>
<tr>
<td>- 3 Clone a sample repo</td>
</tr>
<tr>
<td>- 4 Resolve dependencies</td>
</tr>
<tr>
<td>- 5 Build the workspace with colcon</td>
</tr>
<tr>
<td>- 6 Source the overlay</td>
</tr>
<tr>
<td>- 7 Modify the overlay</td>
</tr>
<tr>
<td><strong>Summary</strong></td>
</tr>
<tr>
<td><strong>Next steps</strong></td>
</tr>
</tbody>
</table>
**Background**

A workspace is a directory containing ROS 2 packages. Before using ROS 2, it’s necessary to source your ROS 2 installation workspace in the terminal you plan to work in. This makes ROS 2’s packages available for you to use in that terminal.

You also have the option of sourcing an “overlay” - a secondary workspace where you can add new packages without interfering with the existing ROS 2 workspace that you’re extending, or “underlay”. Your underlay must contain the dependencies of all the packages in your overlay. Packages in your overlay will override packages in the underlay. It’s also possible to have several layers of underlays and overlays, with each successive overlay using the packages of its parent underlays.

**Prerequisites**

- ROS 2 installation
- colcon installation
- git installation
- turtlesim installation
- Have rosdep installed
- Understanding of basic terminal commands (here’s a guide for Linux)
- Text editor of your choice

**Tasks**

**1 Source ROS 2 environment**

Your main ROS 2 installation will be your underlay for this tutorial. (Keep in mind that an underlay does not necessarily have to be the main ROS 2 installation.)

Depending on how you installed ROS 2 (from source or binaries), and which platform you’re on, your exact source command will vary:

Linux

```bash
source /opt/ros/iron/setup.bash
```

macOS

```bash
. ~/ros2_install/ros2-osx/setup.bash
```

Windows

Remember to use a x64 Native Tools Command Prompt for VS 2019 for executing the following commands, as we are going to build a workspace.

```bash
call C:\dev\ros2\local_setup.bat
```

Consult the installation guide you followed if these commands don’t work for you.
2 Create a new directory

Best practice is to create a new directory for every new workspace. The name doesn’t matter, but it is helpful to have it indicate the purpose of the workspace. Let’s choose the directory name ros2_ws, for “development workspace”:

Linux

```bash
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws/src
```

macOS

```bash
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws/src
```

Windows

```bash
md \ros2_ws\src
cd \ros2_ws\src
```

Another best practice is to put any packages in your workspace into the src directory. The above code creates a src directory inside ros2_ws and then navigates into it.

3 Clone a sample repo

Ensure you’re still in the ros2_ws/src directory before you clone.

In the rest of the beginner developer tutorials, you will create your own packages, but for now you will practice putting a workspace together using existing packages.

If you went through the Beginner: CLI Tools tutorials, you’ll be familiar with turtlesim, one of the packages in ros_tutorials.

A repo can have multiple branches. You need to check out the one that targets your installed ROS 2 distro. When you clone this repo, add the -b argument followed by that branch.

In the ros2_ws/src directory, run the following command:

```bash
git clone https://github.com/ros/ros_tutorials.git -b iron
```

Now ros_tutorials is cloned in your workspace. The ros_tutorials repository contains the turtlesim package, which we’ll use in the rest of this tutorial. The other packages in this repository are not built because they contain a COLCON_IGNORE file.

So far you have populated your workspace with a sample package, but it isn’t a fully-functional workspace yet. You need to resolve the dependencies first and then build the workspace.
4 Resolve dependencies

Before building the workspace, you need to resolve the package dependencies. You may have all the dependencies already, but best practice is to check for dependencies every time you clone. You wouldn’t want a build to fail after a long wait only to realize that you have missing dependencies.

From the root of your workspace (ros2_ws), run the following command:

Linux

```bash
# cd if you're still in the "src" directory with the "ros_tutorials" clone
cd ..
rosdep install -i --from-path src --rosdistro iron -y
```

macOS

rosdep only runs on Linux, so you can skip ahead to section “5 Build the workspace with colcon”.

Windows

rosdep only runs on Linux, so you can skip ahead to section “5 Build the workspace with colcon”.

If you installed ROS 2 on Linux from source or the “fat” archive, you will need to use the rosdep command from their installation instructions. Here are the from-source rosdep section and the “fat” archive rosdep section.

If you already have all your dependencies, the console will return:

```
#All required rosdeps installed successfully
```

Packages declare their dependencies in the package.xml file (you will learn more about packages in the next tutorial). This command walks through those declarations and installs the ones that are missing. You can learn more about rosdep in another tutorial (coming soon).

5 Build the workspace with colcon

From the root of your workspace (ros2_ws), you can now build your packages using the command:

Linux

```
colcon build
```

macOS

```
colcon build
```

Windows

```
colcon build --merge-install
```

Windows doesn’t allow long paths, so merge-install will combine all the paths into the install directory.

The console will return the following message:

```
Starting >>> turtlesim
Finished <<< turtlesim [5.49s]
Summary: 1 package finished [5.58s]
```
Note: Other useful arguments for `colcon build`:

- `--packages-up-to` builds the package you want, plus all its dependencies, but not the whole workspace (saves time)
- `--symlink-install` saves you from having to rebuild every time you tweak python scripts
- `--event-handlers console_direct+` shows console output while building (can otherwise be found in the log directory)

Once the build is finished, enter `ls` in the workspace root (`~/ros2_ws`) and you will see that colcon has created new directories:

```
build  install  log  src
```

The `install` directory is where your workspace’s setup files are, which you can use to source your overlay.

### 6 Source the overlay

Before sourcing the overlay, it is very important that you open a new terminal, separate from the one where you built the workspace. Sourcing an overlay in the same terminal where you built, or likewise building where an overlay is sourced, may create complex issues.

In the new terminal, source your main ROS 2 environment as the “underlay”, so you can build the overlay “on top of” it:

**Linux**

```
source /opt/ros/iron/setup.bash
```

**macOS**

```
. ~/ros2_install/ros2-osx/setup.bash
```

**Windows**

In this case you can use a normal command prompt, as we are not going to build any workspace in this terminal.

```
call C:\dev\ros2\local_setup.bat
```

Go into the root of your workspace:

**Linux**

```
cd ~/ros2_ws
```

**macOS**

```
cd ~/ros2_ws
```

**Windows**

```
cd \ros2_ws
```

In the root, source your overlay:

**Linux**

```
```
source install/local_setup.bash

macOS

. install/local_setup.bash

Windows

call install\setup.bat

Note: Sourcing the local_setup of the overlay will only add the packages available in the overlay to your environment. setup sources the overlay as well as the underlay it was created in, allowing you to utilize both workspaces.

So, sourcing your main ROS 2 installation’s setup and then the ros2_ws overlay’s local_setup, like you just did, is the same as just sourcing ros2_ws’s setup, because that includes the environment of its underlay.

Now you can run the turtlesim package from the overlay:

ros2 run turtlesim turtlesim_node

But how can you tell that this is the overlay turtlesim running, and not your main installation’s turtlesim?

Let’s modify turtlesim in the overlay so you can see the effects:

• You can modify and rebuild packages in the overlay separately from the underlay.
• The overlay takes precedence over the underlay.

7 Modify the overlay

You can modify turtlesim in your overlay by editing the title bar on the turtlesim window. To do this, locate the turtle_frame.cpp file in ~/ros2_ws/src/ros_tutorials/turtlesim/src. Open turtle_frame.cpp with your preferred text editor.

On line 52 you will see the function setWindowTitle("TurtleSim");. Change the value "TurtleSim" to "MyTurtleSim", and save the file.

Return to the first terminal where you ran colcon build earlier and run it again.

Return to the second terminal (where the overlay is sourced) and run turtlesim again:

ros2 run turtlesim turtlesim_node

You will see the title bar on the turtlesim window now says “MyTurtleSim”.
Even though your main ROS 2 environment was sourced in this terminal earlier, the overlay of your `ros2_ws` environment takes precedence over the contents of the underlay.

To see that your underlay is still intact, open a brand new terminal and source only your ROS 2 installation. Run `turtlesim` again:

```
ros2 run turtlesim turtlesim_node
```
You can see that modifications in the overlay did not actually affect anything in the underlay.

Summary

In this tutorial, you sourced your main ROS 2 distro install as your underlay, and created an overlay by cloning and building packages in a new workspace. The overlay gets prepended to the path, and takes precedence over the underlay, as you saw with your modified turtlesim.

Using overlays is recommended for working on a small number of packages, so you don’t have to put everything in the same workspace and rebuild a huge workspace on every iteration.
Next steps

Now that you understand the details behind creating, building and sourcing your own workspace, you can learn how to create your own packages.

Creating a package

Goal: Create a new package using either CMake or Python, and run its executable.

Tutorial level: Beginner

Time: 15 minutes

Contents

• Background
  – 1 What is a ROS 2 package?
  – 2 What makes up a ROS 2 package?
  – 3 Packages in a workspace
• Prerequisites
• Tasks
  – 1 Create a package
  – 2 Build a package
  – 3 Source the setup file
  – 4 Use the package
  – 5 Examine package contents
  – 6 Customize package.xml
• Summary
• Next steps

Background

1 What is a ROS 2 package?

A package is an organizational unit for your ROS 2 code. If you want to be able to install your code or share it with others, then you'll need it organized in a package. With packages, you can release your ROS 2 work and allow others to build and use it easily.

Package creation in ROS 2 uses ament as its build system and colcon as its build tool. You can create a package using either CMake or Python, which are officially supported, though other build types do exist.
2 What makes up a ROS 2 package?

ROS 2 Python and CMake packages each have their own minimum required contents:

**CMake**

- **CMakeLists.txt** file that describes how to build the code within the package
- **include/**<package_name> directory containing the public headers for the package
- **package.xml** file containing meta information about the package
- **src** directory containing the source code for the package

**Python**

- **package.xml** file containing meta information about the package
- **resource/**<package_name> marker file for the package
- **setup.cfg** is required when a package has executables, so `ros2 run` can find them
- **setup.py** containing instructions for how to install the package
- **<package_name>** - a directory with the same name as your package, used by ROS 2 tools to find your package, contains __init__.py

The simplest possible package may have a file structure that looks like:

**CMake**

```plaintext
my_package/
  CMakeLists.txt
  include/my_package/
  package.xml
  src/
```

**Python**

```plaintext
my_package/
  package.xml
  resource/my_package
  setup.cfg
  setup.py
  my_package/
```

3 Packages in a workspace

A single workspace can contain as many packages as you want, each in their own folder. You can also have packages of different build types in one workspace (CMake, Python, etc.). You cannot have nested packages.

Best practice is to have a **src** folder within your workspace, and to create your packages in there. This keeps the top level of the workspace “clean”.

A trivial workspace might look like:

```plaintext
workspace_folder/
  src/
    cpp_package_1/
```

(continues on next page)
Prerequisites

You should have a ROS 2 workspace after following the instructions in the previous tutorial. You will create your package in this workspace.

Tasks

1 Create a package

First, source your ROS 2 installation.

Let's use the workspace you created in the previous tutorial, ros2_ws, for your new package.

Make sure you are in the src folder before running the package creation command.

Linux

```
cd ~/ros2_ws/src
```

macOS

```
cd ~/ros2_ws/src
```

Windows

```
cd \ros2_ws\src
```

The command syntax for creating a new package in ROS 2 is:

CMake

```
ros2 pkg create --build-type ament_cmake <package_name>
```

Python
ros2 pkg create --build-type ament_python <package_name>

For this tutorial, you will use the optional arguments --node-name and --license. --node-name option creates a simple Hello World type executable in the package, and --license declares the license information for the package.

Enter the following command in your terminal:

CMake

ros2 pkg create --build-type ament_cmake --node-name my_node my_package --license Apache-2.0

Python

ros2 pkg create --build-type ament_python --node-name my_node my_package --license Apache-2.0

You will now have a new folder within your workspace’s src directory called my_package.

After running the command, your terminal will return the message:

CMake

going to create a new package
package name: my_package
destination directory: /home/user/ros2_ws/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['<name> <email>']
licenses: ['Apache-2.0']
build type: ament_cmake
dependencies: []
node_name: my_node
creating folder ./my_package
creating ./my_package/package.xml
creating source and include folder
creating folder ./my_package/src
creating folder ./my_package/include/my_package
creating ./my_package/CMakeLists.txt
creating ./my_package/src/my_node.cpp

Python

going to create a new package
package name: my_package
destination directory: /home/user/ros2_ws/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['<name> <email>']
licenses: ['Apache-2.0']
build type: ament_python
dependencies: []
node_name: my_node
You can see the automatically generated files for the new package.

### 2 Build a package

Putting packages in a workspace is especially valuable because you can build many packages at once by running `colcon build` in the workspace root. Otherwise, you would have to build each package individually.

Return to the root of your workspace:

Linux

```
cd ~/ros2_ws
```

macOS

```
cd ~/ros2_ws
```

Windows

```
cd \ros2_ws
```

Now you can build your packages:

Linux

```
colcon build
```

macOS

```
colcon build
```

Windows

```
colcon build --merge-install
```

Windows doesn’t allow long paths, so `merge-install` will combine all the paths into the `install` directory.

Recall from the last tutorial that you also have the `ros_tutorials` packages in your `ros2_ws`. You might have noticed that running `colcon build` also built the `turtlesim` package. That’s fine when you only have a few packages in your workspace, but when there are many packages, `colcon build` can take a long time.
To build only the `my_package` package next time, you can run:

```bash
colcon build --packages-select my_package
```

### 3 Source the setup file

To use your new package and executable, first open a new terminal and source your main ROS 2 installation. Then, from inside the `ros2_ws` directory, run the following command to source your workspace:

**Linux**

```bash
source install/local_setup.bash
```

**macOS**

```bash
. install/local_setup.bash
```

**Windows**

```bash
call install/local_setup.bat
```

Now that your workspace has been added to your path, you will be able to use your new package’s executables.

### 4 Use the package

To run the executable you created using the `--node-name` argument during package creation, enter the command:

```bash
ros2 run my_package my_node
```

Which will return a message to your terminal:

**CMake**

```
hello world my_package package
```

**Python**

```
Hi from my_package.
```

### 5 Examine package contents

Inside `ros2_ws/src/my_package`, you will see the files and folders that `ros2 pkg create` automatically generated:

**CMake**

```
CMakeLists.txt include package.xml src
```

`my_node.cpp` is inside the `src` directory. This is where all your custom C++ nodes will go in the future.

**Python**

```
my_package package.xml resource setup.cfg setup.py test
```

`my_node.py` is inside the `my_package` directory. This is where all your custom Python nodes will go in the future.
6 Customize package.xml

You may have noticed in the return message after creating your package that the fields description and license contain TODO notes. That’s because the package description and license declaration are not automatically set, but are required if you ever want to release your package. The maintainer field may also need to be filled in.

From ros2_ws/src/my_package, open package.xml using your preferred text editor:

CMake

```xml
<?xml version="1.0"?>
<?xml-model
  href="http://download.ros.org/schema/package_format3.xsd"
  schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>my_package</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="user@todo.todo">user</maintainer>
  <license>TODO: License declaration</license>
  <buildtool_depend>ament_cmake</buildtool_depend>
  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>
  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

Python

```xml
<?xml version="1.0"?>
<?xml-model
  href="http://download.ros.org/schema/package_format3.xsd"
  schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>my_package</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="user@todo.todo">user</maintainer>
  <license>TODO: License declaration</license>
  <test_depend>ament_copyright</test_depend>
  <test_depend>ament_flake8</test_depend>
  <test_depend>ament_pep257</test_depend>
  <test_depend>python3-pytest</test_depend>
  <export>
    <build_type>ament_python</build_type>
  </export>
</package>
```

Input your name and email on the maintainer line if it hasn’t been automatically populated for you. Then, edit the
description line to summarize the package:

```
<description>Beginner client libraries tutorials practice package</description>
```

Then, update the license line. You can read more about open source licenses [here](#). Since this package is only for practice, it's safe to use any license. We'll use Apache License 2.0:

```
<license>Apache License 2.0</license>
```

Don’t forget to save once you’re done editing.

Below the license tag, you will see some tag names ending with _depend. This is where your `package.xml` would list its dependencies on other packages, for colcon to search for. `my_package` is simple and doesn’t have any dependencies, but you will see this space being utilized in upcoming tutorials.

**CMake**

You’re all done for now!

**Python**

The `setup.py` file contains the same description, maintainer and license fields as `package.xml`, so you need to set those as well. They need to match exactly in both files. The version and name (`package_name`) also need to match exactly, and should be automatically populated in both files.

Open `setup.py` with your preferred text editor.

```python
from setuptools import find_packages, setup

package_name = 'my_py_pkg'

setup(
    name=package_name,
    version='0.0.0',
    packages=find_packages(exclude=['test']),
    data_files=[
        ('share/ament_index/resource_index/packages', ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='TODO',
    maintainer_email='TODO',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'my_node = my_py_pkg.my_node:main'
        ],
    },
)
```

Edit the maintainer, maintainer_email, and description lines to match `package.xml`.

Don’t forget to save the file.
Summary

You’ve created a package to organize your code and make it easy to use for others.

Your package was automatically populated with the necessary files, and then you used colcon to build it so you can use its executables in your local environment.

Next steps

Next, let’s add something meaningful to a package. You’ll start with a simple publisher/subscriber system, which you can choose to write in either C++ or Python.

Writing a simple publisher and subscriber (C++)

Goal: Create and run a publisher and subscriber node using C++.

Tutorial level: Beginner

Time: 20 minutes

Contents

• Background
• Prerequisites
• Tasks
  – 1 Create a package
  – 2 Write the publisher node
  – 3 Write the subscriber node
  – 4 Build and run
• Summary
• Next steps
• Related content

Background

Nodes are executable processes that communicate over the ROS graph. In this tutorial, the nodes will pass information in the form of string messages to each other over a topic. The example used here is a simple “talker” and “listener” system; one node publishes data and the other subscribes to the topic so it can receive that data.

The code used in these examples can be found here.
Prerequisites

In previous tutorials, you learned how to create a workspace and create a package.

Tasks

1 Create a package

Open a new terminal and source your ROS 2 installation so that ros2 commands will work.

Navigate into the ros2_ws directory created in a previous tutorial.

Recall that packages should be created in the src directory, not the root of the workspace. So, navigate into ros2_ws/src, and run the package creation command:

```
ros2 pkg create --build-type ament_cmake cpp_pubsub
```

Your terminal will return a message verifying the creation of your package cpp_pubsub and all its necessary files and folders.

Navigate into ros2_ws/src/cpp_pubsub/src. Recall that this is the directory in any CMake package where the source files containing executables belong.

2 Write the publisher node

Download the example talker code by entering the following command:

Linux

```
wget -O publisher_member_function.cpp https://raw.githubusercontent.com/ros2/examples/iron/rclcpp/topics/minimal_publisher/member_function.cpp
```

macOS

```
wget -O publisher_member_function.cpp https://raw.githubusercontent.com/ros2/examples/iron/rclcpp/topics/minimal_publisher/member_function.cpp
```

Windows

In a Windows command line prompt:

```
curl -sk https://raw.githubusercontent.com/ros2/examples/iron/rclcpp/topics/minimal_publisher/member_function.cpp -o publisher_member_function.cpp
```

Or in powershell:

```
curl https://raw.githubusercontent.com/ros2/examples/iron/rclcpp/topics/minimal_publisher/member_function.cpp -o publisher_member_function.cpp
```

Now there will be a new file named publisher_member_function.cpp. Open the file using your preferred text editor.
#include <chrono>
#include <functional>
#include <memory>
#include <string>
#include <rclcpp/rclcpp.hpp>
#include "std_msgs/msg/string.hpp"

using namespace std::chrono_literals;

/* This example creates a subclass of Node and uses std::bind() to register a
 * member function as a callback from the timer. */

class MinimalPublisher : public rclcpp::Node
{
  public:
    MinimalPublisher()
      : Node("minimal_publisher"), count_(0)
    {
      publisher_ = this->create_publisher<std_msgs::msg::String>("topic", 10);
      timer_ = this->create_wall_timer(
        500ms, std::bind(&MinimalPublisher::timer_callback, this));
    }

  private:
    void timer_callback()
    {
      auto message = std_msgs::msg::String();
      message.data = "Hello, world! " + std::to_string(count_++);
      RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_str());
      publisher_->publish(message);
    }

    rclcpp::TimerBase::SharedPtr timer_;
    rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
    size_t count_;}

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<MinimalPublisher>());
  rclcpp::shutdown();
  return 0;
}
2.1 Examine the code

The top of the code includes the standard C++ headers you will be using. After the standard C++ headers is the rclcpp/rclcpp.hpp include which allows you to use the most common pieces of the ROS 2 system. Last is std_msgs/msg/string.hpp, which includes the built-in message type you will use to publish data.

```cpp
#include <chrono>
#include <functional>
#include <memory>
#include <string>
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"
using namespace std::chrono_literals;
```

These lines represent the node’s dependencies. Recall that dependencies have to be added to package.xml and CMakeLists.txt, which you’ll do in the next section.

The next line creates the node class MinimalPublisher by inheriting from rclcpp::Node. Every this in the code is referring to the node.

```cpp
class MinimalPublisher : public rclcpp::Node
```

The public constructor names the node minimal_publisher and initializes count_ to 0. Inside the constructor, the publisher is initialized with the String message type, the topic name topic, and the required queue size to limit messages in the event of a backup. Next, timer_ is initialized, which causes the timer_callback function to be executed twice a second.

```cpp
public:
    MinimalPublisher()
    : Node("minimal_publisher"), count_(0)
    {
        publisher_ = this->create_publisher<std_msgs::msg::String>("topic", 10);
        timer_ = this->create_wall_timer(500ms, std::bind(&MinimalPublisher::timer_callback, this));
    }
```

The timer_callback function is where the message data is set and the messages are actually published. The RCLCPP_INFO macro ensures every published message is printed to the console.

```cpp
private:
    void timer_callback()
    {
        auto message = std_msgs::msg::String();
        message.data = "Hello, world! " + std::to_string(count_++);
        RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_str());
        publisher_->publish(message);
    }
```

Last is the declaration of the timer, publisher, and counter fields.

```cpp
rclcpp::TimerBase::SharedPtr timer_;  
rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;  
size_t count_;  
```
Following the `MinimalPublisher` class is `main`, where the node actually executes. `rclcpp::init` initializes ROS 2, and `rclcpp::spin` starts processing data from the node, including callbacks from the timer.

```cpp
int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<MinimalPublisher>());
  rclcpp::shutdown();
  return 0;
}
```

### 2.2 Add dependencies

Navigate one level back to the `ros2_ws/src/cpp_pubsub` directory, where the `CMakeLists.txt` and `package.xml` files have been created for you.

Open `package.xml` with your text editor.

As mentioned in the previous tutorial, make sure to fill in the `<description>`, `<maintainer>` and `<license>` tags:

```xml
<description>Examples of minimal publisher/subscriber using rclcpp</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache License 2.0</license>
```

Add a new line after the `ament_cmake` buildtool dependency and paste the following dependencies corresponding to your node’s include statements:

```xml
<depend>rclcpp</depend>
<depend>std_msgs</depend>
```

This declares the package needs `rclcpp` and `std_msgs` when its code is built and executed.

Make sure to save the file.

### 2.3 CMakeLists.txt

Now open the `CMakeLists.txt` file. Below the existing dependency `find_package(ament_cmake REQUIRED)`, add the lines:

```cpp
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)
```

After that, add the executable and name it `talker` so you can run your node using `ros2 run`:

```cpp
add_executable(talker src/publisher_member_function.cpp)
ament_target_dependencies(talker rclcpp std_msgs)
```

Finally, add the `install(TARGETS...)` section so `ros2 run` can find your executable:

```cpp
install(TARGETS
talker
  DESTINATION lib/${PROJECT_NAME})
```

You can clean up your `CMakeLists.txt` by removing some unnecessary sections and comments, so it looks like this:
```cpp
cmake_minimum_required(VERSION 3.5)
project(cpp_pubsub)
#
# Default to C++14
if(NOT CMAKE_CXX_STANDARD)
   set(CMAKE_CXX_STANDARD 14)
endif()
if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
   add_compile_options(-Wall -Wextra -Wpedantic)
endif()

find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)

add_executable(talker src/publisher_member_function.cpp)
ament_target_dependencies(talker rclcpp std_msgs)

install(TARGETS
talker
   DESTINATION lib/${PROJECT_NAME})
ament_package()
```

You could build your package now, source the local setup files, and run it, but let’s create the subscriber node first so you can see the full system at work.

### 3 Write the subscriber node

Return to `ros2_ws/src/cpp_pubsub/src` to create the next node. Enter the following code in your terminal:

**Linux**

```bash
wget -O subscriber_member_function.cpp https://raw.githubusercontent.com/ros2/examples/iron/rclcpp/topics/minimal_subscriber/member_function.cpp
```

**macOS**

```bash
wget -O subscriber_member_function.cpp https://raw.githubusercontent.com/ros2/examples/iron/rclcpp/topics/minimal_subscriber/member_function.cpp
```

**Windows**

In a Windows command line prompt:

```bash
curl -sk https://raw.githubusercontent.com/ros2/examples/iron/rclcpp/topics/minimal_subscriber/member_function.cpp -o subscriber_member_function.cpp
```

Or in powershell:

```bash
curl https://raw.githubusercontent.com/ros2/examples/iron/rclcpp/topics/minimal_subscriber/member_function.cpp -o subscriber_member_function.cpp
```
Entering `ls` in the console will now return:

```
publisher_member_function.cpp  subscriber_member_function.cpp
```

Open the `subscriber_member_function.cpp` with your text editor.

```cpp
#include <memory>
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"
using std::placeholders::_1;

class MinimalSubscriber : public rclcpp::Node
{
    public:
        MinimalSubscriber()
            : Node("minimal_subscriber")
        {
            subscription_ = this->create_subscription<std_msgs::msg::String>(
                "topic", 10, std::bind(&MinimalSubscriber::topic_callback,
                this, _1));
        }

    private:
        void topic_callback(const std_msgs::msg::String & msg) const
        {
            RCLCPP_INFO(this->get_logger(), "I heard: '\%s\'", msg.data.c_str());
        }

    rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;}

int main(int argc, char * argv[])
{
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<MinimalSubscriber>());
    rclcpp::shutdown();
    return 0;
}
```

### 3.1 Examine the code

The subscriber node’s code is nearly identical to the publisher’s. Now the node is named `minimal_subscriber`, and the constructor uses the node’s `create_subscription` class to execute the callback.

There is no timer because the subscriber simply responds whenever data is published to the topic `topic`.

```cpp
public:
    MinimalSubscriber()
        : Node("minimal_subscriber")
    {
        subscription_ = this->create_subscription<std_msgs::msg::String>(
            "topic", 10, std::bind(&MinimalSubscriber::topic_callback, this, _1));
    }
```
Recall from the topic tutorial that the topic name and message type used by the publisher and subscriber must match to allow them to communicate.

The topic_callback function receives the string message data published over the topic, and simply writes it to the console using the RCLCPP_INFO macro.

The only field declaration in this class is the subscription.

```cpp
private:
    void topic_callback(const std_msgs::msg::String & msg) const
    {
        RCLCPP_INFO(this->get_logger(), "I heard: '\%s'", msg.data.c_str());
    }
    rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
```

The main function is exactly the same, except now it spins the MinimalSubscriber node. For the publisher node, spinning meant starting the timer, but for the subscriber it simply means preparing to receive messages whenever they come.

Since this node has the same dependencies as the publisher node, there's nothing new to add to package.xml.

### 3.2 CMakeLists.txt

Reopen CMakeLists.txt and add the executable and target for the subscriber node below the publisher's entries.

```cmake
add_executable(listener src/subscriber_member_function.cpp)
ament_target_dependencies(listener rclcpp std_msgs)
install(TARGETS talker listener DESTINATION lib/${PROJECT_NAME})
```

Make sure to save the file, and then your pub/sub system should be ready.

### 4 Build and run

You likely already have the rclcpp and std_msgs packages installed as part of your ROS 2 system. It's good practice to run rosdep in the root of your workspace (ros2_ws) to check for missing dependencies before building:

**Linux**

```bash
rosdep install -i --from-path src --rosdistro iron -y
```

**macOS**

rosdep only runs on Linux, so you can skip ahead to next step.

**Windows**

rosdep only runs on Linux, so you can skip ahead to next step.

Still in the root of your workspace, ros2_ws, build your new package:

**Linux**

```bash
```
colcon build --packages-select cpp_pubsub

macOS

colcon build --packages-select cpp_pubsub

Windows

colcon build --merge-install --packages-select cpp_pubsub

Open a new terminal, navigate to `ros2_ws`, and source the setup files:

Linux

`. install/setup.bash`

macOS

`. install/setup.bash`

Windows

call install/setup.bat

Now run the talker node:

`ros2 run cpp_pubsub talker`

The terminal should start publishing info messages every 0.5 seconds, like so:

```
[INFO] [minimal_publisher]: Publishing: "Hello World: 0"
[INFO] [minimal_publisher]: Publishing: "Hello World: 1"
[INFO] [minimal_publisher]: Publishing: "Hello World: 2"
[INFO] [minimal_publisher]: Publishing: "Hello World: 3"
[INFO] [minimal_publisher]: Publishing: "Hello World: 4"
```

Open another terminal, source the setup files from inside `ros2_ws` again, and then start the listener node:

`ros2 run cpp_pubsub listener`

The listener will start printing messages to the console, starting at whatever message count the publisher is on at that time, like so:

```
[INFO] [minimal_subscriber]: I heard: "Hello World: 10"
[INFO] [minimal_subscriber]: I heard: "Hello World: 11"
[INFO] [minimal_subscriber]: I heard: "Hello World: 12"
[INFO] [minimal_subscriber]: I heard: "Hello World: 13"
[INFO] [minimal_subscriber]: I heard: "Hello World: 14"
```

Enter Ctrl+C in each terminal to stop the nodes from spinning.
Summary

You created two nodes to publish and subscribe to data over a topic. Before compiling and running them, you added their dependencies and executables to the package configuration files.

Next steps

Next you’ll create another simple ROS 2 package using the service/client model. Again, you can choose to write it in either C++ or Python.

Related content

There are several ways you could write a publisher and subscriber in C++; check out the minimal_publisher and minimal_subscriber packages in the ros2/examples repo.

Writing a simple publisher and subscriber (Python)

Goal: Create and run a publisher and subscriber node using Python.

Tutorial level: Beginner

Time: 20 minutes
Background

In this tutorial, you will create nodes that pass information in the form of string messages to each other over a topic. The example used here is a simple “talker” and “listener” system; one node publishes data and the other subscribes to the topic so it can receive that data.

The code used in these examples can be found here.

Prerequisites

In previous tutorials, you learned how to create a workspace and create a package.

A basic understanding of Python is recommended, but not entirely necessary.

Tasks

1 Create a package

Open a new terminal and source your ROS 2 installation so that ros2 commands will work.

Navigate into the ros2_ws directory created in a previous tutorial.

Recall that packages should be created in the src directory, not the root of the workspace. So, navigate into ros2_ws/src, and run the package creation command:

```
ros2 pkg create --build-type ament_python py_pubsub
```

Your terminal will return a message verifying the creation of your package py_pubsub and all its necessary files and folders.

2 Write the publisher node

Navigate into ros2_ws/src/py_pubsub/py_pubsub. Recall that this directory is a Python package with the same name as the ROS 2 package it’s nested in.

Download the example talker code by entering the following command:

Linux

```
wget https://raw.githubusercontent.com/ros2/examples/iron/rclpy/topics/minimal_publisher/examples_rclpy_minimal_publisher/publisher_member_function.py
```

macOS

```
wget https://raw.githubusercontent.com/ros2/examples/iron/rclpy/topics/minimal_publisher/examples_rclpy_minimal_publisher/publisher_member_function.py
```

Windows

In a Windows command line prompt:

```
curl -sk https://raw.githubusercontent.com/ros2/examples/iron/rclpy/topics/minimal_publisher/examples_rclpy_minimal_publisher/publisher_member_function.py -o publisher_member_function.py
```
Or in powershell:

```
curl https://raw.githubusercontent.com/ros2/examples/iron/rclpy/topics/minimal_publisher/examples_rclpy_minimal_publisher/publisher_member_function.py -o publisher_member_function.py
```

Now there will be a new file named `publisher_member_function.py` adjacent to `__init__.py`.

Open the file using your preferred text editor.

```python
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalPublisher(Node):
    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'topic', 10)
        timer_period = 0.5  # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.i = 0

    def timer_callback(self):
        msg = String()
        msg.data = 'Hello World: %d' % self.i
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)
        self.i += 1

def main(args=None):
    rclpy.init(args=args)

    minimal_publisher = MinimalPublisher()

    rclpy.spin(minimal_publisher)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_publisher.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```
2.1 Examine the code

The first lines of code after the comments import rclpy so its Node class can be used.

```python
import rclpy
from rclpy.node import Node
```

The next statement imports the built-in string message type that the node uses to structure the data that it passes on the topic.

```python
from std_msgs.msg import String
```

These lines represent the node’s dependencies. Recall that dependencies have to be added to package.xml, which you’ll do in the next section.

Next, the MinimalPublisher class is created, which inherits from (or is a subclass of) Node.

```python
class MinimalPublisher(Node):
```

Following is the definition of the class’s constructor. super().__init__ calls the Node class’s constructor and gives it your node name, in this case minimal_publisher.

create_publisher declares that the node publishes messages of type String (imported from the std_msgs.msg module), over a topic named topic, and that the “queue size” is 10. Queue size is a required QoS (quality of service) setting that limits the amount of queued messages if a subscriber is not receiving them fast enough.

Next, a timer is created with a callback to execute every 0.5 seconds. self.i is a counter used in the callback.

```python
def __init__(self):
    super().__init__('minimal_publisher')
    self.publisher_ = self.create_publisher(String, 'topic', 10)
    timer_period = 0.5  # seconds
    self.timer = self.create_timer(timer_period, self.timer_callback)
    self.i = 0
```

timer_callback creates a message with the counter value appended, and publishes it to the console with get_logger().info.

```python
def timer_callback(self):
    msg = String()
    msg.data = 'Hello World: %d' % self.i
    self.publisher_.publish(msg)
    self.get_logger().info('Publishing: %s' % msg.data)
    self.i += 1
```

Lastly, the main function is defined.

```python
def main(args=None):
    rclpy.init(args=args)

    minimal_publisher = MinimalPublisher()

    rclpy.spin(minimal_publisher)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically)
```

(continues on next page)
minimal_publisher.destroy_node()
rclpy.shutdown()

First the rclpy library is initialized, then the node is created, and then it “spins” the node so its callbacks are called.

## 2.2 Add dependencies

Navigate one level back to the `ros2_ws/src/py_pubsub` directory, where the `setup.py`, `setup.cfg`, and `package.xml` files have been created for you.

Open `package.xml` with your text editor.

As mentioned in the previous tutorial, make sure to fill in the `<description>`, `<maintainer>` and `<license>` tags:

```
<description>Examples of minimal publisher/subscriber using rclpy</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache License 2.0</license>
```

After the lines above, add the following dependencies corresponding to your node’s import statements:

```
<exec_depend>rclpy</exec_depend>
<exec_depend>std_msgs</exec_depend>
```

This declares the package needs rclpy and std_msgs when its code is executed.

Make sure to save the file.

## 2.3 Add an entry point

Open the `setup.py` file. Again, match the `maintainer`, `maintainer_email`, `description` and `license` fields to your `package.xml`:

```
maintainer='YourName',
maintainer_email='you@email.com',
description='Examples of minimal publisher/subscriber using rclpy',
license='Apache License 2.0',
```

Add the following line within the `console_scripts` brackets of the `entry_points` field:

```
entry_points={
    'console_scripts': [
        'talker = py_pubsub.publisher_member_function:main',
    ],
},
```

Don’t forget to save.
2.4 Check setup.cfg

The contents of the setup.cfg file should be correctly populated automatically, like so:

```bash
[develop]
script_dir=$base/lib/py_pubsub
[install]
install_scripts=$base/lib/py_pubsub
```

This is simply telling setuptools to put your executables in lib, because ros2 run will look for them there.

You could build your package now, source the local setup files, and run it, but let's create the subscriber node first so you can see the full system at work.

3 Write the subscriber node

Return to ros2_ws/src/py_pubsub/py_pubsub to create the next node. Enter the following code in your terminal:

Linux

```bash
wget https://raw.githubusercontent.com/ros2/examples/iron/rclpy/topics/minimal_
˓→subscriber/examples_rclpy_minimal_subscriber/subscriber_member_function.py
```

macOS

```bash
wget https://raw.githubusercontent.com/ros2/examples/iron/rclpy/topics/minimal_
˓→subscriber/examples_rclpy_minimal_subscriber/subscriber_member_function.py
```

Windows

In a Windows command line prompt:

```bash
curl -sk https://raw.githubusercontent.com/ros2/examples/iron/rclpy/topics/minimal_
˓→subscriber/examples_rclpy_minimal_subscriber/subscriber_member_function.py -o␣
˓→subscriber_member_function.py
```

Or in powershell:

```bash
curl https://raw.githubusercontent.com/ros2/examples/iron/rclpy/topics/minimal_
˓→subscriber/examples_rclpy_minimal_subscriber/subscriber_member_function.py -o␣
˓→subscriber_member_function.py
```

Now the directory should have these files:

```bash
__init__.py publisher_member_function.py subscriber_member_function.py
```
3.1 Examine the code

Open the subscriber_member_function.py with your text editor.

```python
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalSubscriber(Node):
    def __init__(self):
        super().__init__('minimal_subscriber')
        self.subscription = self.create_subscription(String, 'topic',
                                                    self.listener_callback,
                                                    10)
        # prevent unused variable warning

    def listener_callback(self, msg):
        self.get_logger().info('I heard: "%s" % msg.data')

def main(args=None):
    rclpy.init(args=args)

    minimal_subscriber = MinimalSubscriber()

    rclpy.spin(minimal_subscriber)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_subscriber.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

The subscriber node's code is nearly identical to the publisher's. The constructor creates a subscriber with the same arguments as the publisher. Recall from the topics tutorial that the topic name and message type used by the publisher and subscriber must match to allow them to communicate.

The subscriber's constructor and callback don't include any timer definition, because it doesn't need one. Its callback gets called as soon as it receives a message.
The callback definition simply prints an info message to the console, along with the data it received. Recall that the publisher defines `msg.data = 'Hello World: %d' % self.i`

```python
def listener_callback(self, msg):
    self.get_logger().info('I heard: "%s" % msg.data')
```

The main definition is almost exactly the same, replacing the creation and spinning of the publisher with the subscriber.

```python
minimal_subscriber = MinimalSubscriber()
rclpy.spin(minimal_subscriber)
```

Since this node has the same dependencies as the publisher, there’s nothing new to add to `package.xml`. The `setup.cfg` file can also remain untouched.

### 3.2 Add an entry point

Reopen `setup.py` and add the entry point for the subscriber node below the publisher's entry point. The `entry_points` field should now look like this:

```python
entry_points={
    'console_scripts': [
        'talker = py_pubsub.publisher_member_function:main',
        'listener = py_pubsub.subscriber_member_function:main',
    ],
},
```

Make sure to save the file, and then your pub/sub system should be ready.

### 4 Build and run

You likely already have the `rclpy` and `std_msgs` packages installed as part of your ROS 2 system. It’s good practice to run `rosdep` in the root of your workspace (`ros2_ws`) to check for missing dependencies before building:

**Linux**

```bash
rosdep install -i --from-path src --rosdistro iron -y
```

**macOS**

`rosdep` only runs on Linux, so you can skip ahead to next step.

**Windows**

`rosdep` only runs on Linux, so you can skip ahead to next step.

Still in the root of your workspace, `ros2_ws`, build your new package:

**Linux**

```bash
colcon build --packages-select py_pubsub
```

**macOS**

```bash
colcon build --packages-select py_pubsub
```
Windows

```bash
colcon build --merge-install --packages-select py_pubsub
```

Open a new terminal, navigate to `ros2_ws`, and source the setup files:

Linux

```bash
source install/setup.bash
```

macOS

```bash
. install/setup.bash
```

Windows

```bash
call install/setup.bat
```

Now run the talker node:

```bash
ros2 run py_pubsub talker
```

The terminal should start publishing info messages every 0.5 seconds, like so:

```plaintext
[INFO] [minimal_publisher]: Publishing: "Hello World: 0"
[INFO] [minimal_publisher]: Publishing: "Hello World: 1"
[INFO] [minimal_publisher]: Publishing: "Hello World: 2"
[INFO] [minimal_publisher]: Publishing: "Hello World: 3"
[INFO] [minimal_publisher]: Publishing: "Hello World: 4"
...
```

Open another terminal, source the setup files from inside `ros2_ws` again, and then start the listener node:

```bash
ros2 run py_pubsub listener
```

The listener will start printing messages to the console, starting at whatever message count the publisher is on at that time, like so:

```plaintext
[INFO] [minimal_subscriber]: I heard: "Hello World: 10"
[INFO] [minimal_subscriber]: I heard: "Hello World: 11"
[INFO] [minimal_subscriber]: I heard: "Hello World: 12"
[INFO] [minimal_subscriber]: I heard: "Hello World: 13"
[INFO] [minimal_subscriber]: I heard: "Hello World: 14"
```

Enter Ctrl+C in each terminal to stop the nodes from spinning.
Summary

You created two nodes to publish and subscribe to data over a topic. Before running them, you added their dependencies and entry points to the package configuration files.

Next steps

Next you’ll create another simple ROS 2 package using the service/client model. Again, you can choose to write it in either C++ or Python.

Related content

There are several ways you could write a publisher and subscriber in Python; check out the minimal_publisher and minimalSubscriber packages in the ros2/examples repo.

Writing a simple service and client (C++)

Goal: Create and run service and client nodes using C++.

Tutorial level: Beginner

Time: 20 minutes

Contents

- Background
- Prerequisites
- Tasks
  - 1 Create a package
  - 2 Write the service node
  - 3 Write the client node
  - 4 Build and run
- Summary
- Next steps
- Related content
Background

When nodes communicate using services, the node that sends a request for data is called the client node, and the one that responds to the request is the service node. The structure of the request and response is determined by a .srv file. The example used here is a simple integer addition system; one node requests the sum of two integers, and the other responds with the result.

Prerequisites

In previous tutorials, you learned how to create a workspace and create a package.

Tasks

1 Create a package

Open a new terminal and source your ROS 2 installation so that ros2 commands will work.

Navigate into the ros2_ws directory created in a previous tutorial.

Recall that packages should be created in the src directory, not the root of the workspace. Navigate into ros2_ws/src and create a new package:

```bash
ros2 pkg create --build-type ament_cmake cpp_srvcli --dependencies rclcpp example_interfaces
```

Your terminal will return a message verifying the creation of your package cpp_srvcli and all its necessary files and folders.

The --dependencies argument will automatically add the necessary dependency lines to package.xml and CMakeLists.txt. example_interfaces is the package that includes the .srv file you will need to structure your requests and responses:

```xml
int64 a
int64 b
---
int64 sum
```

The first two lines are the parameters of the request, and below the dashes is the response.

1.1 Update package.xml

Because you used the --dependencies option during package creation, you don’t have to manually add dependencies to package.xml or CMakeLists.txt.

As always, though, make sure to add the description, maintainer email and name, and license information to package.xml.

```xml
<description>C++ client server tutorial</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache License 2.0</license>
```
2 Write the service node

Inside the ros2_ws/src/cpp_srvcli/src directory, create a new file called add_two_ints_server.cpp and paste the following code within:

```cpp
#include "rclcpp/rclcpp.hpp"
#include "example_interfaces/srv/add_two_ints.hpp"
#include <memory>

void add(const std::shared_ptr<example_interfaces::srv::AddTwoInts::Request> request, 
         std::shared_ptr<example_interfaces::srv::AddTwoInts::Response> response) 
{
    response->sum = request->a + request->b;
    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Incoming request

    a: %ld" " b: %ld", 
    request->a, request->b);
    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "sending back response: [%ld]", (long_
    _int)response->sum);
}

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);

    std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("add_two_ints_server");

    rclcpp::Service<example_interfaces::srv::AddTwoInts>::SharedPtr service =
    node->create_service<example_interfaces::srv::AddTwoInts>("add_two_ints", &add);

    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Ready to add two ints.");

    rclcpp::spin(node);
    rclcpp::shutdown();
}
```

2.1 Examine the code

The first two `#include` statements are your package dependencies.

The `add` function adds two integers from the request and gives the sum to the response, while notifying the console of its status using logs.

```cpp
void add(const std::shared_ptr<example_interfaces::srv::AddTwoInts::Request> request, 
         std::shared_ptr<example_interfaces::srv::AddTwoInts::Response> response) 
{
    response->sum = request->a + request->b;
    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Incoming request

    a: %ld" " b: %ld", 
    request->a, request->b);
    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "sending back response: [%ld]", (long_
    _int)response->sum);
}
```

The main function accomplishes the following, line by line:
• Initializes ROS 2 C++ client library:

```cpp
rclcpp::init(argc, argv);
```

• Creates a node named `add_two_ints_server`:

```cpp
std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("add_two_ints_server →");
```

• Creates a service named `add_two_ints` for that node and automatically advertises it over the network with the `&add` method:

```cpp
rclcpp::Service<example_interfaces::srv::AddTwoInts>::SharedPtr service =
node->create_service<example_interfaces::srv::AddTwoInts>("add_two_ints", &add);
```

• Prints a log message when it’s ready:

```cpp
RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Ready to add two ints.");
```

• Spins the node, making the service available.

```cpp
rclcpp::spin(node);
```

### 2.2 Add executable

The `add_executable` macro generates an executable you can run using `ros2 run`. Add the following code block to `CMakeLists.txt` to create an executable named `server`:

```cmake
add_executable(server src/add_two_ints_server.cpp)
ament_target_dependencies(server rclcpp example_interfaces)
```

So `ros2 run` can find the executable, add the following lines to the end of the file, right before `ament_package()`:

```cmake
install(TARGETS
   server
   DESTINATION lib/${PROJECT_NAME})
```

You could build your package now, source the local setup files, and run it, but let’s create the client node first so you can see the full system at work.

### 3 Write the client node

Inside the `ros2_ws/src/cpp_srvcli/src` directory, create a new file called `add_two_ints_client.cpp` and paste the following code within:

```cpp
#include "rclcpp/rclcpp.hpp"
#include "example_interfaces/srv/add_two_ints.hpp"

#include <chrono>
#include <cstdlib>
#include <memory>
(continues on next page)"
```cpp
using namespace std::chrono_literals;

int main(int argc, char **argv)
{
  rclcpp::init(argc, argv);

  if (argc != 3) {
    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "usage: add_two_ints_client X Y");
    return 1;
  }

  std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("add_two_ints_client");
  rclcpp::Client<example_interfaces::srv::AddTwoInts>::SharedPtr client =
    node->create_client<example_interfaces::srv::AddTwoInts>("add_two_ints");

  auto request = std::make_shared<example_interfaces::srv::AddTwoInts::Request>();
  request->a = atoll(argv[1]);
  request->b = atoll(argv[2]);

  while (!client->wait_for_service(1s)) {
    if (!rclcpp::ok()) {
      RCLCPP_ERROR(rclcpp::get_logger("rclcpp"), "Interrupted while waiting for the service. Exiting.");
      return 0;
    }
    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "service not available, waiting again...");
  }

  auto result = client->async_send_request(request);
  // Wait for the result.
  if (rclcpp::spin_until_future_complete(node, result) ==
      rclcpp::FutureReturnCode::SUCCESS)
  {
    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Sum: %ld", result.get()->sum);
  } else {
    RCLCPP_ERROR(rclcpp::get_logger("rclcpp"), "Failed to call service add_two_ints");
  }

  rclcpp::shutdown();
  return 0;
}
```
3.1 Examine the code

Similar to the service node, the following lines of code create the node and then create the client for that node:

```cpp
std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("add_two_ints_client");
rclcpp::Client<example_interfaces::srv::AddTwoInts>::SharedPtr client = node->create_client<example_interfaces::srv::AddTwoInts>("add_two_ints");
```

Next, the request is created. Its structure is defined by the .srv file mentioned earlier.

```cpp
auto request = std::make_shared<example_interfaces::srv::AddTwoInts::Request>();
request->a = atoll(argv[1]);
request->b = atoll(argv[2]);
```

The `while` loop gives the client 1 second to search for service nodes in the network. If it can't find any, it will continue waiting.

```cpp
RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "service not available, waiting again...");
```

If the client is canceled (e.g. by you entering `Ctrl+C` into the terminal), it will return an error log message stating it was interrupted.

```cpp
RCLCPP_ERROR(rclcpp::get_logger("rclcpp"), "Interrupted while waiting for the service. Exiting.");
```

Then the client sends its request, and the node spins until it receives its response, or fails.

3.2 Add executable

Return to `CMakeLists.txt` to add the executable and target for the new node. After removing some unnecessary boilerplate from the automatically generated file, your `CMakeLists.txt` should look like this:

```cmake
cmake_minimum_required(VERSION 3.5)
project(cpp_srvcli)

find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(example_interfaces REQUIRED)

add_executable(server src/add_two_ints_server.cpp)
ament_target_dependencies(server rclcpp example_interfaces)

add_executable(client src/add_two_ints_client.cpp)
ament_target_dependencies(client rclcpp example_interfaces)

install(TARGETS
    server
    client
    DESTINATION lib/${PROJECT_NAME})

ament_package()
```
4 Build and run

It's good practice to run `rosdep` in the root of your workspace (`ros2_ws`) to check for missing dependencies before building:

**Linux**

```
rosdep install -i --from-path src --rosdistro iron -y
```

**macOS**

rosdep only runs on Linux, so you can skip ahead to next step.

**Windows**

rosdep only runs on Linux, so you can skip ahead to next step.

Navigate back to the root of your workspace, `ros2_ws`, and build your new package:

**Linux**

```
colcon build --packages-select cpp_srvcli
```

**macOS**

```
colcon build --packages-select cpp_srvcli
```

**Windows**

```
colcon build --merge-install --packages-select cpp_srvcli
```

Open a new terminal, navigate to `ros2_ws`, and source the setup files:

**Linux**

```
source install/setup.bash
```

**macOS**

```
./install/setup.bash
```

**Windows**

```
call install/setup.bat
```

Now run the service node:

```
ros2 run cpp_srvcli server
```

The terminal should return the following message, and then wait:

```
[INFO] [rclcpp]: Ready to add two ints.
```

Open another terminal, source the setup files from inside `ros2_ws` again. Start the client node, followed by any two integers separated by a space:

```
ros2 run cpp_srvcli client 2 3
```

If you chose 2 and 3, for example, the client would receive a response like this:
Return to the terminal where your service node is running. You will see that it published log messages when it received the request and the data it received, and the response it sent back:

```
[INFO] [rclcpp]: Sum: 5
```

Enter Ctrl+C in the server terminal to stop the node from spinning.

**Summary**

You created two nodes to request and respond to data over a service. You added their dependencies and executables to the package configuration files so that you could build and run them, and see a service/client system at work.

**Next steps**

In the last few tutorials you've been utilizing interfaces to pass data across topics and services. Next, you’ll learn how to create custom interfaces.

**Related content**

- There are several ways you could write a service and client in C++; check out the `minimal_service` and `minimal_client` packages in the `ros2/examples` repo.

**Writing a simple service and client (Python)**

**Goal:** Create and run service and client nodes using Python.

**Tutorial level:** Beginner

**Time:** 20 minutes

**Contents**

- Background
- Prerequisites
- Tasks
  - 1 Create a package
  - 2 Write the service node
  - 3 Write the client node
  - 4 Build and run
- Summary
- Next steps
Background

When nodes communicate using services, the node that sends a request for data is called the client node, and the one that responds to the request is the service node. The structure of the request and response is determined by a .srv file.

The example used here is a simple integer addition system; one node requests the sum of two integers, and the other responds with the result.

Prerequisites

In previous tutorials, you learned how to create a workspace and create a package.

Tasks

1 Create a package

Open a new terminal and source your ROS 2 installation so that ros2 commands will work.

Navigate into the ros2_ws directory created in a previous tutorial.

Recall that packages should be created in the src directory, not the root of the workspace. Navigate into ros2_ws/src and create a new package:

```
ros2 pkg create --build-type ament_python py_srvcli --dependencies rclpy example_interfaces
```

Your terminal will return a message verifying the creation of your package py_srvcli and all its necessary files and folders.

The --dependencies argument will automatically add the necessary dependency lines to package.xml. example_interfaces is the package that includes the .srv file you will need to structure your requests and responses:

```
int64 a
int64 b
---
int64 sum
```

The first two lines are the parameters of the request, and below the dashes is the response.

1.1 Update package.xml

Because you used the --dependencies option during package creation, you don’t have to manually add dependencies to package.xml.

As always, though, make sure to add the description, maintainer email and name, and license information to package.xml.

```
<description>Python client server tutorial</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache License 2.0</license>
```
1.2 Update setup.py

Add the same information to the setup.py file for the maintainer, maintainer_email, description and license fields:

```
maintainer='Your Name',
maintainer_email='you@email.com',
description='Python client server tutorial',
license='Apache License 2.0',
```

2 Write the service node

Inside the ros2_ws/src/py_srvcli/py_srvcli directory, create a new file called service_member_function.py and paste the following code within:

```
from example_interfaces.srv import AddTwoInts

import rclpy
from rclpy.node import Node

class MinimalService(Node):

    def __init__(self):
        super().__init__('minimal_service')
        self.srv = self.create_service(AddTwoInts, 'add_two_ints', self.add_two_ints_callback)

    def add_two_ints_callback(self, request, response):
        response.sum = request.a + request.b
        self.get_logger().info('Incoming request

    return response

def main():
    rclpy.init()

    minimal_service = MinimalService()

    rclpy.spin(minimal_service)

    rclpy.shutdown()

if __name__ == '__main__':
    main()
```
2.1 Examine the code

The first import statement imports the AddTwoInts service type from the example_interfaces package. The following import statement imports the ROS 2 Python client library, and specifically the Node class.

```python
from example_interfaces.srv import AddTwoInts
import rclpy
from rclpy.node import Node
```

The MinimalService class constructor initializes the node with the name minimal_service. Then, it creates a service and defines the type, name, and callback.

```python
def __init__(self):
    super().__init__('minimal_service')
    self.srv = self.create_service(AddTwoInts, 'add_two_ints', self.add_two_ints_callback)
```

The definition of the service callback receives the request data, sums it, and returns the sum as a response.

```python
def add_two_ints_callback(self, request, response):
    response.sum = request.a + request.b
    self.get_logger().info('Incoming request
a: %d b: %d % (request.a, request.b))

    return response
```

Finally, the main class initializes the ROS 2 Python client library, instantiates the MinimalService class to create the service node and spins the node to handle callbacks.

2.2 Add an entry point

To allow the ros2 run command to run your node, you must add the entry point to setup.py (located in the ros2_ws/src/py_srvcli directory).

Add the following line between the 'console_scripts': brackets:

```python
'service = py_srvcli.service_member_function:main',
```

3 Write the client node

Inside the ros2_ws/src/py_srvcli/py_srvcli directory, create a new file called client_member_function.py and paste the following code within:

```python
import sys

from example_interfaces.srv import AddTwoInts
import rclpy
from rclpy.node import Node

class MinimalClientAsync(Node):
```

(continues on next page)
def __init__(self):
    super().__init__('minimal_client_async')
    self.cli = self.create_client(AddTwoInts, 'add_two_ints')
    while not self.cli.wait_for_service(timeout_sec=1.0):
        self.get_logger().info('service not available, waiting again...')
    self.req = AddTwoInts.Request()

def send_request(self, a, b):
    self.req.a = a
    self.req.b = b
    self.future = self.cli.call_async(self.req)
    rclpy.spin_until_future_complete(self, self.future)
    return self.future.result()

def main():
    rclpy.init()

    minimal_client = MinimalClientAsync()
    response = minimal_client.send_request(int(sys.argv[1]), int(sys.argv[2]))
    minimal_client.get_logger().info('Result of add_two_ints: for %d + %d = %d' %
                                     (int(sys.argv[1]), int(sys.argv[2]), response.sum))
    minimal_client.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
3.2 Add an entry point

Like the service node, you also have to add an entry point to be able to run the client node.

The `entry_points` field of your `setup.py` file should look like this:

```python
entry_points={
    'console_scripts': [
        'service = py_srvcli.service_member_function:main',
        'client = py_srvcli.client_member_function:main',
    ],
},
```

4 Build and run

It's good practice to run `rosdep` in the root of your workspace (`ros2_ws`) to check for missing dependencies before building:

Linux

```bash
rosdep install -i --from-path src --rosdistro iron -y
```

macOS

rosdep only runs on Linux, so you can skip ahead to next step.

Windows

rosdep only runs on Linux, so you can skip ahead to next step.

Navigate back to the root of your workspace, `ros2_ws`, and build your new package:

```bash
colcon build --packages-select py_srvcli
```

Open a new terminal, navigate to `ros2_ws`, and source the setup files:

Linux

```bash
source install/setup.bash
```

macOS

```bash
. install/setup.bash
```

Windows

```bash
call install/setup.bat
```

Now run the service node:

```bash
ros2 run py_srvcli service
```

The node will wait for the client’s request.

Open another terminal and source the setup files from inside `ros2_ws` again. Start the client node, followed by any two integers separated by a space:
ros2 run py_srvcli client 2 3

If you chose 2 and 3, for example, the client would receive a response like this:

```
[INFO] [minimal_client_async]: Result of add_two_ints: for 2 + 3 = 5
```

Return to the terminal where your service node is running. You will see that it published log messages when it received the request:

```
[INFO] [minimal_service]: Incoming request
  a: 2 b: 3
```

Enter Ctrl+C in the server terminal to stop the node from spinning.

**Summary**

You created two nodes to request and respond to data over a service. You added their dependencies and executables to the package configuration files so that you could build and run them, allowing you to see a service/client system at work.

**Next steps**

In the last few tutorials you’ve been utilizing interfaces to pass data across topics and services. Next, you’ll learn how to create custom interfaces.

**Related content**

- There are several ways you could write a service and client in Python; check out the `minimal_client` and `minimal_service` packages in the `ros2/examples` repo.
- In this tutorial, you used the `call_async()` API in your client node to call the service. There is another service call API available for Python called synchronous calls. We do not recommend using synchronous calls, but if you’d like to learn more about them, read the guide to *Synchronous vs. asynchronous clients*.

**Creating custom msg and srv files**

**Goal:** Define custom interface files (.msg and .srv) and use them with Python and C++ nodes.

**Tutorial level:** Beginner

**Time:** 20 minutes
Background

In previous tutorials you utilized message and service interfaces to learn about topics, services, and simple publisher/subscriber (C++/Python) and service/client (C++/Python) nodes. The interfaces you used were predefined in those cases.

While it’s good practice to use predefined interface definitions, you will probably need to define your own messages and services sometimes as well. This tutorial will introduce you to the simplest method of creating custom interface definitions.

Prerequisites

You should have a ROS 2 workspace.

This tutorial also uses the packages created in the publisher/subscriber (C++ and Python) and service/client (C++ and Python) tutorials to try out the new custom messages.

Tasks

1 Create a new package

For this tutorial you will be creating custom .msg and .srv files in their own package, and then utilizing them in a separate package. Both packages should be in the same workspace.

Since we will use the pub/sub and service/client packages created in earlier tutorials, make sure you are in the same workspace as those packages (ros2_ws/src), and then run the following command to create a new package:

```bash
ros2 pkg create --build-type ament_cmake tutorial_interfaces
```

`tutorial_interfaces` is the name of the new package. Note that it is, and can only be, a CMake package, but this doesn’t restrict in which type of packages you can use your messages and services. You can create your own custom interfaces in a CMake package, and then use it in a C++ or Python node, which will be covered in the last section.

The .msg and .srv files are required to be placed in directories called msg and srv respectively. Create the directories in `ros2_ws/src/tutorial_interfaces`:

```bash
mkdir msg srv
```
2 Create custom definitions

2.1 msg definition

In the `tutorial_interfaces/msg` directory you just created, make a new file called `Num.msg` with one line of code declaring its data structure:

```c
int64 num
```

This is a custom message that transfers a single 64-bit integer called `num`.

Also in the `tutorial_interfaces/msg` directory you just created, make a new file called `Sphere.msg` with the following content:

```c
go/geometry_msgs/Point center
float64 radius
```

This custom message uses a message from another message package (`geometry_msgs/Point` in this case).

2.2 srv definition

Back in the `tutorial_interfaces/srv` directory you just created, make a new file called `AddThreeInts.srv` with the following request and response structure:

```c
int64 a
int64 b
int64 c
---
int64 sum
```

This is your custom service that requests three integers named `a`, `b`, and `c`, and responds with an integer called `sum`.

3 CMakeLists.txt

To convert the interfaces you defined into language-specific code (like C++ and Python) so that they can be used in those languages, add the following lines to `CMakeLists.txt`:

```c
find_package(geometry_msgs REQUIRED)
find_package(rosidl_default_generators REQUIRED)
rosidl_generate_interfaces(${PROJECT_NAME}
    "msg/Num.msg"
    "msg/Sphere.msg"
    "srv/AddThreeInts.srv"
    DEPENDENCIES geometry_msgs # Add packages that above messages depend on, in this case...
    )
```

Note: The first argument (library name) in the rosidl_generate_interfaces must match `${PROJECT_NAME}` (see https://github.com/ros2/rosidl/issues/441#issuecomment-591025515).
4 package.xml

Because the interfaces rely on `rosidl_default_generators` for generating language-specific code, you need to declare a build tool dependency on it. `rosidl_default_runtime` is a runtime or execution-stage dependency, needed to be able to use the interfaces later. The `rosidl_interface_packages` is the name of the dependency group that your package, `tutorial_interfaces`, should be associated with, declared using the `<member_of_group>` tag.

Add the following lines within the `<package>` element of `package.xml`:

```xml
<depend>geometry_msgs</depend>
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

5 Build the tutorial_interfaces package

Now that all the parts of your custom interfaces package are in place, you can build the package. In the root of your workspace (~/ros2_ws), run the following command:

Linux

```
colon build --packages-select tutorial_interfaces
```

macOS

```
colon build --packages-select tutorial_interfaces
```

Windows

```
colon build --merge-install --packages-select tutorial_interfaces
```

Now the interfaces will be discoverable by other ROS 2 packages.

6 Confirm msg and srv creation

In a new terminal, run the following command from within your workspace (ros2_ws) to source it:

Linux

```
source install/setup.bash
```

macOS

```
. install/setup.bash
```

Windows

```
call install/setup.bat
```

Now you can confirm that your interface creation worked by using the `ros2 interface show` command:

```
ros2 interface show tutorial_interfaces/msg/Num
```

should return:
And

```bash
ros2 interface show tutorial_interfaces/msg/Sphere
```

should return:

```markdown
geometry_msgs/Point center
  float64 x
  float64 y
  float64 z
float64 radius
```

And

```bash
ros2 interface show tutorial_interfaces/srv/AddThreeInts
```

should return:

```markdown
int64 a
int64 b
int64 c
---
int64 sum
```

## 7 Test the new interfaces

For this step you can use the packages you created in previous tutorials. A few simple modifications to the nodes, CMakeLists.txt and package.xml files will allow you to use your new interfaces.

### 7.1 Testing `Num.msg` with pub/sub

With a few modifications to the publisher/subscriber package created in a previous tutorial (C++ or Python), you can see `Num.msg` in action. Since you’ll be changing the standard string msg to a numerical one, the output will be slightly different.

**Publisher**

C++

```cpp
#include <chrono>
#include <memory>
#include "rclcpp/rclcpp.hpp"
#include "tutorial_interfaces/msg/num.hpp"  // CHANGE

using namespace std::chrono_literals;

class MinimalPublisher : public rclcpp::Node {
{
```
```cpp
public:
    MinimalPublisher()
    : Node("minimal_publisher"), count_(0)
    {
        publisher_ = this->create_publisher<tutorial_interfaces::msg::Num>("topic", 10); // CHANGE
        timer_ = this->create_wall_timer(500ms, std::bind(&MinimalPublisher::timer_callback, this));
    }
private:
    void timer_callback()
    {
        auto message = tutorial_interfaces::msg::Num(); // CHANGE
        message.num = this->count_++;
        RCLCPP_INFO_STREAM(this->get_logger(), "Publishing: " << message.num << "!"); // CHANGE
        publisher_->publish(message);
    }

    rclcpp::TimerBase::SharedPtr timer_;
    rclcpp::Publisher<tutorial_interfaces::msg::Num>::SharedPtr publisher_; // CHANGE
    size_t count_;}

int main(int argc, char * argv[])
{
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<MinimalPublisher>());
    rclcpp::shutdown();
    return 0;
}
```

Python

```python
import rclpy
from rclpy.node import Node

from tutorial_interfaces.msg import Num # CHANGE

class MinimalPublisher(Node):
    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(Num, 'topic', 10)  # CHANGE
        self.timer_period = 0.5
        self.timer = self.create_timer(self.timer_period, self.timer_callback)
        self.i = 0
```

(continues on next page)
def timer_callback(self):
    msg = Num()  # CHANGE
    msg.num = self.i  # CHANGE
    self.publisher_.publish(msg)
    self.get_logger().info('Publishing: "%d" % msg.num')  # CHANGE
    self.i += 1

def main(args=None):
    rclpy.init(args=args)

    minimal_publisher = MinimalPublisher()

    rclpy.spin(minimal_publisher)
    minimal_publisher.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
int main(int argc, char * argv[]) {
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<MinimalSubscriber>())
    rclcpp::shutdown();
    return 0;
}

Python

import rclpy
from rclpy.node import Node

from tutorial_interfaces.msg import Num # CHANGE

class MinimalSubscriber(Node):

    def __init__(self):
        super().__init__('minimal_subscriber')
        self.subscription = self.create_subscription(Num,
                                                    'topic',
                                                    self.listener_callback,
                                                    10)
        self.subscription

    def listener_callback(self, msg):
        self.get_logger().info('I heard: %d' % msg.num) # CHANGE

def main(args=None):
    rclpy.init(args=args)
    minimal_subscriber = MinimalSubscriber()
    rclpy.spin(minimal_subscriber)
    minimal_subscriber.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()

CMakeLists.txt

Add the following lines (C++ only):

#...
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(tutorial_interfaces REQUIRED) # CHANGE

add_executable(talker src/publisher_member_function.cpp)
ament_target_dependencies(talker rclcpp tutorial_interfaces) # CHANGE

add_executable(listener src/subscriber_member_function.cpp)
ament_target_dependencies(listener rclcpp tutorial_interfaces) # CHANGE

install(TARGETS
talker
listener
DESTINATION lib/${PROJECT_NAME})

ament_package()

package.xml
Add the following line:
C++
<depend>tutorial_interfaces</depend>

Python
<exec_depend>tutorial_interfaces</exec_depend>

After making the above edits and saving all the changes, build the package:
C++
On Linux/macOS:
colcon build --packages-select cpp_pubsub

On Windows:
colcon build --merge-install --packages-select cpp_pubsub

Python
On Linux/macOS:
colcon build --packages-select py_pubsub

On Windows:
colcon build --merge-install --packages-select py_pubsub

Then open two new terminals, source ros2_ws in each, and run:
C++
ros2 run cpp_pubsub talker
ros2 run cpp_pubsub listener

Python
ros2 run py_pubsub talker
ros2 run py_pubsub listener

Since `Num.msg` relays only an integer, the talker should only be publishing integer values, as opposed to the string it published previously:

```
[INFO] [minimal_publisher]: Publishing: '0'
[INFO] [minimal_publisher]: Publishing: '1'
[INFO] [minimal_publisher]: Publishing: '2'
```

### 7.2 Testing AddThreeInts.srv with service/client

With a few modifications to the service/client package created in a previous tutorial (C++ or Python), you can see AddThreeInts.srv in action. Since you’ll be changing the original two integer request srv to a three integer request srv, the output will be slightly different.

**Service**

C++

```cpp
#include "rclcpp/rclcpp.hpp"
#include "tutorial_interfaces/srv/add_three_ints.hpp"

#include <memory>

void add(const std::shared_ptr<tutorial_interfaces::srv::AddThreeInts::Request> request,
          std::shared_ptr<tutorial_interfaces::srv::AddThreeInts::Response> response)
{
    response->sum = request->a + request->b + request->c;
    RCLCPP_INFO(rclcpp::get_logger("rclcpp"),
                "Incoming request
                    a: %ld
                    b: %ld
                    c: %ld",
                request->a, request->b, request->c);
    RCLCPP_INFO(rclcpp::get_logger("rclcpp"),
                "sending back response: [%ld]", (long int)response->sum);
}

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);

    std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("add_three_ints_server");
    // CHANGE
```

(continues on next page)
`rclcpp::Service<tutorial_interfaces::srv::AddThreeInts>::SharedPtr service =`  
  // CHANGE  
  node->create_service<tutorial_interfaces::srv::AddThreeInts>("add_three_ints", &add);  
  // CHANGE  
  RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Ready to add three ints.");  
  // CHANGE  
  rclcpp::spin(node);  
  rclcpp::shutdown();

Python

```python
from tutorial_interfaces.srv import AddThreeInts

import rclpy
from rclpy.node import Node

class MinimalService(Node):
    def __init__(self):
        super().__init__('minimal_service')
        self.srv = self.create_service(AddThreeInts, 'add_three_ints', self.add_three_ints_callback)  
        # CHANGE

    def add_three_ints_callback(self, request, response):
        response.sum = request.a + request.b + request.c  
        # CHANGE  
        self.get_logger().info('Incoming request
    a: %d
    b: %d
    c: %d' % (request.a, request.b, request.c))  
        # CHANGE
        return response

def main(args=None):
    rclpy.init(args=args)

    minimal_service = MinimalService()

    rclpy.spin(minimal_service)
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

Client

C++

```cpp
#include "rclcpp/rclcpp.hpp"
```
#include "tutorial_interfaces/srv/add_three_ints.hpp"

#include <chrono>
#include <cstdlib>
#include <memory>
using namespace std::chrono_literals;

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);

    if (argc != 4) {
        // CHANGE
        RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "usage: add_three_ints_client X Y Z");
        // CHANGE
        return 1;
    }

    std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("add_three_ints_client");
    // CHANGE
    rclcpp::Client<tutorial_interfaces::srv::AddThreeInts>::SharedPtr client =
    // CHANGE
    node->create_client<tutorial_interfaces::srv::AddThreeInts>("add_three_ints");
    // CHANGE

    auto request = std::make_shared<
    tutorial_interfaces::srv::AddThreeInts::Request>();

    request->a = atoll(argv[1]);
    request->b = atoll(argv[2]);
    request->c = atoll(argv[3]);

    while (!client->wait_for_service(1s)) {
        if (!rclcpp::ok()) {
            RCLCPP_ERROR(rclcpp::get_logger("rclcpp"), "Interrupted while waiting for the service. Exiting.");
            return 0;
        }
        RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "service not available, waiting again...");
    }

    auto result = client->async_send_request(request);
    // Wait for the result.
    if (rclcpp::spin_until_future_complete(node, result) ==
    // CHANGE
    rclcpp::FutureReturnCode::SUCCESS)
    {
        RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Sum: %ld", result.get()->sum);
    }
    else {
        RCLCPP_ERROR(rclcpp::get_logger("rclcpp"), "Failed to call service add_three_ints");
        // CHANGE
    }
}
```python
from tutorial_interfaces.srv import AddThreeInts  # CHANGE
import sys
import rclpy
from rclpy.node import Node

class MinimalClientAsync(Node):
    def __init__(self):
        super().__init__('minimal_client_async')
        self.cli = self.create_client(AddThreeInts, 'add_three_ints')  # CHANGE
        while not self.cli.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('service not available, waiting again...')
        self.req = AddThreeInts.Request()
        # CHANGE
    def send_request(self):
        self.req.a = int(sys.argv[1])
        self.req.b = int(sys.argv[2])
        self.req.c = int(sys.argv[3])
        self.future = self.cli.call_async(self.req)

def main(args=None):
    rclpy.init(args=args)

    minimal_client = MinimalClientAsync()
    minimal_client.send_request()

    while rclpy.ok():
        rclpy.spin_once(minimal_client)
        if minimal_client.future.done():
            try:
                response = minimal_client.future.result()
                minimal_client.get_logger().info('Result of add_three_ints: for %d + %d + %d = %d' % (minimal_client.req.a, minimal_client.req.b, minimal_client.req.c, response.sum))  # CHANGE
            except Exception as e:
                minimal_client.get_logger().info('Service call failed %r' % (e,))
            else:
                minimal_client.get_logger().info('Result of add_three_ints: for %d + %d + %d = %d' % (minimal_client.req.a, minimal_client.req.b, minimal_client.req.c, response.sum))  # CHANGE
                break
    minimal_client.destroy_node()
```

(continues on next page)
rclpy.shutdown()

if __name__ == '__main__':
    main()

CMakeLists.txt
Add the following lines (C++ only):

#...
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(tutorial_interfaces REQUIRED)  # CHANGE
add_executable(server src/add_two_ints_server.cpp)
ament_target_dependencies(server
  rclcpp tutorial_interfaces)  # CHANGE
add_executable(client src/add_two_ints_client.cpp)
ament_target_dependencies(client
  rclcpp tutorial_interfaces)  # CHANGE
install(TARGETS
  server
  client
  DESTINATION lib/${PROJECT_NAME})
ament_package()

package.xml
Add the following line:

C++
<depend>tutorial_interfaces</depend>

Python
<exec_depend>tutorial_interfaces</exec_depend>

After making the above edits and saving all the changes, build the package:

C++
On Linux/macOS:
colcon build --packages-select cpp_srvcli

On Windows:
colcon build --merge-install --packages-select cpp_srvcli

Python
On Linux/macOS:
```
colcon build --packages-select py_srvcli
```

On Windows:
```
colcon build --merge-install --packages-select py_srvcli
```

Then open two new terminals, source `ros2_ws` in each, and run:

C++
```
ros2 run cpp_srvcli server
ros2 run cpp_srvcli client 2 3 1
```

Python
```
ros2 run py_srvcli service
ros2 run py_srvcli client 2 3 1
```

**Summary**

In this tutorial, you learned how to create custom interfaces in their own package and how to utilize those interfaces in other packages.

This tutorial only scratches the surface about defining custom interfaces. You can learn more about it in *About ROS 2 interfaces*.

**Next steps**

The *next tutorial* covers more ways to use interfaces in ROS 2.

**Implementing custom interfaces**

**Goal:** Learn more ways to implement custom interfaces in ROS 2.

**Tutorial level:** Beginner

**Time:** 15 minutes

**Contents**

- *Background*
- *Prerequisites*
- *Tasks*
  - 1 Create a package
  - 2 Create a msg file
Background

In a previous tutorial, you learned how to create custom msg and srv interfaces. While best practice is to declare interfaces in dedicated interface packages, sometimes it can be convenient to declare, create and use an interface all in one package.

Recall that interfaces can currently only be defined in CMake packages. It is possible, however, to have Python libraries and nodes in CMake packages (using ament_cmake_python), so you could define interfaces and Python nodes together in one package. We’ll use a CMake package and C++ nodes here for the sake of simplicity.

This tutorial will focus on the msg interface type, but the steps here are applicable to all interface types.

Prerequisites

We assume you’ve reviewed the basics in the Creating custom msg and srv files tutorial before working through this one.

You should have ROS 2 installed, a workspace, and an understanding of creating packages.

As always, don’t forget to source ROS 2 in every new terminal you open.

Tasks

1 Create a package

In your workspace src directory, create a package more_interfaces and make a directory within it for msg files:

```bash
ros2 pkg create --build-type ament_cmake more_interfaces
mkdir more_interfaces/msg
```

2 Create a msg file

Inside more_interfaces/msg, create a new file AddressBook.msg, and paste the following code to create a message meant to carry information about an individual:

```c
uint8 PHONE_TYPE_HOME=0
uint8 PHONE_TYPE_WORK=1
uint8 PHONE_TYPE_MOBILE=2

string first_name
```
string last_name
string phone_number
uint8 phone_type

This message is composed of these fields:

- first_name: of type string
- last_name: of type string
- phone_number: of type string
- phone_type: of type uint8, with several named constant values defined

Note that it’s possible to set default values for fields within a message definition. See Interfaces for more ways you can customize interfaces.

Next, we need to make sure that the msg file is turned into source code for C++, Python, and other languages.

### 2.1 Build a msg file

Open `package.xml` and add the following lines:

```xml
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

Note that at build time, we need `rosidl_default_generators`, while at runtime, we only need `rosidl_default_runtime`.

Open `CMakeLists.txt` and add the following lines:

Find the package that generates message code from msg/srv files:

```cmake
find_package(rosidl_default_generators REQUIRED)
```

Declare the list of messages you want to generate:

```cmake
set(msg_files
   "msg/AddressBook.msg"
)
```

By adding the .msg files manually, we make sure that CMake knows when it has to reconfigure the project after you add other .msg files.

Generate the messages:

```cmake
rosidl_generate_interfaces(${PROJECT_NAME}
   ${msg_files}
)
```

Also make sure you export the message runtime dependency:

```cmake
ament_export_dependencies(rosidl_default_runtime)
```
Now you’re ready to generate source files from your msg definition. We’ll skip the compile step for now as we’ll do it all together below in step 4.

### 2.2 (Extra) Set multiple interfaces

**Note:** You can use `set` in `CMakeLists.txt` to neatly list all of your interfaces:

```cmake
set(msg_files
    "msg/Message1.msg"
    "msg/Message2.msg"
    # etc
)
set(srv_files
    "srv/Service1.srv"
    "srv/Service2.srv"
    # etc
)
And generate all lists at once like so:

```cmake
rosidl_generate_interfaces(${PROJECT_NAME}
    ${msg_files}
    ${srv_files}
)
```

### 3 Use an interface from the same package

Now we can start writing code that uses this message.

In `more_interfaces/src` create a file called `publish_address_book.cpp` and paste the following code:

```cpp
#include <chrono>
#include <memory>
#include "rclcpp/rclcpp.hpp"
#include "more_interfaces/msg/address_book.hpp"

using namespace std::chrono_literals;

class AddressBookPublisher : public rclcpp::Node
{
public:
    AddressBookPublisher()
    : Node("address_book_publisher")
    {
        address_book_publisher_ =
            this->create_publisher<more_interfaces::msg::AddressBook>("address_book", 10);

        auto publish_msg = [this]() -> void {
```
```cpp
auto message = more_interfaces::msg::AddressBook();

message.first_name = "John";
message.last_name = "Doe";
message.phone_number = "1234567890";
message.phone_type = message.PHONE_TYPE_MOBILE;

std::cout << "Publishing Contact\nFirst:" << message.first_name << 
" Last:" << message.last_name << std::endl;

this->address_book_publisher_->publish(message);
}
timer_ = this->create_wall_timer(1s, publish_msg);
}

private:
    rclcpp::Publisher<more_interfaces::msg::AddressBook>::SharedPtr address_book_publisher_;  
    rclcpp::TimerBase::SharedPtr timer_;  

};

int main(int argc, char * argv[])
{   
rclcpp::init(argc, argv);
rclcpp::spin(std::make_shared<AddressBookPublisher>());
rclcpp::shutdown();

    return 0;
}

3.1 The code explained

Include the header of our newly created AddressBook.msg.

```cpp
#include "more_interfaces/msg/address_book.hpp"
```  

Create a node and an AddressBook publisher.

```cpp
using namespace std::chrono_literals;

class AddressBookPublisher : public rclcpp::Node
{
public:
    AddressBookPublisher()
        : Node("address_book_publisher")
    {
        address_book_publisher_ = 
            this->create_publisher<more_interfaces::msg::AddressBook>("address_book");
    
    }

    Create a callback to publish the messages periodically.
```
auto publish_msg = [this]() -> void {

Create an AddressBook message instance that we will later publish.

auto message = more_interfaces::msg::AddressBook();

Populate AddressBook fields.

message.first_name = "John";
message.last_name = "Doe";
message.phone_number = "1234567890";
message.phone_type = message.PHONE_TYPE_MOBILE;

Finally send the message periodically.

std::cout << "Publishing Contact First:" << message.first_name << " Last:" << message.last_name << std::endl;
this->address_book_publisher_->publish(message);

Create a 1 second timer to call our publish_msg function every second.

timer_ = this->create_wall_timer(1s, publish_msg);

3.2 Build the publisher

We need to create a new target for this node in the CMakeLists.txt:

find_package(rclcpp REQUIRED)

add_executable(publish_address_book src/publish_address_book.cpp)
ament_target_dependencies(publish_address_book rclcpp)

install(TARGETS
   publish_address_book
   DESTINATION lib/${PROJECT_NAME})

3.3 Link against the interface

In order to use the messages generated in the same package we need to use the following CMake code:

rosidl_get_typesupport_target(cpp_typesupport_target
   ${PROJECT_NAME} rosidl_typesupport_cpp)

target_link_libraries(publish_address_book "${cpp_typesupport_target}"
)

This finds the relevant generated C++ code from AddressBook.msg and allows your target to link against it.

You may have noticed that this step was not necessary when the interfaces being used were from a different package that was built independently. This CMake code is only required when you want to use interfaces in the same package as the one in which they are defined.
4 Try it out

Return to the root of the workspace to build the package:

Linux

```
cd ~/ros2_ws
colcon build --packages-up-to more_interfaces
```

macOS

```
cd ~/ros2_ws
colcon build --packages-up-to more_interfaces
```

Windows

```
cd /ros2_ws
colcon build --merge-install --packages-up-to more_interfaces
```

Then source the workspace and run the publisher:

Linux

```
source install/local_setup.bash
ros2 run more_interfaces publish_address_book
```

macOS

```
. install/local_setup.bash
ros2 run more_interfaces publish_address_book
```

Windows

```
call install/local_setup.bat
ros2 run more_interfaces publish_address_book
```

Or using Powershell:

```
install/local_setup.ps1
ros2 run more_interfaces publish_address_book
```

You should see the publisher relaying the msg you defined, including the values you set in publish_address_book.cpp.

To confirm the message is being published on the address_book topic, open another terminal, source the workspace, and call topic echo:

Linux

```
source install/setup.bash
ros2 topic echo /address_book
```

macOS

```
. install/setup.bash
ros2 topic echo /address_book
```

Windows

```
call install/setup.bat  
ros2 topic echo /address_book

Or using Powershell:

install/setup.ps1  
ros2 topic echo /address_book

We won’t create a subscriber in this tutorial, but you can try to write one yourself for practice (use Writing a simple publisher and subscriber (C++) to help).

5 (Extra) Use an existing interface definition

**Note:** You can use an existing interface definition in a new interface definition. For example, let’s say there is a message named Contact.msg that belongs to an existing ROS 2 package named rosidl_tutorials_msgs. Assume that its definition is identical to our custom-made AddressBook.msg interface from earlier.

In that case you could have defined AddressBook.msg (an interface in the package with your nodes) as type Contact (an interface in a separate package). You could even define AddressBook.msg as an array of type Contact, like so:

```
rosidl_tutorials_msgs/Contact[] address_book
```

To generate this message you would need to declare a dependency on Contact.msg's package, rosidl_tutorials_msgs, in package.xml:

```
<build_depend>rosidl_tutorials_msgs</build_depend>
<exec_depend>rosidl_tutorials_msgs</exec_depend>
```

And in CMakeLists.txt:

```
find_package(rosidl_tutorials_msgs REQUIRED)
rosidl_generate_interfaces(${PROJECT_NAME} ${msg_files} DEPENDENCIES rosidl_tutorials_msgs)
```

You would also need to include the header of Contact.msg in you publisher node in order to be able to add contacts to your address_book.

```
#include "rosidl_tutorials_msgs/msg/contact.hpp"
```

You could change the callback to something like this:

```
auto publish_msg = [this]() -> void {
    auto msg = std::make_shared<more_interfaces::msg::AddressBook>();
    {
        rosidl_tutorials_msgs::msg::Contact contact;
        contact.first_name = "John";
        contact.last_name = "Doe";
        contact.phone_number = "1234567890";
    }
}
```

(continues on next page)
Building and running these changes would show the msg defined as expected, as well as the array of msgs defined above.

**Summary**

In this tutorial, you tried out different field types for defining interfaces, then built an interface in the same package where it’s being used.

You also learned how to use another interface as a field type, as well as the package.xml, CMakeLists.txt, and #include statements necessary for utilizing that feature.

**Next steps**

Next you will create a simple ROS 2 package with a custom parameter that you will learn to set from a launch file. Again, you can choose to write it in either C++ or Python.

**Related content**

There are several design articles on ROS 2 interfaces and the IDL (interface definition language).
Using parameters in a class (C++)

Goal: Create and run a class with ROS parameters using C++.

Tutorial level: Beginner

Time: 20 minutes

Contents

- Background
- Prerequisites
- Tasks
  - 1 Create a package
  - 2 Write the C++ node
  - 3 Build and run
- Summary
- Next steps

Background

When making your own nodes you will sometimes need to add parameters that can be set from the launch file. This tutorial will show you how to create those parameters in a C++ class, and how to set them in a launch file.

Prerequisites

In previous tutorials, you learned how to create a workspace and create a package. You have also learned about parameters and their function in a ROS 2 system.

Tasks

1 Create a package

Open a new terminal and source your ROS 2 installation so that ros2 commands will work.

Follow these instructions to create a new workspace named ros2_ws.

Recall that packages should be created in the src directory, not the root of the workspace. Navigate into ros2_ws/src and create a new package:

```bash
ros2 pkg create --build-type ament_cmake cpp_parameters --dependencies rclcpp
```

Your terminal will return a message verifying the creation of your package cpp_parameters and all its necessary files and folders.

The --dependencies argument will automatically add the necessary dependency lines to package.xml and CMakeLists.txt.
1.1 Update package.xml

Because you used the --dependencies option during package creation, you don’t have to manually add dependencies to package.xml or CMakeLists.txt.

As always, though, make sure to add the description, maintainer email and name, and license information to package.xml.

```xml
<description>C++ parameter tutorial</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache License 2.0</license>
```

2 Write the C++ node

Inside the ros2_ws/src/cpp_parameters/src directory, create a new file called cpp_parameters_node.cpp and paste the following code within:

```cpp
#include <chrono>
#include <functional>
#include <string>
#include <rclcpp/rclcpp.hpp>
using namespace std::chrono_literals;

class MinimalParam : public rclcpp::Node {
public:
    MinimalParam() : Node("minimal_param_node")
    {
        this->declare_parameter("my_parameter", "world");

        timer_ = this->create_wall_timer(1000ms, std::bind(&MinimalParam::timer_callback, this));
    }

    void timer_callback()
    {
        std::string my_param = this->get_parameter("my_parameter").as_string();

        RCLCPP_INFO(this->get_logger(), "Hello %s!", my_param.c_str());

        std::vector<rclcpp::Parameter> all_new_parameters{rclcpp::Parameter("my_parameter", "world")};
        this->set_parameters(all_new_parameters);
    }
private:
    rclcpp::TimerBase::SharedPtr timer_;}
```

(continues on next page)
```cpp
int main(int argc, char ** argv)
{
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<MinimalParam>());
    rclcpp::shutdown();
    return 0;
}
```

### 2.1 Examine the code

The `#include` statements at the top are the package dependencies.

The next piece of code creates the class and the constructor. The first line of this constructor creates a parameter with the name `my_parameter` and a default value of `world`. The parameter type is inferred from the default value, so in this case it would be set to a string type. Next the `timer_` is initialized with a period of 1000ms, which causes the `timer_callback` function to be executed once a second.

```cpp
class MinimalParam : public rclcpp::Node
{
public:
    MinimalParam()
    : Node("minimal_param_node")
    {
        this->declare_parameter("my_parameter", "world");
        timer_ = this->create_wall_timer(1000ms, std::bind(&MinimalParam::timer_callback, this));
    }
}
```

The first line of our `timer_callback` function gets the parameter `my_parameter` from the node, and stores it in `my_param`. Next the RCLCPP_INFO function ensures the event is logged. The `set_parameters` function then sets the parameter `my_parameter` back to the default string value `world`. In the case that the user changed the parameter externally, this ensures it is always reset back to the original.

```cpp
void timer_callback()
{
    std::string my_param = this->get_parameter("my_parameter").as_string();

    RCLCPP_INFO(this->get_logger(), "Hello %s!", my_param.c_str());

    std::vector<rclcpp::Parameter> all_new_parameters{rclcpp::Parameter("my_parameter", "world")};
    this->set_parameters(all_new_parameters);
}
```

Last is the declaration of `timer_`.

```cpp
private:
    rclcpp::TimerBase::SharedPtr timer_;  
```

Following our `MinimalParam` is our `main`. Here ROS 2 is initialized, an instance of the `MinimalParam` class is constructed, and `rclcpp::spin` starts processing data from the node.
2.1.1 (Optional) Add ParameterDescriptor

Optionally, you can set a descriptor for the parameter. Descriptors allow you to specify a text description of the parameter and its constraints, like making it read-only, specifying a range, etc. For that to work, the code in the constructor has to be changed to:

```cpp
class MinimalParam : public rclcpp::Node {
public:
    MinimalParam()
    : Node("minimal_param_node")
    {
        auto param_desc = rcl_interfaces::msg::ParameterDescriptor{};
        param_desc.description = "This parameter is mine!";
        this->declare_parameter("my_parameter", "world", param_desc);
        timer_ = this->create_wall_timer(1000ms, std::bind(&MinimalParam::timer_callback, this));
    }
};
```

The rest of the code remains the same. Once you run the node, you can then run `ros2 param describe /minimal_param_node my_parameter` to see the type and description.

2.2 Add executable

Now open the CMakeLists.txt file. Below the dependency `find_package(rclcpp REQUIRED)` add the following lines of code:

```cmake
add_executable(minimal_param_node src/cpp_parameters_node.cpp)
ament_target_dependencies(minimal_param_node rclcpp)
install(TARGETS minimal_param_node DESTINATION lib/${PROJECT_NAME})
```
3 Build and run

It's good practice to run rosdep in the root of your workspace (ros2_ws) to check for missing dependencies before building:

Linux

```
rosdep install -i --from-path src --rosdistro iron -y
```

macOS

rosdep only runs on Linux, so you can skip ahead to next step.

Windows

rosdep only runs on Linux, so you can skip ahead to next step.

Navigate back to the root of your workspace, ros2_ws, and build your new package:

Linux

```
colcon build --packages-select cpp_parameters
```

macOS

```
colcon build --packages-select cpp_parameters
```

Windows

```
colcon build --merge-install --packages-select cpp_parameters
```

Open a new terminal, navigate to ros2_ws, and source the setup files:

Linux

```
source install/setup.bash
```

macOS

```
. install/setup.bash
```

Windows

```
call install/setup.bat
```

Now run the node:

```
ros2 run cpp_parameters minimal_param_node
```

The terminal should return the following message every second:

```
[INFO] [minimal_param_node]: Hello world!
```

Now you can see the default value of your parameter, but you want to be able to set it yourself. There are two ways to accomplish this.
3.1 Change via the console

This part will use the knowledge you have gained from the tutorial about parameters and apply it to the node you have just created.

Make sure the node is running:

```
ros2 run cpp_parameters minimal_param_node
```

Open another terminal, source the setup files from inside `ros2_ws` again, and enter the following line:

```
ros2 param list
```

There you will see the custom parameter `my_parameter`. To change it, simply run the following line in the console:

```
ros2 param set /minimal_param_node my_parameter earth
```

You know it went well if you got the output `Set parameter successful`. If you look at the other terminal, you should see the output change to `[INFO] [minimal_param_node]: Hello earth!`

3.2 Change via a launch file

You can also set the parameter in a launch file, but first you will need to add the launch directory. Inside the `ros2_ws/src/cpp_parameters/` directory, create a new directory called `launch`. In there, create a new file called `cpp_parameters_launch.py`

```python
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package="cpp_parameters",
            executable="minimal_param_node",
            name="custom_minimal_param_node",
            output="screen",
            emulate_tty=True,
            parameters=[
                {"my_parameter": "earth"}
            ]
        )
    ])
```

Here you can see that we set `my_parameter` to `earth` when we launch our node `minimal_param_node`. By adding the two lines below, we ensure our output is printed in our console.

```
output="screen",
emulate_tty=True,
```

Now open the `CMakeLists.txt` file. Below the lines you added earlier, add the following lines of code.

```cmake
install(
    DIRECTORY launch
)
```

(continues on next page)
Open a console and navigate to the root of your workspace, ros2_ws, and build your new package:

Linux

```
colcon build --packages-select cpp_parameters
```

macOS

```
colcon build --packages-select cpp_parameters
```

Windows

```
colcon build --merge-install --packages-select cpp_parameters
```

Then source the setup files in a new terminal:

Linux

```
source install/setup.bash
```

macOS

```
. install/setup.bash
```

Windows

```
call install/setup.bat
```

Now run the node using the launch file we have just created:

```python
ros2 launch cpp_parameters cpp_parameters_launch.py
```

The terminal should return the following message the first time:

```
[INFO] [custom_minimal_param_node]: Hello earth!
```

Further outputs should show [INFO] [minimal_param_node]: Hello world! every second.

**Summary**

You created a node with a custom parameter that can be set either from a launch file or the command line. You added the dependencies, executables, and a launch file to the package configuration files so that you could build and run them, and see the parameter in action.
Next steps

Now that you have some packages and ROS 2 systems of your own, the next tutorial will show you how to examine issues in your environment and systems in case you have problems.

Using parameters in a class (Python)

Goal: Create and run a class with ROS parameters using Python.

Tutorial level: Beginner

Time: 20 minutes

Contents

• Background
• Prerequisites
• Tasks
  – 1 Create a package
  – 2 Write the Python node
  – 3 Build and run
• Summary
• Next steps

Background

When making your own nodes you will sometimes need to add parameters that can be set from the launch file. This tutorial will show you how to create those parameters in a Python class, and how to set them in a launch file.

Prerequisites

In previous tutorials, you learned how to create a workspace and create a package. You have also learned about parameters and their function in a ROS 2 system.

Tasks

1 Create a package

Open a new terminal and source your ROS 2 installation so that ros2 commands will work.

Follow these instructions to create a new workspace named ros2_ws.

Recall that packages should be created in the src directory, not the root of the workspace. Navigate into ros2_ws/src and create a new package:
ros2 pkg create --build-type ament_python python_parameters --dependencies rclpy

Your terminal will return a message verifying the creation of your package python_parameters and all its necessary files and folders.

The --dependencies argument will automatically add the necessary dependency lines to package.xml and CMakeLists.txt.

1.1 Update package.xml

Because you used the --dependencies option during package creation, you don’t have to manually add dependencies to package.xml or CMakeLists.txt.

As always, though, make sure to add the description, maintainer email and name, and license information to package.xml.

<description>Python parameter tutorial</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache License 2.0</license>

2 Write the Python node

Inside the ros2_ws/src/python_parameters/python_parameters directory, create a new file called python_parameters_node.py and paste the following code within:

```python
import rclpy
import rclpy.node

class MinimalParam(rclpy.node.Node):
    def __init__(self):
        super().__init__('minimal_param_node')

        self.declare_parameter('my_parameter', 'world')

        self.timer = self.create_timer(1, self.timer_callback)

    def timer_callback(self):
        my_param = self.get_parameter('my_parameter').get_parameter_value().string_value

        self.get_logger().info('Hello %s! % my_param)

        my_new_param = rclpy.parameter.Parameter(
            'my_parameter',
            rclpy.Parameter.Type.STRING,
            'world'
        )

        all_new_parameters = [my_new_param]
        self.set_parameters(all_new_parameters)

    def main():
        rclpy.init()
```

(continues on next page)
2.1 Examine the code

The import statements at the top are used to import the package dependencies. The next piece of code creates the class and the constructor. The line `self.declare_parameter('my_parameter', 'world')` of the constructor creates a parameter with the name `my_parameter` and a default value of `world`. The parameter type is inferred from the default value, so in this case it would be set to a string type. Next the timer is initialized with a period of 1, which causes the `timer_callback` function to be executed once a second.

```python
class MinimalParam(rclpy.node.Node):
    def __init__(self):
        super().__init__('minimal_param_node')
        self.declare_parameter('my_parameter', 'world')
        self.timer = self.create_timer(1, self.timer_callback)
```

The first line of our `timer_callback` function gets the parameter `my_parameter` from the node, and stores it in `my_param`. Next the `get_logger` function ensures the event is logged. The `set_parameters` function then sets the parameter `my_parameter` back to the default string value `world`. In the case that the user changed the parameter externally, this ensures it is always reset back to the original.

```python
def timer_callback(self):
    my_param = self.get_parameter('my_parameter').get_parameter_value().string_value
    self.get_logger().info('Hello %s! % my_param)

    my_new_param = rclpy.parameter.Parameter('my_parameter',
                                             rclpy.Parameter.Type.STRING,
                                             'world')
    all_new_parameters = [my_new_param]
    self.set_parameters(all_new_parameters)
```

Following the `timer_callback` is our `main`. Here ROS 2 is initialized, an instance of the `MinimalParam` class is constructed, and `rclpy.spin` starts processing data from the node.

```python
def main():
    rclpy.init()
    node = MinimalParam()
    rclpy.spin(node)

if __name__ == '__main__':
    main()
```
2.1.1 (Optional) Add ParameterDescriptor

Optionally, you can set a descriptor for the parameter. Descriptors allow you to specify a text description of the parameter and its constraints, like making it read-only, specifying a range, etc. For that to work, the `__init__` code has to be changed to:

```
# ...

class MinimalParam(rclpy.node.Node):
    def __init__(self):
        super().__init__('minimal_param_node')

        from rcl_interfaces.msg import ParameterDescriptor
        my_parameter_descriptor = ParameterDescriptor(description='This parameter is mine!')

        self.declare_parameter('my_parameter', 'world', my_parameter_descriptor)

        self.timer = self.create_timer(1, self.timer_callback)
```

The rest of the code remains the same. Once you run the node, you can then run `ros2 param describe /minimal_param_node my_parameter` to see the type and description.

2.2 Add an entry point

Open the `setup.py` file. Again, match the `maintainer`, `maintainer_email`, `description` and `license` fields to your `package.xml`:

```
maintainer='YourName',
maintainer_email='you@email.com',
description='Python parameter tutorial',
license='Apache License 2.0',
```

Add the following line within the `console_scripts` brackets of the `entry_points` field:

```
entry_points={
    'console_scripts': [
        'minimal_param_node = python_parameters.python_parameters_node:main',
    ],
},
```

Don’t forget to save.
3 Build and run

It's good practice to run `rosdep` in the root of your workspace (`ros2_ws`) to check for missing dependencies before building:

Linux

```bash
rosdep install -i --from-path src --rosdistro iron -y
```

macOS

rosdep only runs on Linux, so you can skip ahead to next step.

Windows

rosdep only runs on Linux, so you can skip ahead to next step.

Navigate back to the root of your workspace, `ros2_ws`, and build your new package:

Linux

```bash
colcon build --packages-select python_parameters
```

macOS

```bash
colcon build --packages-select python_parameters
```

Windows

```bash
colcon build --merge-install --packages-select python_parameters
```

Open a new terminal, navigate to `ros2_ws`, and source the setup files:

Linux

```bash
source install/setup.bash
```

macOS

```bash
. install/setup.bash
```

Windows

```bash
call install/setup.bat
```

Now run the node:

```bash
ros2 run python_parameters minimal_param_node
```

The terminal should return the following message every second:

```
[INFO] [parameter_node]: Hello world!
```

Now you can see the default value of your parameter, but you want to be able to set it yourself. There are two ways to accomplish this.
3.1 Change via the console

This part will use the knowledge you have gained from the tutorial about parameters and apply it to the node you have just created.

Make sure the node is running:

```bash
ros2 run python_parameters minimal_param_node
```

Open another terminal, source the setup files from inside `ros2_ws` again, and enter the following line:

```bash
ros2 param list
```

There you will see the custom parameter `my_parameter`. To change it, simply run the following line in the console:

```bash
ros2 param set /minimal_param_node my_parameter earth
```

You know it went well if you get the output `Set parameter successful`. If you look at the other terminal, you should see the output change to `[INFO] [minimal_param_node]: Hello earth!`

Since the node afterwards set the parameter back to `world`, further outputs show `[INFO] [minimal_param_node]: Hello world!

3.2 Change via a launch file

You can also set parameters in a launch file, but first you will need to add a launch directory. Inside the `ros2_ws/src/python_parameters/` directory, create a new directory called `launch`. In there, create a new file called `python_parameters_launch.py`

```python
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='python_parameters',
            executable='minimal_param_node',
            name='custom_minimal_param_node',
            output='screen',
            emulate_tty=True,
            parameters=[
                {'my_parameter': 'earth'}
            ]
        )
    ])
```

Here you can see that we set `my_parameter` to `earth` when we launch our node `parameter_node`. By adding the two lines below, we ensure our output is printed in our console.

```python
output="screen",
emulate_tty=True,
```

Now open the `setup.py` file. Add the `import` statements to the top of the file, and the other new statement to the `data_files` parameter to include all launch files:
import os
from glob import glob

# ...

setup(
    # ...
    data_files=[
        # ...
        (os.path.join('share', package_name), glob('launch/*launch.[pxy][yma]*')),
    ]
)

Open a console and navigate to the root of your workspace, ros2_ws, and build your new package:

Linux

```bash
colcon build --packages-select python_parameters
```

macOS

```bash
colcon build --packages-select python_parameters
```

Windows

```bash
colcon build --merge-install --packages-select python_parameters
```

Then source the setup files in a new terminal:

Linux

```bash
source install/setup.bash
```

macOS

```bash
. install/setup.bash
```

Windows

```bash
call install/setup.bat
```

Now run the node using the launch file we have just created:

```bash
ros2 launch python_parameters python_parameters_launch.py
```

The terminal should return the following message the first time:

```
[INFO] [custom_minimal_param_node]: Hello earth!
```

Further outputs should show `[INFO] [minimal_param_node]: Hello world! every second.`
Summary

You created a node with a custom parameter that can be set either from a launch file or the command line. You added the dependencies, executables, and a launch file to the package configuration files so that you could build and run them, and see the parameter in action.

Next steps

Now that you have some packages and ROS 2 systems of your own, the next tutorial will show you how to examine issues in your environment and systems in case you have problems.

Using ros2doctor to identify issues

Goal: Identify issues in your ROS 2 setup using the ros2doctor tool.

Tutorial level: Beginner

Time: 10 minutes

Background

When your ROS 2 setup is not running as expected, you can check its settings with the ros2doctor tool. ros2doctor checks all aspects of ROS 2, including platform, version, network, environment, running systems and more, and warns you about possible errors and reasons for issues.
Prerequisites

ros2doctor is part of the ros2cli package. As long as you have ros2cli installed (which any normal install should have), you will be able to use ros2doctor.

This tutorial uses turtlesim to illustrate some of the examples.

Tasks

1 Check your setup

Let’s examine your general ROS 2 setup as a whole with ros2doctor. First, source ROS 2 in a new terminal, then enter the command:

```
ros2 doctor
```

This will conduct checks over all your setup modules and return warnings and errors.

If your ROS 2 setup is in perfect shape, you’ll see a message similar to this:

```
All <n> checks passed
```

However, it’s not unusual to have a few warnings returned. A UserWarning doesn’t mean your setup is unusable; it’s more likely just an indication that something is configured in a way that's not ideal.

If you do receive a warning, it will look something like this:

```
<path>: <line>: UserWarning: <message>
```

For example, ros2doctor will find this warning if you’re using an unstable ROS 2 distribution:

```
UserWarning: Distribution <distro> is not fully supported or tested. To get more consistent features, download a stable version at https://index.ros.org/doc/ros2/Installation/
```

If ros2doctor only finds warnings in your system, you will still receive the All <n> checks passed message.

Most checks are categorized as warnings as opposed to errors. It’s mostly up to you, the user, to determine the importance of the feedback ros2doctor returns. If it does find a rare error in your setup, indicated by UserWarning: ERROR:, the check is considered failed.

You will see a message similar to the following list of issue feedback:

```
1/3 checks failed
Failed modules: network
```

An error indicates the system is missing important settings or functions that are crucial to ROS 2. Errors should be addressed to ensure the system functions properly.
## 2 Check a system

You can also examine a running ROS 2 system to identify possible causes for issues. To see `ros2doctor` working on a running system, let's run `turtlesim`, which has nodes actively communicating with each other.

Start up the system by opening a new terminal, sourcing ROS 2, and entering the command:

```
ros2 run turtlesim turtlesim_node
```

Open another terminal and source ROS 2 to run the teleop controls:

```
ros2 run turtlesim turtle_teleop_key
```

Now run `ros2doctor` again in its own terminal. You will see the warnings and errors you had the last time you ran `ros2doctor` on your setup if you had any. Following those will be a couple new warnings relating to the system itself:

```
UserWarning: Publisher without subscriber detected on /turtle1/color_sensor.
UserWarning: Publisher without subscriber detected on /turtle1/pose.
```

It seems that the `/turtlesim` node publishes data to two topics that aren't being subscribed to, and `ros2doctor` thinks this could possibly lead to issues.

If you run commands to echo the `/color_sensor` and `/pose` topics, those warnings will disappear because the publishers will have subscribers.

You can try this by opening two new terminals while `turtlesim` is still running, sourcing ROS 2 in each, and running each of the following commands in their own terminal:

```
ros2 topic echo /turtle1/color_sensor
ros2 topic echo /turtle1/pose
```

Then run `ros2doctor` in its terminal again. The `publisher without subscriber` warnings will be gone. (Make sure to enter Ctrl+C in the terminals where you ran `echo`).

Now try exiting either the `turtlesim` window or quitting the teleop and running `ros2doctor` again. You'll see more warnings indicating `publisher without subscriber` or `subscriber without publisher` for different topics, now that one node in the system isn’t available.

In a complex system with many nodes, `ros2doctor` would be invaluable for identifying possible reasons for communication issues.

## 3 Get a full report

While `ros2doctor` will let you know warnings about your network, system, etc., running it with the `--report` argument will give you much more detail to help you analyze issues.

You might want to use `--report` if you get a warning about your network setup and want to find out exactly what part of your configuration is causing the warning.

It’s also very helpful when you need to open a support ticket to get help with ROS 2. You can copy and paste the relevant parts of your report into the ticket so the people helping you can better understand your environment and provide better assistance.

To get a full report, enter the following command in the terminal:
ros2 doctor --report

Which will return a list of information categorized into five groups:

NETWORK CONFIGURATION
...

PLATFORM INFORMATION
...

RMW MIDDLEWARE
...

ROS 2 INFORMATION
...

TOPIC LIST
...

You can crosscheck the information here against the warnings you get when running ros2 doctor. For example, if ros2doctor returned the warning (mentioned earlier) that your distribution is “not fully supported or tested”, you might take a look at the ROS 2 INFORMATION section of the report:

distribution name : <distro>
distribution type : ros2
distribution status : prerelease
release platforms : {'<platform>': ['<version>']}

Here you can see the distribution status is prerelease, which explains why it’s not fully supported.

Summary

ros2doctor will inform you of problems in your ROS 2 setup and running systems. You can get a deeper look at information behind those warnings by using the --report argument.

Keep in mind, ros2doctor is not a debug tool; it won’t help with errors in your code or on the implementation side of your system.
Related content

ros2doctor’s README will tell you more about different arguments. You might want to take a look around the ros2doctor repo as well, since it’s fairly beginner friendly and a great place to get started with contributing.

Next steps

You’ve completed the beginner level tutorials!

Creating and using plugins (C++)

Goal: Learn to create and load a simple plugin using pluginlib.

Tutorial level: Beginner

Time: 20 minutes

Contents

• Background
• Prerequisites
• Tasks
  – 1 Create the Base Class Package
  – 2 Create the Plugin Package
    * 2.1 Source code for the plugins
    * 2.2 Plugin Declaration XML
    * 2.3 CMake Plugin Declaration
  – 3 Use the Plugins
  – 4 Build and run
• Summary

Background

This tutorial is derived from http://wiki.ros.org/pluginlib and Writing and Using a Simple Plugin Tutorial.

pluginlib is a C++ library for loading and unloading plugins from within a ROS package. Plugins are dynamically loadable classes that are loaded from a runtime library (i.e. shared object, dynamically linked library). With pluginlib, one does not have to explicitly link their application against the library containing the classes – instead pluginlib can open a library containing exported classes at any point without the application having any prior awareness of the library or the header file containing the class definition. Plugins are useful for extending/modifying application behavior without needing the application source code.
Prerequisites

This tutorial assumes basic C++ knowledge and that you have pluginlib installed.

```
sudo apt-get install ros-iron-pluginlib
```

Tasks

In this tutorial, you will create two new packages, one that defines the base class, and another that provides the plugins. The base class will define a generic polygon class, and then our plugins will define specific shapes.

1 Create the Base Class Package

Create a new empty package in your `ros2_ws/src` folder with the following command:

```
ros2 pkg create --build-type ament_cmake polygon_base --dependencies pluginlib --node-name area_node
```

Open your favorite editor, edit `ros2_ws/src/polygon_base/include/polygon_base/regular_polygon.hpp`, and paste the following inside of it:

```
#ifndef POLYGON_BASE_REGULAR_POLYGON_HPP
#define POLYGON_BASE_REGULAR_POLYGON_HPP

namespace polygon_base
{
  class RegularPolygon
  {
    public:
      virtual void initialize(double side_length) = 0;
      virtual double area() = 0;
      virtual ~RegularPolygon(){}
    protected:
      RegularPolygon(){
    }
  } // namespace polygon_base

#ifdef POLYGON_BASE_REGULAR_POLYGON_HPP
#define POLYGON_BASE_REGULAR_POLYGON_HPP

#endif // POLYGON_BASE_REGULAR_POLYGON_HPP
```

This code above should be pretty self-explanatory... we're creating an abstract class called `RegularPolygon`. One thing to notice is the presence of the initialize method. With pluginlib, a constructor without parameters is required, so if any parameters to the class are needed, we use the initialize method to pass them to the object.

We need to make this header available to other classes, so open `ros2_ws/src/polygon_base/CMakeLists.txt` for editing. Add the following lines after the `ament_target_dependencies` command:

```
install(
  DIRECTORY include/
  DESTINATION include
)
```

And add this command before the `ament_package` command:
ament_export_include_directories(
   include
)

We will return to this package later to write our test node.

2 Create the Plugin Package

Now we’re going to write two non-virtual implementations of our abstract class. Create a second empty package in your ros2_ws/src folder with the following command:

```
ros2 pkg create --build-type ament_cmake polygon_plugins --dependencies polygon_base pluginlib --library-name polygon_plugins
```

2.1 Source code for the plugins

Open ros2_ws/src/polygon_plugins/src/polygon_plugins.cpp for editing, and paste the following inside of it:

```
#include <polygon_base/regular_polygon.hpp>
#include <cmath>

namespace polygon_plugins
{
    class Square : public polygon_base::RegularPolygon
    {
        public:
            void initialize(double side_length) override
            {
                side_length_ = side_length;
            }

            double area() override
            {
                return side_length_ * side_length_;
            }

        protected:
            double side_length_;}

    class Triangle : public polygon_base::RegularPolygon
    {
        public:
            void initialize(double side_length) override
            {
                side_length_ = side_length;
            }

            double area() override
            {

(continues on next page)
The implementation of the Square and Triangle classes should be fairly straightforward: save the side length, and use it to calculate the area. The only piece that is pluginlib specific is the last three lines, which invokes some magical macros that register the classes as actual plugins. Let’s go through the arguments to the PLUGINLIB_EXPORT_CLASS macro:

1. The fully-qualified type of the plugin class, in this case, polygon_plugins::Square.
2. The fully-qualified type of the base class, in this case, polygon_base::RegularPolygon.

### 2.2 Plugin Declaration XML

The steps above make it so that instances of our plugins can be created once the library they exist in is loaded, but the plugin loader still needs a way to find that library and to know what to reference within that library. To this end, we’ll also create an XML file that, along with a special export line in the package manifest, makes all the necessary information about our plugins available to the ROS toolchain.

Create ros2_ws/src/polygon_plugins/plugins.xml with the following code:

```xml
<library path="polygon_plugins">
  <class type="polygon_plugins::Square" base_class_type="polygon_base::RegularPolygon">
    <description>This is a square plugin.</description>
  </class>
  <class type="polygon_plugins::Triangle" base_class_type="polygon_base::RegularPolygon">
    <description>This is a triangle plugin.</description>
  </class>
</library>
```

A couple things to note:

1. The library tag gives the relative path to a library that contains the plugins that we want to export. In ROS 2, that is just the name of the library. In ROS 1, it contained the prefix lib or sometimes lib/lib (i.e. lib/libpolygon_plugins), but here it is simpler.
2. The class tag declares a plugin that we want to export from our library. Let’s go through its parameters:
   - type: The fully qualified type of the plugin. For us, that’s polygon_plugins::Square.
• **base_class**: The fully qualified base class type for the plugin. For us, that's `polygon_base::RegularPolygon`.

• **description**: A description of the plugin and what it does.

• **name**: There used to be a name attribute, but it is no longer required.

### 2.3 CMake Plugin Declaration

The last step is to export your plugins via `CMakeLists.txt`. This is a change from ROS 1, where the exporting was done via `package.xml`. Add the following line to your `ros2_ws/src/polygon_plugins/CMakeLists.txt` after the line reading `find_package(pluginlib REQUIRED):

```cpp
pluginlib_export_plugin_description_file(polygon_base plugins.xml)
```

The arguments to the `pluginlib_export_plugin_description_file` command are:

1. The package with the base class, i.e. `polygon_base`.
2. The relative path to the Plugin Declaration xml, i.e. `plugins.xml`.

### 3 Use the Plugins

Now it’s time to use the plugins. This can be done in any package, but here we’re going to do it in the base package. Edit `ros2_ws/src/polygon_base/src/area_node.cpp` to contain the following:

```cpp
#include <pluginlib/class_loader.hpp>
#include <polygon_base/regular_polygon.hpp>

int main(int argc, char** argv)
{
    // To avoid unused parameter warnings
    (void) argc;
    (void) argv;

    pluginlib::ClassLoader<polygon_base::RegularPolygon> poly_loader("polygon_base",
                                                                         "polygon_base::RegularPolygon");

    try
    {
        std::shared_ptr<polygon_base::RegularPolygon> triangle = poly_loader.
                                  createSharedInstance("polygon_plugins::Triangle");
        triangle->initialize(10.0);

        std::shared_ptr<polygon_base::RegularPolygon> square = poly_loader.
                                  createSharedInstance("polygon_plugins::Square");
        square->initialize(10.0);

        printf("Triangle area: %.2f\n", triangle->area());
        printf("Square area: %.2f\n", square->area());
    }
    catch(pluginlib::PluginlibException& ex)
    {
        printf("The plugin failed to load for some reason. Error: %s\n", ex.what());
    }
}
```

(continues on next page)
The **ClassLoader** is the key class to understand, defined in the `class_loader.hpp` header file:

- It is templated with the base class, i.e. `polygon_base::RegularPolygon`.
- The first argument is a string for the package name of the base class, i.e. `polygon_base`.
- The second argument is a string with the fully qualified base class type for the plugin, i.e. `polygon_base::RegularPolygon`.

There are a number of ways to instantiate an instance of the class. In this example, we're using shared pointers. We just need to call `createSharedInstance` with the fully-qualified type of the plugin class, in this case, `polygon_plugins::Square`.

Important note: the `polygon_base` package in which this node is defined does NOT depend on the `polygon_plugins` class. The plugins will be loaded dynamically without any dependency needing to be declared. Furthermore, we're instantiating the classes with hardcoded plugin names, but you can also do so dynamically with parameters, etc.

### 4 Build and run

Navigate back to the root of your workspace, `ros2_ws`, and build your new packages:

```
colcon build --packages-select polygon_base polygon_plugins
```

From `ros2_ws`, be sure to source the setup files:

**Linux**

```
source install/setup.bash
```

**macOS**

```
. install/setup.bash
```

**Windows**

```
call install/setup.bat
```

Now run the node:

```
ros2 run polygon_base area_node
```

It should print:

```
Triangle area: 43.30
Square area: 100.00
```
Summary

Congratulations! You've just written and used your first plugins.

Intermediate

Managing Dependencies with rosdep

Goal: Manage external dependencies using rosdep.

Tutorial level: Intermediate

Time: 5 minutes

Contents

• What is rosdep?
• A little about package.xml files
• How does rosdep work?
• How do I know what keys to put in my package.xml?
• What if my library isn't in rosdistro?
• How do I use the rosdep tool?

Author: Steve Macenski

This tutorial will explain how to manage external dependencies using rosdep.

What is rosdep?

rosdep is ROS's dependency management utility that can work with ROS packages and external libraries. rosdep is a command-line utility for identifying and installing dependencies to build or install a package. It can be or is invoked when:

• Building a workspace and needing appropriate dependencies to build the packages within
• Install packages (e.g. sudo apt install ros-iron-demo-nodes-cpp) to check the dependencies needed for it to execute
• and more!

It has the ability to work over a single package or over a directory of packages (e.g. workspace).
A little about package.xml files

A package’s package.xml file contains a set of dependencies. The dependencies in this file are generally referred to as “rosdep keys”. These are represented in the tags `<depend>`, `<test_depend>`, `<exec_depend>`, `<build_depend>`, and `<build_export_depend>`. They specify in what situation each of the dependencies are required in.

- For dependencies only used in testing the code (e.g. gtest), use `<test_depend>`.
- For dependencies only used in building the code, use `<build_depend>`.
- For dependencies needed by headers the code exports, use `<build_export_depend>`.
- For dependencies only used when running the code, use `<exec_depend>`.
- For mixed purposes, use `<depend>`, which covers build, export, and execution time dependencies.

These dependencies are manually populated in the package.xml file by the package’s creators and should be an exhaustive list of any non-builtin libraries and packages it requires.

How does rosdep work?

rosdep will check for package.xml files in its path or for a specific package and find the rosdep keys stored within. These keys are then cross-referenced against a central index to find the appropriate ROS package or software library in various package managers. Finally, once the packages are found, they are installed and ready to go!

The central index is known as rosdistro, which may be found here. We’ll explore that more in the next section.

How do I know what keys to put in my package.xml?

Great question, I’m glad you asked!

For ROS packages (e.g. nav2_bt_navigator), you may simply place the name of the package. You can find a list of all released ROS packages in rosdistro at `<distro>/distribution.yaml` for your given ROS distribution.

For non-ROS package system dependencies, we will need to find the keys for a particular library. In general, there are two files of interest: rosdep/base.yaml and rosdep/python.yaml. base.yaml in general contains the apt system dependencies. python.yaml in general contains the pip python dependencies.

To find a key, search for your library in this file and find the name in `yaml` that contains it. This is the key to put in a package.xml file.

For example, imagine a package had a dependency on doxygen because it is a great piece of software that cares about quality documentation (hint hint). We would search base.yaml for doxygen and come across:

```yaml
doxygen:
  arch: [doxygen]
  debian: [doxygen]
  fedora: [doxygen]
  freebsd: [doxygen]
  gentoo: [app-doc/doxygen]
  macports: [doxygen]
  nixos: [doxygen]
  openembedded: [doxygen@meta-oe]
  opensuse: [doxygen]
  rhel: [doxygen]
  ubuntu: [doxygen]
```
That means our rosdep key is doxygen, which would resolve to those various names in different operating system’s package managers for installation.

**What if my library isn't in rosdistro?**

If your library isn’t in rosdistro, you can experience the greatness that is open-source software development: you can add it yourself! Pull requests for rosdistro are typically merged well within a week.

Detailed instructions may be found here for how to contribute new rosdep keys. If for some reason these may not be contributed openly, it is possible to fork rosdistro and maintain a alternate index for use.

**How do I use the rosdep tool?**

Now that we have some understanding of rosdep, package.xml, and rosdistro, we’re ready to use the utility itself! Firstly, if this is the first time using rosdep, it must be initialized via:

```bash
sudo rosdep init
rosdep update
```

This will initialize rosdep and update will update the locally cached rosdistro index. It is a good idea to update rosdep on occasion to get the latest index.

Finally, we can run rosdep install to install dependencies. Typically, this is run over a workspace with many packages in a single call to install all dependencies. A call for that would appear as the following, if in the root of the workspace with directory src containing source code.

```bash
rosdep install --from-paths src -y --ignore-src
```

Breaking that down:

- `--from-paths src` specifies the path to check for package.xml files to resolve keys for
- `-y` means to default yes to all prompts from the package manager to install without prompts
- `--ignore-src` means to ignore installing dependencies, even if a rosdep key exists, if the package itself is also in the workspace.

There are additional arguments and options available. Use rosdep -h to see them.

**Creating an action**

**Goal:** Define an action in a ROS 2 package.

**Tutorial level:** Intermediate

**Time:** 5 minutes

**Contents**

- Background
- Prerequisites
- Tasks
  - 1 Creating an interface package

4.8. ROS 2 Documentation
Background

You learned about actions previously in the *Understanding actions* tutorial. Like the other communication types and their respective interfaces (topics/msg and services/srv), you can also custom-define actions in your packages. This tutorial shows you how to define and build an action that you can use with the action server and action client you will write in the next tutorial.

Prerequisites

You should have *ROS 2* and *colcon* installed.

You should know how to set up a *workspace* and create packages.

Remember to *source your ROS 2 installation* first.

Tasks

1 Creating an interface package

Linux

```
mkdir -p ~/ros2_ws/src # you can reuse an existing workspace with this naming convention
cd ~/ros2_ws/src
ros2 pkg create --license Apache-2.0 custom_action_interfaces
```

macOS

```
mkdir -p ~/ros2_ws/src
    cd ~/ros2_ws/src
    ros2 pkg create --license Apache-2.0 custom_action_interfaces
```

Windows

```
md \ros2_ws\src
    cd \ros2_ws\src
    ros2 pkg create --license Apache-2.0 custom_action_interfaces
```
2 Defining an action

Actions are defined in .action files of the form:

# Request
---
# Result
---
# Feedback

An action definition is made up of three message definitions separated by ---.

- A request message is sent from an action client to an action server initiating a new goal.
- A result message is sent from an action server to an action client when a goal is done.
- Feedback messages are periodically sent from an action server to an action client with updates about a goal.

An instance of an action is typically referred to as a goal.

Say we want to define a new action “Fibonacci” for computing the Fibonacci sequence.

Create an action directory in our ROS 2 package custom_action_interfaces:

Linux

```
cd custom_action_interfaces
mkdir action
```

macOS

```
cd custom_action_interfaces
mkdir action
```

Windows

```
cd custom_action_interfaces
md action
```

Within the action directory, create a file called Fibonacci.action with the following contents:

```
int32 order
---
int32[] sequence
---
int32[] partial_sequence
```

The goal request is the order of the Fibonacci sequence we want to compute, the result is the final sequence, and the feedback is the partial_sequence computed so far.
3 Building an action

Before we can use the new Fibonacci action type in our code, we must pass the definition to the rosidl code generation pipeline.

This is accomplished by adding the following lines to our CMakeLists.txt before the ament_package() line, in the custom_action_interfaces:

```bash
find_package(rosidl_default_generators REQUIRED)
rosidl_generate_interfaces(${PROJECT_NAME}
  "action/Fibonacci.action"
)
```

We should also add the required dependencies to our package.xml:

```xml
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

We should now be able to build the package containing the Fibonacci action definition:

```bash
# Change to the root of the workspace
cd ~/ros2_ws
# Build
colcon build
```

We're done!

By convention, action types will be prefixed by their package name and the word action. So when we want to refer to our new action, it will have the full name custom_action_interfaces/action/Fibonacci.

We can check that our action built successfully with the command line tool. First source our workspace:

Linux

```bash
source install/local_setup.bash
```

macOS

```bash
source install/local_setup.bash
```

Windows

```bash
call install\local_setup.bat
```

Now check that our action definition exists:

```bash
ros2 interface show custom_action_interfaces/action/Fibonacci
```

You should see the Fibonacci action definition printed to the screen.
Summary

In this tutorial, you learned the structure of an action definition. You also learned how to correctly build a new action interface using CMakeLists.txt and package.xml, and how to verify a successful build.

Next steps

Next, let’s utilize your newly defined action interface by creating an action service and client (in Python or C++).

Related content

For more detailed information about ROS actions, please refer to the design article.

Writing an action server and client (C++)

Goal: Implement an action server and client in C++.

Tutorial level: Intermediate

Time: 15 minutes

Contents

- Background
- Prerequisites
- Tasks
  - 1 Creating the custom_action_cpp package
  - 2 Writing an action server
  - 3 Writing an action client
- Summary
- Related content

Background

Actions are a form of asynchronous communication in ROS. Action clients send goal requests to action servers. Action servers send goal feedback and results to action clients.
Prerequisites

You will need the `custom_action_interfaces` package and the `Fibonacci.action` interface defined in the previous tutorial, *Creating an action*.

Tasks

1 Creating the custom_action_cpp package

As we saw in the *Creating a package* tutorial, we need to create a new package to hold our C++ and supporting code.

1.1 Creating the custom_action_cpp package

Go into the action workspace you created in the *previous tutorial* (remember to source the workspace), and create a new package for the C++ action server:

Linux

```bash
cd ~/ros2_ws/src
ros2 pkg create --dependencies custom_action_interfaces rclcpp rclcpp_action rclcpp_components --license Apache-2.0 -- custom_action_cpp
```

macOS

```bash
cd ~/ros2_ws/src
ros2 pkg create --dependencies custom_action_interfaces rclcpp rclcpp_action rclcpp_components --license Apache-2.0 -- custom_action_cpp
```

Windows

```bash
cd \ros2_ws\src
ros2 pkg create --dependencies custom_action_interfaces rclcpp rclcpp_action rclcpp_components --license Apache-2.0 -- custom_action_cpp
```

1.2 Adding in visibility control

In order to make the package compile and work on Windows, we need to add in some “visibility control”. For more details, see *Windows Symbol Visibility in the Windows Tips and Tricks document*.

Open up `custom_action_cpp/include/custom_action_cpp/visibility_control.h`, and put the following code in:

```c
#ifndef CUSTOM_ACTION_CPP__VISIBILITY_CONTROL_H_
#define CUSTOM_ACTION_CPP__VISIBILITY_CONTROL_H_

#ifdef __cplusplus
extern "C"
{
#endif

// This logic was borrowed (then namespaced) from the examples on the gcc wiki:
```

(continues on next page)
2 Writing an action server

Let's focus on writing an action server that computes the Fibonacci sequence using the action we created in the Creating an action tutorial.
2.1 Writing the action server code

Open up custom_action_cpp/src/fibonacci_action_server.cpp, and put the following code in:

```cpp
#include <functional>
#include <memory>
#include <thread>
#include "custom_action_interfaces/action/fibonacci.hpp"
#include "rclcpp/rclcpp.hpp"
#include "rclcpp_action/rclcpp_action.hpp"
#include "rclcpp_components/register_node_macro.hpp"

namespace custom_action_cpp
{
  class FibonacciActionServer : public rclcpp::Node
  {
  public:
    using Fibonacci = custom_action_interfaces::action::Fibonacci;
    using GoalHandleFibonacci = rclcpp_action::ServerGoalHandle<Fibonacci>;

    CUSTOM_ACTION_CPP_PUBLIC
    explicit FibonacciActionServer(const rclcpp::NodeOptions & options = ~)
    : Node("fibonacci_action_server", options)
    {
      using namespace std::placeholders;
      this->action_server_ = rclcpp_action::create_server<Fibonacci>(
        this,
        "fibonacci",
        std::bind(&FibonacciActionServer::handle_goal, this, _1, _2),
        std::bind(&FibonacciActionServer::handle_cancel, this, _1),
        std::bind(&FibonacciActionServer::handle_accepted, this, _1));
    }
  private:
    rclcpp_action::Server<Fibonacci>::SharedPtr action_server_;

    rclcpp_action::GoalResponse handle_goal(
      const rclcpp_action::GoalUUID & uuid,
      std::shared_ptr<GoalHandle<Fibonacci>> goal)
    {
      RCLCPP_INFO(this->get_logger(), "Received goal request with order \%d", goal->order);
      (void)uuid;
      return rclcpp_action::GoalResponse::ACCEPT_AND_EXECUTE;
    }

    rclcpp_action::CancelResponse handle_cancel(
      const std::shared_ptr<GoalHandleFibonacci> goal_handle)
    {
      (continues on next page)
    }
  }
}
```

(continues on next page)
RCLCPP_INFO(this->get_logger(), "Received request to cancel goal");
(void)goal_handle;
return rclcpp_action::CancelResponse::ACCEPT;
}

void handle_accepted(const std::shared_ptr<GoalHandleFibonacci> goal_handle) {
using namespace std::placeholders;
// this needs to return quickly to avoid blocking the executor, so spin up a new...
std::thread{std::bind(&FibonacciActionServer::execute, this, _1), goal_handle}.detach();
}

void execute(const std::shared_ptr<GoalHandleFibonacci> goal_handle) {
RCLCPP_INFO(this->get_logger(), "Executing goal");
rclcpp::Rate loop_rate(1);
const auto goal = goal_handle->get_goal();
auto feedback = std::make_shared<Fibonacci::Feedback>();
auto & sequence = feedback->partial_sequence;
sequence.push_back(0);
sequence.push_back(1);
auto result = std::make_shared<Fibonacci::Result>();
for (int i = 1; (i < goal->order) && rclcpp::ok(); ++i) {
    // Check if there is a cancel request
    if (goal_handle->is_cancelling()) {
        result->sequence = sequence;
        goal_handle->canceled(result);
        RCLCPP_INFO(this->get_logger(), "Goal canceled");
        return;
    }
    // Update sequence
    sequence.push_back(sequence[i] + sequence[i - 1]);
    // Publish feedback
    goal_handle->publish_feedback(feedback);
    RCLCPP_INFO(this->get_logger(), "Publish feedback");

    loop_rate.sleep();
}

    // Check if goal is done
    if (rclcpp::ok()) {
        result->sequence = sequence;
        goal_handle->succeed(result);
        RCLCPP_INFO(this->get_logger(), "Goal succeeded");
    }
}; // class FibonacciActionServer

} // namespace custom_action_cpp
(continues on next page)
The first few lines include all of the headers we need to compile.

Next we create a class that is a derived class of rclcpp::Node:

```cpp
class FibonacciActionServer : public rclcpp::Node
```

The constructor for the FibonacciActionServer class initializes the node name as fibonacci_action_server:

```cpp
explicit FibonacciActionServer(const rclcpp::NodeOptions & options = rclcpp::NodeOptions())
    : Node("fibonacci_action_server", options)
```

The constructor also instantiates a new action server:

```cpp
this->action_server_ = rclcpp_action::create_server<Fibonacci>(
    this,
    "fibonacci",
    std::bind(&FibonacciActionServer::handle_goal, this, _1, _2),
    std::bind(&FibonacciActionServer::handle_cancel, this, _1),
    std::bind(&FibonacciActionServer::handle_accepted, this, _1));
```

An action server requires 6 things:

1. The templated action type name: Fibonacci.
2. A ROS 2 node to add the action to: this.
3. The action name: 'fibonacci'.
4. A callback function for handling goals: handle_goal
5. A callback function for handling cancellation: handle_cancel.
6. A callback function for handling goal accept: handle_accept.

The implementation of the various callbacks is next in the file. Note that all of the callbacks need to return quickly, otherwise we risk starving the executor.

We start with the callback for handling new goals:

```cpp
rclcpp_action::GoalResponse handle_goal(
    const rclcpp_action::GoalUUID & uuid,
    std::shared_ptr<const Fibonacci::Goal> goal)
{
    RCLCPP_INFO(this->get_logger(), "Received goal request with order %d", goal->order);
    (void)uuid;
    return rclcpp_action::GoalResponse::ACCEPT_AND_EXECUTE;
}
```

This implementation just accepts all goals.

Next up is the callback for dealing with cancellation:
rclcpp_action::CancelResponse handle_cancel(
    const std::shared_ptr<GoalHandleFibonacci> goal_handle)
{
    RCLCPP_INFO(this->get_logger(), "Received request to cancel goal");
    (void)goal_handle;
    return rclcpp_action::CancelResponse::ACCEPT;
}

This implementation just tells the client that it accepted the cancellation.

The last of the callbacks accepts a new goal and starts processing it:

```cpp
void handle_accepted(const std::shared_ptr<GoalHandleFibonacci> goal_handle)
{
    using namespace std::placeholders;
    // this needs to return quickly to avoid blocking the executor, so spin up a new thread
    std::thread{std::bind(&FibonacciActionServer::execute, this, _1), goal_handle}.detach();
}
```

Since the execution is a long-running operation, we spawn off a thread to do the actual work and return from handle_accepted quickly.

All further processing and updates are done in the execute method in the new thread:

```cpp
void execute(const std::shared_ptr<GoalHandleFibonacci> goal_handle)
{
    RCLCPP_INFO(this->get_logger(), "Executing goal");
    rclcpp::Rate loop_rate(1);
    const auto goal = goal_handle->get_goal();
    auto feedback = std::make_shared<Fibonacci::Feedback>();
    auto & sequence = feedback->partial_sequence;
    sequence.push_back(0);
    sequence.push_back(1);
    auto result = std::make_shared<Fibonacci::Result>();
    for (int i = 1; (i < goal->order) && rclcpp::ok(); ++i) {
        // Check if there is a cancel request
        if (goal_handle->is_cancelling()) {
            result->sequence = sequence;
            goal_handle->canceled(result);
            RCLCPP_INFO(this->get_logger(), "Goal canceled");
            return;
        }
        // Update sequence
        sequence.push_back(sequence[i] + sequence[i - 1]);
        // Publish feedback
        goal_handle->publish_feedback(feedback);
        RCLCPP_INFO(this->get_logger(), "Publish feedback");
    }
    loop_rate.sleep();
}
```

(continues on next page)
This work thread processes one sequence number of the Fibonacci sequence every second, publishing a feedback update for each step. When it has finished processing, it marks the `goal_handle` as succeeded, and quits.

We now have a fully functioning action server. Let’s get it built and running.

### 2.2 Compiling the action server

In the previous section we put the action server code into place. To get it to compile and run, we need to do a couple of additional things.

First we need to setup the `CMakeLists.txt` so that the action server is compiled. Open up `custom_action_cpp/CMakeLists.txt`, and add the following right after the `find_package` calls:

```
add_library(action_server SHARED
    src/fibonacci_action_server.cpp)
target_include_directories(action_server PRIVATE
    $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
    $<INSTALL_INTERFACE:include>)
target_compile_definitions(action_server PRIVATE "CUSTOM_ACTION_CPP_BUILDING_DLL")
ament_target_dependencies(action_server "custom_action_interfaces" "rclcpp" "rclcpp_action" "rclcpp_components")
rclcpp_components_register_node(action_server PLUGIN "custom_action_cpp::FibonacciActionServer" EXECUTABLE fibonacci_action_server)
install(TARGETS
    action_server
    ARCHIVE DESTINATION lib
    LIBRARY DESTINATION lib
    RUNTIME DESTINATION bin)
```

And now we can compile the package. Go to the top-level of the `ros2_ws`, and run:

```
colcon build
```

This should compile the entire workspace, including the `fibonacci_action_server` in the `custom_action_cpp` package.
2.3 Running the action server

Now that we have the action server built, we can run it. Source the workspace we just built (ros2_ws), and try to run the action server:

```
ros2 run custom_action_cpp fibonacci_action_server
```

3 Writing an action client

3.1 Writing the action client code

Open up custom_action_cpp/src/fibonacci_action_client.cpp, and put the following code in:

```cpp
#include <functional>
#include <future>
#include <memory>
#include <string>
#include <sstream>
#include "custom_action_interfaces/action/fibonacci.hpp"
#include "rclcpp/rclcpp.hpp"
#include "rclcpp_action/rclcpp_action.hpp"
#include "rclcpp_components/register_node_macro.hpp"

namespace custom_action_cpp
{
    class FibonacciActionClient : public rclcpp::Node
    {
    public:
        using Fibonacci = custom_action_interfaces::action::Fibonacci;
        using GoalHandleFibonacci = rclcpp_action::ClientGoalHandle<Fibonacci>;
        explicit FibonacciActionClient(const rclcpp::NodeOptions & options)
            : Node("fibonacci_action_client", options)
        {
            this->client_ptr_ = rclcpp_action::create_client<Fibonacci>(this, "fibonacci");
            using namespace std::placeholders;
            this->timer_ = this->create_wall_timer(std::chrono::milliseconds(500),
                std::bind(&FibonacciActionClient::send_goal, this, fibonacci");

            this->timer_ = this->create_wall_timer(
                std::chrono::milliseconds(500),
                std::bind(&FibonacciActionClient::send_goal, this));
        }

        void send_goal()
        {
            using namespace std::placeholders;
            this->timer_ = this->create_wall_timer(
                std::chrono::milliseconds(500),
                std::bind(&FibonacciActionClient::send_goal, this));
        }
    }
}
```

(continues on next page)
if (!this->client_ptr_->wait_for_action_server()) {
    RCLCPP_ERROR(this->get_logger(), "Action server not available after waiting");
    rclcpp::shutdown();
}

auto goal_msg = Fibonacci::Goal();
goal_msg.order = 10;
RCLCPP_INFO(this->get_logger(), "Sending goal");

auto send_goal_options = rclcpp_action::Client<Fibonacci>::SendGoalOptions();
send_goal_options.goal_response_callback =
    std::bind(&FibonacciActionClient::goal_response_callback, this, _1);
send_goal_options.feedback_callback =
    std::bind(&FibonacciActionClient::feedback_callback, this, _1, _2);
send_goal_options.result_callback =
    std::bind(&FibonacciActionClient::result_callback, this, _1);
this->client_ptr_->async_send_goal(goal_msg, send_goal_options);

private:
    rclcpp_action::Client<Fibonacci>::SharedPtr client_ptr_;  
rclcpp::TimerBase::SharedPtr timer_; 

    void goal_response_callback(const GoalHandleFibonacci::SharedPtr & goal_handle) 
    {
        if (!goal_handle) {
            RCLCPP_ERROR(this->get_logger(), "Goal was rejected by server");
        } else {
            RCLCPP_INFO(this->get_logger(), "Goal accepted by server, waiting for result");
        }
    } 

    void feedback_callback(GoalHandleFibonacci::SharedPtr, 
                           const std::shared_ptr<const Fibonacci::Feedback> feedback) 
    {
        std::stringstream ss;
        ss << "Next number in sequence received: ";
        for (auto number : feedback->partial_sequence) {
            ss << number << " ";
        }
        RCLCPP_INFO(this->get_logger(), ss.str().c_str());
    }

    void result_callback(const GoalHandleFibonacci::WrappedResult & result) 
    {
        switch (result.code) {
            case rclcpp_action::ResultCode::SUCCEEDED: break;
            case rclcpp_action::ResultCode::ABORTED: 
                RCLCPP_ERROR(this->get_logger(), "Goal was aborted");
                break;
        }
    }
return;
case rclcpp_action::ResultCode::CANCELED:
    RCLCPP_ERROR(this->get_logger(), "Goal was canceled");
    return;
default:
    RCLCPP_ERROR(this->get_logger(), "Unknown result code");
    return;
}
std::stringstream ss;
ss << "Result received: ";
for (auto number : result.result->sequence) {
    ss << number << " ";
}
RCLCPP_INFO(this->get_logger(), ss.str().c_str());
rclcpp::shutdown();
}; // class FibonacciActionClient
} // namespace custom_action_cpp

RCLCPP_COMPONENTS_REGISTER_NODE(custom_action_cpp::FibonacciActionClient)

The first few lines include all of the headers we need to compile.

Next we create a class that is a derived class of rclcpp::Node:

class FibonacciActionClient : public rclcpp::Node

The constructor for the FibonacciActionClient class initializes the node name as fibonacci_action_client:

explicit FibonacciActionClient(const rclcpp::NodeOptions & options) :
    Node("fibonacci_action_client", options)

The constructor also instantiates a new action client:

    this->client_ptr_ = rclcpp_action::create_client<Fibonacci>(
        this,
        "fibonacci");

An action client requires 3 things:

1. The templated action type name: Fibonacci.
2. A ROS 2 node to add the action client to: this.
3. The action name: 'fibonacci'.

We also instantiate a ROS timer that will kick off the one and only call to send_goal:

    this->timer_ = this->create_wall_timer(
        std::chrono::milliseconds(500),
        std::bind(&FibonacciActionClient::send_goal, this));

When the timer expires, it will call send_goal:
```cpp
void send_goal()
{
    using namespace std::placeholders;
    this->timer_->cancel();

    if (!this->client_ptr_->wait_for_action_server()) {
        RCLCPP_ERROR(this->get_logger(), "Action server not available after waiting");
        rclcpp::shutdown();
    }

    auto goal_msg = Fibonacci::Goal();
goal_msg.order = 10;

    RCLCPP_INFO(this->get_logger(), "Sending goal");

    auto send_goal_options = rclcpp_action::Client<Fibonacci>::SendGoalOptions();
    send_goal_options.goal_response_callback =
        std::bind(&FibonacciActionClient::goal_response_callback,
                  this, _1);
    send_goal_options.feedback_callback =
        std::bind(&FibonacciActionClient::feedback_callback,
                  this, _1, _2);
    send_goal_options.result_callback =
        std::bind(&FibonacciActionClient::result_callback,
                  this, _1);
    this->client_ptr_->async_send_goal(goal_msg, send_goal_options);
}
```

This function does the following:

1. Cancels the timer (so it is only called once).
2. Waits for the action server to come up.
3. Instantiates a new `Fibonacci::Goal`.
4. Sets the response, feedback, and result callbacks.
5. Sends the goal to the server.

When the server receives and accepts the goal, it will send a response to the client. That response is handled by `goal_response_callback`:

```cpp
void goal_response_callback(const GoalHandleFibonacci::SharedPtr & goal_handle)
{
    if (!goal_handle) {
        RCLCPP_ERROR(this->get_logger(), "Goal was rejected by server");
    } else {
        RCLCPP_INFO(this->get_logger(), "Goal accepted by server, waiting for result");
    }
}
```

Assuming the goal was accepted by the server, it will start processing. Any feedback to the client will be handled by the `feedback_callback`:

```cpp
void feedback_callback(
    GoalHandleFibonacci::SharedPtr,
    const std::shared_ptr<const Fibonacci::Feedback> feedback)
```

(continues on next page)
When the server is finished processing, it will return a result to the client. The result is handled by the `result_callback`:

```cpp
void result_callback(const GoalHandleFibonacci::WrappedResult & result) {
    switch (result.code) {
        case rclcpp_action::ResultCode::SUCCEEDED:
            break;
        case rclcpp_action::ResultCode::ABORTED:
            RCLCPP_ERROR(this->get_logger(), "Goal was aborted");
            return;
        case rclcpp_action::ResultCode::CANCELED:
            RCLCPP_ERROR(this->get_logger(), "Goal was canceled");
            return;
        default:
            RCLCPP_ERROR(this->get_logger(), "Unknown result code");
            return;
    }
    std::stringstream ss;
    ss << "Result received: ";
    for (auto number : result.result->sequence) {
        ss << number << " ";
    }
    RCLCPP_INFO(this->get_logger(), ss.str().c_str());
    rclcpp::shutdown();
} // class FibonacciActionClient
```

We now have a fully functioning action client. Let’s get it built and running.

### 3.2 Compiling the action client

In the previous section we put the action client code into place. To get it to compile and run, we need to do a couple of additional things.

First we need to setup the CMakeLists.txt so that the action client is compiled. Open up `custom_action_cpp/CMakeLists.txt`, and add the following right after the `find_package` calls:

```cmake
add_library(action_client SHARED
    src/fibonacci_action_client.cpp)
target_include_directories(action_client PRIVATE
    ${<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
    ${<INSTALL_INTERFACE:include>})
```

(continues on next page)
target_compile_definitions(action_client
PRIVATE "CUSTOM_ACTION_CPP_BUILDING_DLL")
ament_target_dependencies(action_client
"custom_action_interfaces"
"rclcpp"
"rclcpp_action"
"rclcpp_components")
rclcpp_components_register_node(action_client PLUGIN "custom_action_
--cpp::FibonacciActionClient" EXECUTABLE fibonacci_action_client)
install(TARGETS
action_client
ARCHIVE DESTINATION lib
LIBRARY DESTINATION lib
RUNTIME DESTINATION bin)

And now we can compile the package. Go to the top-level of the ros2_ws, and run:

```
colon build
```

This should compile the entire workspace, including the fibonacci_action_client in the custom_action_cpp package.

### 3.3 Running the action client

Now that we have the action client built, we can run it. First make sure that an action server is running in a separate terminal. Now source the workspace we just built (ros2_ws), and try to run the action client:

```
ros2 run custom_action_cpp fibonacci_action_client
```

You should see logged messages for the goal being accepted, feedback being printed, and the final result.

### Summary

In this tutorial, you put together a C++ action server and action client line by line, and configured them to exchange goals, feedback, and results.

### Related content

- There are several ways you could write an action server and client in C++; check out the minimal_action_server and minimal_action_client packages in the ros2/examples repo.
- For more detailed information about ROS actions, please refer to the design article.
Writing an action server and client (Python)

Goal: Implement an action server and client in Python.

Tutorial level: Intermediate

Time: 15 minutes

Contents

• Background
• Prerequisites
• Tasks
  – 1 Writing an action server
  – 2 Writing an action client
• Summary
• Related content

Background

Actions are a form of asynchronous communication in ROS 2. Action clients send goal requests to action servers. Action servers send goal feedback and results to action clients.

Prerequisites

You will need the custom_action_interfaces package and the Fibonacci.action interface defined in the previous tutorial, Creating an action.

Tasks

1 Writing an action server

Let’s focus on writing an action server that computes the Fibonacci sequence using the action we created in the Creating an action tutorial.

Until now, you’ve created packages and used ros2 run to run your nodes. To keep things simple in this tutorial, however, we’ll scope the action server to a single file. If you’d like to see what a complete package for the actions tutorials looks like, check out action_tutorials.

Open a new file in your home directory, let’s call it fibonacci_action_server.py, and add the following code:

```python
import rclpy
from rclpy.action import ActionServer
from rclpy.node import Node
from custom_action_interfaces.action import Fibonacci
```

(continues on next page)
class FibonacciActionServer(Node):

    def __init__(self):
        super().__init__('fibonacci_action_server')
        self._action_server = ActionServer(
            self,
            Fibonacci,
            'fibonacci',
            self.execute_callback)

    def execute_callback(self, goal_handle):
        self.get_logger().info('Executing goal...')
        result = Fibonacci.Result()
        return result

def main(args=None):
    rclpy.init(args=args)

    fibonacci_action_server = FibonacciActionServer()

    rclpy.spin(fibonacci_action_server)

if __name__ == '__main__':
    main()

Line 8 defines a class FibonacciActionServer that is a subclass of Node. The class is initialized by calling the Node constructor, naming our node fibonacci_action_server:

    super().__init__('fibonacci_action_server')

In the constructor we also instantiate a new action server:

    self._action_server = ActionServer(
        self,
        Fibonacci,
        'fibonacci',
        self.execute_callback)

An action server requires four arguments:
1. A ROS 2 node to add the action client to: self.
2. The type of the action: Fibonacci (imported in line 5).
3. The action name: 'fibonacci'.
4. A callback function for executing accepted goals: self.execute_callback. This callback must return a result message for the action type.

We also define an execute_callback method in our class:
This is the method that will be called to execute a goal once it is accepted.

Let’s try running our action server:

Linux

```bash
python3 fibonacci_action_server.py
```

macOS

```bash
python3 fibonacci_action_server.py
```

Windows

```bash
python fibonacci_action_server.py
```

In another terminal, we can use the command line interface to send a goal:

```bash
ros2 action send_goal fibonacci custom_action_interfaces/action/Fibonacci "{order: 5}"
```

In the terminal that is running the action server, you should see a logged message “Executing goal...” followed by a warning that the goal state was not set. By default, if the goal handle state is not set in the execute callback it assumes the aborted state.

We can use the method `succeed()` on the goal handle to indicate that the goal was successful:

```python
def execute_callback(self, goal_handle):
    self.get_logger().info('Executing goal...')
    goal_handle.succeed()
    result = Fibonacci.Result()
    return result
```

Now if you restart the action server and send another goal, you should see the goal finished with the status SUCCEEDED.

Now let’s make our goal execution actually compute and return the requested Fibonacci sequence:

```python
def execute_callback(self, goal_handle):
    self.get_logger().info('Executing goal...')
    sequence = [0, 1]
    for i in range(1, goal_handle.request.order):
        sequence.append(sequence[i] + sequence[i-1])
    goal_handle.succeed()
    result = Fibonacci.Result()
    result.sequence = sequence
    return result
```

After computing the sequence, we assign it to the result message field before returning.
Again, restart the action server and send another goal. You should see the goal finish with the proper result sequence.

### 1.2 Publishing feedback

One of the nice things about actions is the ability to provide feedback to an action client during goal execution. We can make our action server publish feedback for action clients by calling the goal handle’s `publish_feedback()` method.

We’ll replace the `sequence` variable, and use a feedback message to store the sequence instead. After every update of the feedback message in the for-loop, we publish the feedback message and sleep for dramatic effect:

```python
import time
import rclpy
from rclpy.action import ActionServer
from rclpy.node import Node
from custom_action_interfaces.action import Fibonacci

class FibonacciActionServer(Node):
    def __init__(self):
        super().__init__('fibonacci_action_server')
        self._action_server = ActionServer(self, Fibonacci, 'fibonacci', self.execute_callback)

    def execute_callback(self, goal_handle):
        self.get_logger().info('Executing goal...')
        feedback_msg = Fibonacci.Feedback()
        feedback_msg.partial_sequence = [0, 1]
        for i in range(1, goal_handle.request.order):
            feedback_msg.partial_sequence.append(feedback_msg.partial_sequence[i] + feedback_msg.partial_sequence[i-1])
            self.get_logger().info('Feedback: {0}'.format(feedback_msg.partial_sequence))
            goal_handle.publish_feedback(feedback_msg)
            time.sleep(1)
        goal_handle.succeed()
        result = Fibonacci.Result()
        result.sequence = feedback_msg.partial_sequence
        return result

def main(args=None):
    rclpy.init(args=args)
    fibonacci_action_server = FibonacciActionServer()
```

(continues on next page)
After restarting the action server, we can confirm that feedback is now published by using the command line tool with the \(--feedback\) option:

```
ros2 action send_goal --feedback fibonacci custom_action_interfaces/action/Fibonacci "--{order: 5}"
```

### 2 Writing an action client

We'll also scope the action client to a single file. Open a new file, let’s call it `fibonacci_action_client.py`, and add the following boilerplate code:

```python
import rclpy
from rclpy.action import ActionClient
from rclpy.node import Node
from custom_action_interfaces.action import Fibonacci

class FibonacciActionClient(Node):
    def __init__(self):
        super().__init__('fibonacci_action_client')
        self._action_client = ActionClient(self, Fibonacci, 'fibonacci')

    def send_goal(self, order):
        goal_msg = Fibonacci.Goal()
        goal_msg.order = order
        self._action_client.wait_for_server()
        return self._action_client.send_goal_async(goal_msg)

def main(args=None):
    rclpy.init(args=args)
    action_client = FibonacciActionClient()
    future = action_client.send_goal(10)
    rclpy.spin_until_future_complete(action_client, future)
```

(continues on next page)
main()

We’ve defined a class `FibonacciActionClient` that is a subclass of `Node`. The class is initialized by calling the `Node` constructor, naming our node `fibonacci_action_client`:

```
super().__init__('fibonacci_action_client')
```

Also in the class constructor, we create an action client using the custom action definition from the previous tutorial on *Creating an action*:

```
self._action_client = ActionClient(self, Fibonacci, 'fibonacci')
```

We create an `ActionClient` by passing it three arguments:

1. A ROS 2 node to add the action client to: `self`
2. The type of the action: `Fibonacci`
3. The action name: 'fibonacci'

Our action client will be able to communicate with action servers of the same action name and type.

We also define a method `send_goal` in the `FibonacciActionClient` class:

```
def send_goal(self, order):
    goal_msg = Fibonacci.Goal()
    goal_msg.order = order

    self._action_client.wait_for_server()

    return self._action_client.send_goal_async(goal_msg)
```

This method waits for the action server to be available, then sends a goal to the server. It returns a future that we can later wait on.

After the class definition, we define a function `main()` that initializes ROS 2 and creates an instance of our `FibonacciActionClient` node. It then sends a goal and waits until that goal has been completed.

Finally, we call `main()` in the entry point of our Python program.

Let’s test our action client by first running the action server built earlier:

Linux

```
python3 fibonacci_action_server.py
```

macOS

```
python3 fibonacci_action_server.py
```

Windows

```
python fibonacci_action_server.py
```

In another terminal, run the action client:

Linux

```
You should see messages printed by the action server as it successfully executes the goal:

```
[INFO] [fibonacci_action_server]: Executing goal...
[INFO] [fibonacci_action_server]: Feedback: array('i', [0, 1, 1])
[INFO] [fibonacci_action_server]: Feedback: array('i', [0, 1, 1, 2])
[INFO] [fibonacci_action_server]: Feedback: array('i', [0, 1, 1, 2, 3])
[INFO] [fibonacci_action_server]: Feedback: array('i', [0, 1, 1, 2, 3, 5])
# etc.
```

The action client should start up, and then quickly finish. At this point, we have a functioning action client, but we don’t see any results or get any feedback.

### 2.1 Getting a result

So we can send a goal, but how do we know when it is completed? We can get the result information with a couple steps. First, we need to get a goal handle for the goal we sent. Then, we can use the goal handle to request the result.

Here’s the complete code for this example:

```python
import rclpy
from rclpy.action import ActionClient
from rclpy.node import Node
from custom_action_interfaces.action import Fibonacci

class FibonacciActionClient(Node):
    def __init__(self):
        super().__init__("fibonacci_action_client")
        self._action_client = ActionClient(self, Fibonacci, 'fibonacci')

    def send_goal(self, order):
        goal_msg = Fibonacci.Goal()
        goal_msg.order = order
        self._action_client.wait_for_server()
        self._send_goal_future = self._action_client.send_goal_async(goal_msg)
        self._send_goal_future.add_done_callback(self.goal_response_callback)

    def goal_response_callback(self, future):
        future.result()
```

(continues on next page)
goal_handle = future.result()
if not goal_handle.accepted:
    self.get_logger().info('Goal rejected :(')
    return

self.get_logger().info('Goal accepted :')

self._get_result_future = goal_handle.get_result_future
self._get_result_future.add_done_callback(self.get_result_callback)

def get_result_callback(self, future):
    result = future.result().result
    self.get_logger().info('Result: {0}.format(result.sequence))
rclpy.shutdown()

def main(args=None):
    rclpy.init(args=args)

    action_client = FibonacciActionClient()
    action_client.send_goal(10)
    rclpy.spin(action_client)

    if __name__ == '__main__':
        main()

The `ActionClient.send_goal_async()` method returns a future to a goal handle. First we register a callback for when the future is complete:

```
self._send_goal_future.add_done_callback(self.goal_response_callback)
```

Note that the future is completed when an action server accepts or rejects the goal request. Let’s look at the `goal_response_callback` in more detail. We can check to see if the goal was rejected and return early since we know there will be no result:

```
def goal_response_callback(self, future):
    goal_handle = future.result()
    if not goal_handle.accepted:
        self.get_logger().info('Goal rejected :(')
        return
    
    self.get_logger().info('Goal accepted :')
```

Now that we’ve got a goal handle, we can use it to request the result with the method `get_result_async()`. Similar to sending the goal, we will get a future that will complete when the result is ready. Let’s register a callback just like we did for the goal response:

```
self._get_result_future = goal_handle.get_result_async()
self._get_result_future.add_done_callback(self.get_result_callback)
```
In the callback, we log the result sequence and shutdown ROS 2 for a clean exit:

```python
def get_result_callback(self, future):
    result = future.result().result
    self.get_logger().info('Result: {}'.format(result.sequence))
    rclpy.shutdown()
```

With an action server running in a separate terminal, go ahead and try running our Fibonacci action client!

Linux

```bash
python3 fibonacci_action_client.py
```

macOS

```bash
python3 fibonacci_action_client.py
```

Windows

```bash
python fibonacci_action_client.py
```

You should see logged messages for the goal being accepted and the final result.

### 2.2 Getting feedback

Our action client can send goals. Nice! But it would be great if we could get some feedback about the goals we send from the action server.

Here’s the complete code for this example:

```python
import rclpy
from rclpy.action import ActionClient
from rclpy.node import Node
from custom_action_interfaces.action import Fibonacci

class FibonacciActionClient(Node):
    def __init__(self):
        super().__init__('fibonacci_action_client')
        self._action_client = ActionClient(self, Fibonacci, 'fibonacci')

    def send_goal(self, order):
        goal_msg = Fibonacci.Goal()
        goal_msg.order = order

        self._action_client.wait_for_server()

        self._send_goal_future = self._action_client.send_goal_async(goal_msg, feedback_callback=self.feedback_callback)

        self._send_goal_future.add_done_callback(self.goal_response_callback)
```

(continues on next page)
def goal_response_callback(self, future):
    goal_handle = future.result()
    if not goal_handle.accepted:
        self.get_logger().info('Goal rejected :(')
        return

    self.get_logger().info('Goal accepted :)
    self._get_result_future = goal_handle.get_result_async()
    self._get_result_future.add_done_callback(self.get_result_callback)

def get_result_callback(self, future):
    result = future.result().result
    self.get_logger().info('Result: {0}'.format(result.sequence))
    rclpy.shutdown()

def feedback_callback(self, feedback_msg):
    feedback = feedback_msg.feedback
    self.get_logger().info('Received feedback: {0}'.format(feedback.partial_sequence))

def main(args=None):
    rclpy.init(args=args)

    action_client = FibonacciActionClient()

    action_client.send_goal(10)

    rclpy.spin(action_client)

if __name__ == '__main__':
    main()

Here's the callback function for feedback messages:

def feedback_callback(self, feedback_msg):
    feedback = feedback_msg.feedback
    self.get_logger().info('Received feedback: {0}'.format(feedback.partial_sequence))

In the callback we get the feedback portion of the message and print the partial_sequence field to the screen.

We need to register the callback with the action client. This is achieved by additionally passing the callback to the action client when we send a goal:

    self._send_goal_future = self._action_client.send_goal_async(goal_msg, feedback_callback=self.feedback_callback)

We're all set. If we run our action client, you should see feedback being printed to the screen.
Summary

In this tutorial, you put together a Python action server and action client line by line, and configured them to exchange goals, feedback, and results.

Related content

- There are several ways you could write an action server and client in Python; check out the `minimal_action_server` and `minimal_action_client` packages in the `ros2/examples` repo.
- For more detailed information about ROS actions, please refer to the design article.

Composing multiple nodes in a single process

Table of Contents

- Background
- Run the demos
  - Discover available components
  - Run-time composition using ROS services with a publisher and subscriber
  - Run-time composition using ROS services with a server and client
  - Compile-time composition using ROS services
  - Run-time composition using dlopen
  - Composition using launch actions
- Advanced Topics
  - Unloading components
  - Remapping container name and namespace
  - Remap component names and namespaces
  - Passing parameter values into components
  - Passing additional arguments into components
- Composable nodes as shared libraries
- Composing Non-Node Derived Components

Goal: Compose multiple nodes into a single process.

Tutorial level: Intermediate

Time: 20 minutes
Background

See the conceptual article.

Run the demos

The demos use executables from rclcpp_components, ros2component, and composition packages, and can be run with the following commands.

Discover available components

To see what components are registered and available in the workspace, execute the following in a shell:

```
ros2 component types
```

The terminal will return the list of all available components:

```bash
(... components of other packages here)
composition
    composition::Talker
    composition::Listener
    composition::NodeLikeListener
    composition::Server
    composition::Client
(... components of other packages here)
```

Run-time composition using ROS services with a publisher and subscriber

In the first shell, start the component container:

```
ros2 run rclcpp_components component_container
```

Open the second shell and verify that the container is running via ros2 command line tools:

```
ros2 component list
```

You should see a name of the component:

```
/ComponentManager
```

In the second shell load the talker component (see talker source code):

```
ros2 component load /ComponentManager composition::Talker
```

The command will return the unique ID of the loaded component as well as the node name:

```
Loaded component 1 into '/ComponentManager' container node as '/talker'
```

Now the first shell should show a message that the component was loaded as well as repeated message for publishing a message.

Run another command in the second shell to load the listener component (see listener source code):
ros2 component load /ComponentManager composition composition::Listener

Terminal will return:

Loaded component 2 into '/ComponentManager' container node as '/listener'

The ros2 command line utility can now be used to inspect the state of the container:

ros2 component list

You will see the following result:

/ComponentManager
  1 /talker
  2 /listener

Now the first shell should show repeated output for each received message.

**Run-time composition using ROS services with a server and client**

The example with a server and a client is very similar.

In the first shell:

```bash
ros2 run rclcpp_components component_container
```

In the second shell (see server and client source code):

```bash
ros2 component load /ComponentManager composition composition::Server
ros2 component load /ComponentManager composition composition::Client
```

In this case the client sends a request to the server, the server processes the request and replies with a response, and the client prints the received response.

**Compile-time composition using ROS services**

This demos shows that the same shared libraries can be reused to compile a single executable running multiple components. The executable contains all four components from above: talker and listener as well as server and client.

In the shell call (see source code):

```bash
ros2 run composition manual_composition
```

This should show repeated messages from both pairs, the talker and the listener as well as the server and the client.

**Note:** Manually-composed components will not be reflected in the `ros2 component list` command line tool output.
Run-time composition using dlopen

This demo presents an alternative to run-time composition by creating a generic container process and explicitly passing the libraries to load without using ROS interfaces. The process will open each library and create one instance of each “rclcpp::Node” class in the library source code.

Linux

```bash
ros2 run composition dlopen_composition `ros2 pkg prefix composition`/lib/libtalker_component.so `ros2 pkg prefix composition`/lib/liblistener_component.so
```

macOS

```bash
ros2 run composition dlopen_composition `ros2 pkg prefix composition`/lib/libtalker_component.dylib `ros2 pkg prefix composition`/lib/liblistener_component.dylib
```

Windows

```bash
> ros2 pkg prefix composition
to get the path to where composition is installed. Then call

```bash
> ros2 run composition dlopen_composition <path_to_composition_install>/bin\talker_component.dll <path_to_composition_install>/bin\listener_component.dll
```

Now the shell should show repeated output for each sent and received message.

**Note:** dlopen-composed components will not be reflected in the ros2 component list command line tool output.

Composition using launch actions

While the command line tools are useful for debugging and diagnosing component configurations, it is frequently more convenient to start a set of components at the same time. To automate this action, we can use a launch file:

```bash
ros2 launch composition composition_demo_launch.py
```

Advanced Topics

Now that we have seen the basic operation of components, we can discuss a few more advanced topics.

Unloading components

In the first shell, start the component container:

```bash
ros2 run rclcpp_components component_container
```

Verify that the container is running via ros2 command line tools:

```bash
ros2 component list
```

You should see a name of the component:
In the second shell load both the talker and listener as we have before:

```bash
ros2 component load /ComponentManager composition composition::Talker
ros2 component load /ComponentManager composition composition::Listener
```

Use the unique ID to unload the node from the component container.

```bash
ros2 component unload /ComponentManager 1 2
```

The terminal should return:

```
Unloaded component 1 from '/ComponentManager' container
Unloaded component 2 from '/ComponentManager' container
```

In the first shell, verify that the repeated messages from talker and listener have stopped.

### Remapping container name and namespace

The component manager name and namespace can be remapped via standard command line arguments:

```bash
ros2 run rclcpp_components component_container --ros-args -r __node:=MyContainer -r __ns:=/ns
```

In a second shell, components can be loaded by using the updated container name:

```bash
ros2 component load /ns/MyContainer composition composition::Listener
```

**Note:** Namespace remappings of the container do not affect loaded components.

### Remap component names and namespaces

Component names and namespaces may be adjusted via arguments to the load command.

In the first shell, start the component container:

```bash
ros2 run rclcpp_components component_container
```

Some examples of how to remap names and namespaces.

Remap node name:

```bash
ros2 component load /ComponentManager composition composition::Talker --node-name talker2
```

Remap namespace:

```bash
ros2 component load /ComponentManager composition composition::Talker --node-namespace /ns
```

Remap both:
Now use `ros2` command line utility:

```
ros2 component list
```

In the console you should see corresponding entries:

```
/ComponentManager
  1 /talker2
  2 /ns/talker
  3 /ns2/talker3
```

**Note:** Namespace remappings of the container do not affect loaded components.

### Passing parameter values into components

The `ros2 component load` command-line supports passing arbitrary parameters to the node as it is constructed. This functionality can be used as follows:

```
ros2 component load /ComponentManager image_tools image_tools::Cam2Image -p burger_mode:=true
```

### Passing additional arguments into components

The `ros2 component load` command-line supports passing particular options to the component manager for use when constructing the node. As of now, the only command-line option that is supported is to instantiate a node using intra-process communication. This functionality can be used as follows:

```
ros2 component load /ComponentManager composition composition::Talker -e use_intra_process_comms:=true
```

### Composable nodes as shared libraries

If you want to export a composable node as a shared library from a package and use that node in another package that does link-time composition, add code to the CMake file which imports the actual targets in downstream packages.

Then install the generated file and export the generated file.

A practical example can be seen here: [ROS Discourse - Ament best practice for sharing libraries](https://discourse.ros.org/t/ament-best-practice-for-sharing-libraries/10420/2)
Composing Non-Node Derived Components

In ROS 2, components allow for more efficient use of system resources and provide a powerful feature that enables you to create reusable functionality that is not tied to a specific node.

One advantage of using components is that they allow you to create non-node derived functionality as standalone executables or shared libraries that can be loaded into the ROS system as needed.

To create a component that is not derived from a node, follow these guidelines:

1. Implement a constructor that takes `const rclcpp::NodeOptions&` as its argument.
2. Implement the `get_node_base_interface()` method, which should return a `NodeBaseInterface::SharedPtr`. You can use the `get_node_base_interface()` method of a node that you create in your constructor to provide this interface.

Here’s an example of a component that is not derived from a node, which listens to a ROS topic: `node_like_listener_component`.

For more information on this topic, you can refer to this discussion.

Monitoring for parameter changes (C++)

Goal: Learn to use the ParameterEventHandler class to monitor and respond to parameter changes.

Tutorial level: Intermediate

Time: 20 minutes

Background

Often a node needs to respond to changes to its own parameters or another node’s parameters. The ParameterEventHandler class makes it easy to listen for parameter changes so that your code can respond to them. This tutorial will show you how to use the C++ version of the ParameterEventHandler class to monitor for changes to a node’s own parameters as well as changes to another node’s parameters.
Vulcanexus Documentation, Release 1.0.0

Prerequisites

Before starting this tutorial, you should first complete the following tutorials:

- Understanding parameters
- Using parameters in a class (C++)

Tasks

In this tutorial, you will create a new package to contain some sample code, write some C++ code to use the ParameterEventHandler class, and test the resulting code.

1 Create a package

First, open a new terminal and source your ROS 2 installation so that ros2 commands will work.

Follow these instructions to create a new workspace named ros2_ws.

Recall that packages should be created in the src directory, not the root of the workspace. So, navigate into ros2_ws/src and then create a new package there:

```
ros2 pkg create --build-type ament_cmake cpp_parameter_event_handler --dependencies
˓→rclcpp
```

Your terminal will return a message verifying the creation of your package cpp_parameter_event_handler and all its necessary files and folders.

The --dependencies argument will automatically add the necessary dependency lines to package.xml and CMakeLists.txt.

1.1 Update package.xml

Because you used the --dependencies option during package creation, you don’t have to manually add dependencies to package.xml or CMakeLists.txt. As always, though, make sure to add the description, maintainer email and name, and license information to package.xml.

```
<description>C++ parameter events client tutorial</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache License 2.0</license>
```

2 Write the C++ node

Inside the ros2_ws/src/cpp_parameter_event_handler/src directory, create a new file called parameter_event_handler.cpp and paste the following code within:

```
#include <memory>
#include "rclcpp/rclcpp.hpp"

class SampleNodeWithParameters : public rclcpp::Node
```
```cpp
public:
    SampleNodeWithParameters()
    : Node("node_with_parameters")
    {
        this->declare_parameter("an_int_param", 0);

        // Create a parameter subscriber that can be used to monitor parameter changes
        // (for this node's parameters as well as other nodes' parameters)
        param_subscriber_ = std::make_shared<rclcpp::ParameterEventHandler>(this);

        // Set a callback for this node's integer parameter, "an_int_param"
        auto cb = [this](const rclcpp::Parameter & p) {
            RCLCPP_INFO(this->get_logger(), "cb: Received an update to parameter \"%s\" of type %s: \"%ld\"",
                p.get_name().c_str(),
                p.get_type_name().c_str(),
                p.as_int());
            cb_handle_ = param_subscriber_->add_parameter_callback("an_int_param", cb);
        }
    }

private:
    std::shared_ptr<rclcpp::ParameterEventHandler> param_subscriber_;  // Parameter EventHandler
    std::shared_ptr<rclcpp::ParameterCallbackHandle> cb_handle_;  // Parameter Callback Handle

int main(int argc, char ** argv)
{
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<SampleNodeWithParameters>());
    rclcpp::shutdown();

    return 0;
}
```

**2.1 Examine the code**

The first statement, `#include <memory>` is included so that the code can utilize the `std::make_shared` template. The next, `#include "rclcpp/rclcpp.hpp"` is included to allow the code to reference the various functionality provided by the rclcpp interface, including the `ParameterEventHandler` class.

After the class declaration, the code defines a class, `SampleNodeWithParameters`. The constructor for the class declares an integer parameter `an_int_param`, with a default value of 0. Next, the code creates a `ParameterEventHandler` that will be used to monitor changes to parameters. Finally, the code creates a lambda function and sets it as the callback to invoke whenever `an_int_param` is updated.

**Note:** It is very important to save the handle that is returned by `add_parameter_callback`; otherwise, the callback will not be properly registered.
SampleNodeWithParameters()
: Node("node_with_parameters")
{
    this->declare_parameter("an_int_param", 0);

    // Create a parameter subscriber that can be used to monitor parameter changes
    // (for this node's parameters as well as other nodes' parameters)
    param_subscriber_ = std::make_shared<rclcpp::ParameterEventHandler>(this);

    // Set a callback for this node's integer parameter, "an_int_param"
    auto cb = [this](const rclcpp::Parameter & p) {
        RCLCPP_INFO(this->get_logger(), "cb: Received an update to parameter "
        p.get_name().c_str(), " of type %s: \%
        p.as_int());
    };
    cb_handle_ = param_subscriber_->add_parameter_callback("an_int_param", cb);
}

Following the SampleNodeWithParameters is a typical main function which initializes ROS, spins the sample node so that it can send and receive messages, and then shuts down after the user enters ^C at the console.

int main(int argc, char ** argv)
{
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<SampleNodeWithParameters>());
    rclcpp::shutdown();

    return 0;
}

2.2 Add executable

To build this code, first open the CMakeLists.txt file and add the following lines of code below the dependency
find_package(rclcpp REQUIRED)

add_executable(parameter_event_handler src/parameter_event_handler.cpp)
ament_target_dependencies(parameter_event_handler rclcpp)

install(TARGETS
    parameter_event_handler
    DESTINATION lib/${PROJECT_NAME}
)

3 Build and run

It's good practice to run `rosdep` in the root of your workspace (`ros2_ws`) to check for missing dependencies before building:

Linux

```bash
rosdep install -i --from-path src --rosdistro $ROS_DISTRO -y
```

macOS

rosdep only runs on Linux, so you can skip ahead to next step.

Windows

rosdep only runs on Linux, so you can skip ahead to next step.

Navigate back to the root of your workspace, `ros2_ws`, and build your new package:

```bash
colcon build --packages-select cpp_parameter_event_handler
```

Open a new terminal, navigate to `ros2_ws`, and source the setup files:

Linux

```
. install/setup.bash
```

macOS

```
. install/setup.bash
```

Windows

```
call install/setup.bat
```

Now run the node:

```bash
ros2 run cpp_parameter_event_handler parameter_event_handler
```

The node is now active and has a single parameter and will print a message whenever this parameter is updated. To test this, open up another terminal and source the ROS setup file as before (`. install/setup.bash`) and execute the following command:

```bash
ros2 param set node_with_parameters an_int_param 43
```

The terminal running the node will display a message similar to the following:

```
[INFO] [1606950498.422461764] [node_with_parameters]: cb: Received an update to parameter "an_int_param" of type integer: "43"
```

The callback we set previously in the node has been invoked and has displayed the new updated value. You can now terminate the running `parameter_event_handler` sample using `^C` in the terminal.
3.1 Monitor changes to another node's parameters

You can also use the ParameterEventHandler to monitor parameter changes to another node’s parameters. Let’s update the SampleNodeWithParameters class to also monitor for changes to a parameter in another node. We will use the parameter_blackboard demo application to host a double parameter that we will monitor for updates.

First update the constructor to add the following code after the existing code:

```cpp
// Now, add a callback to monitor any changes to the remote node's parameter. In this case, we supply the remote node name.
auto cb2 = [this](const rclcpp::Parameter & p) {
    RCLCPP_INFO(this->get_logger(), "cb2: Received an update to parameter "%s" of type: %s: "]%lf"", p.get_name().c_str(), p.get_type_name().c_str(), p.as_double());
};

auto remote_node_name = std::string("parameter_blackboard");
auto remote_param_name = std::string("a_double_param");
cb_handle2_ = param_subscriber_->add_parameter_callback(remote_param_name, cb2, remote_node_name);
```

Then add another member variable, cb_handle2 for the additional callback handle:

```cpp
private:
    std::shared_ptr<rclcpp::ParameterEventHandler> param_subscriber_{};
    std::shared_ptr<rclcpp::ParameterCallbackHandle> cb_handle_{};
    std::shared_ptr<rclcpp::ParameterCallbackHandle> cb_handle2_{}; // Add this
};
```

In a terminal, navigate back to the root of your workspace, ros2_ws, and build your updated package as before:

```
colcon build --packages-select cpp_parameter_event_handler
```

Then source the setup files:

Linux

```
. install/setup.bash
```

macOS

```
. install/setup.bash
```

Windows

```
call install/setup.bat
```

Now, to test monitoring of remote parameters, first run the newly-built parameter_event_handler code:

```
ros2 run cpp_parameter_event_handler parameter_event_handler
```

Next, from another terminal (with ROS initialized), run the parameter_blackboard demo application, as follows:
Finally, from a third terminal (with ROS initialized), let’s set a parameter on the parameter_blackboard node:

```
ros2 param set parameter_blackboard a_double_param 3.45
```

Upon executing this command, you should see output in the parameter_event_handler window, indicating that the callback function was invoked upon the parameter update:

```
[INFO] [1606952588.237531933] [node_with_parameters]: cb2: Received an update to parameter "a_double_param" of type: double: "3.45"
```

**Summary**

You created a node with a parameter and used the ParameterEventHandler class to set a callback to monitor changes to that parameter. You also used the same class to monitor changes to a remote node. The ParameterEventHandler is a convenient way to monitor for parameter changes so that you can then respond to the updated values.

**Related content**

To learn how to adapt ROS 1 parameter files for ROS 2, see the *Migrating YAML parameter files from ROS 1 to ROS2* tutorial.

**Launch**

ROS 2 Launch files allow you to start up and configure a number of executables containing ROS 2 nodes simultaneously.

**Creating a launch file**

**Goal:** Create a launch file to run a complex ROS 2 system.

**Tutorial level:** Intermediate

**Time:** 10 minutes

**Contents**

- Prerequisites
- Background
- Tasks
  - Setup
  - 1 Write the launch file
  - 3 ros2 launch
  - 4 Introspect the system with rqt_graph
- Summary
Prerequisites

This tutorial uses the `rqt_graph` and `turtlesim` packages.
You will also need to use a text editor of your preference.
As always, don’t forget to source ROS 2 in every new terminal you open.

Background

The launch system in ROS 2 is responsible for helping the user describe the configuration of their system and then execute it as described. The configuration of the system includes what programs to run, where to run them, what arguments to pass them, and ROS-specific conventions which make it easy to reuse components throughout the system by giving them each a different configuration. It is also responsible for monitoring the state of the processes launched, and reporting and/or reacting to changes in the state of those processes.

Launch files written in Python, XML, or YAML can start and stop different nodes as well as trigger and act on various events. See Using Python, XML, and YAML for ROS 2 Launch Files for a description of the different formats. The package providing this framework is `launch_ros`, which uses the non-ROS-specific `launch` framework underneath.

The design document details the goal of the design of ROS 2’s launch system (not all functionality is currently available).

Tasks

1 Setup

Create a new directory to store your launch files:

```
mkdir launch
```

2 Write the launch file

Let’s put together a ROS 2 launch file using the `turtlesim` package and its executables. As mentioned above, this can either be in Python, XML, or YAML.

Python

Copy and paste the complete code into the `launch/turtlesim_mimic_launch.py` file:

```python
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='turtlesim',
            namespace='turtlesim1',
            executable='turtlesim_node',
            name='sim'
        ),
        Node(
            package='turtlesim',
```
namespace='turtlesim2',
executable='turtlesim_node',
name='sim'
),
Node(  
    package='turtlesim',
    executable='mimic',
    name='mimic',
    remappings=[
        ('/input/pose', '/turtlesim1/turtle1/pose'),
        ('/output/cmd_vel', '/turtlesim2/turtle1/cmd_vel'),
    ]
)
})

XML
Copy and paste the complete code into the launch/turtlesim_mimic_launch.xml file:

```xml
<launch>
    <node pkg="turtlesim" exec="turtlesim_node" name="sim" namespace="turtlesim1"/>
    <node pkg="turtlesim" exec="turtlesim_node" name="sim" namespace="turtlesim2"/>
    <node pkg="turtlesim" exec="mimic" name="mimic">
        <remap from="/input/pose" to="/turtlesim1/turtle1/pose"/>
        <remap from="/output/cmd_vel" to="/turtlesim2/turtle1/cmd_vel"/>
    </node>
</launch>
```

YAML
Copy and paste the complete code into the launch/turtlesim_mimic_launch.yaml file:

```yaml
launch:
  - node:
      pkg: "turtlesim"
      exec: "turtlesim_node"
      name: "sim"
      namespace: "turtlesim1"
  - node:
      pkg: "turtlesim"
      exec: "turtlesim_node"
      name: "sim"
      namespace: "turtlesim2"
  - node:
      pkg: "turtlesim"
      exec: "mimic"
      name: "mimic"
      remap:
        - from: "/input/pose"
```

(continues on next page)
to: "/turtlesim1/turtle1/pose"
- from: "/output/cmd_vel"
to: "/turtlesim2/turtle1/cmd_vel"

2.1 Examine the launch file

All of the launch files above are launching a system of three nodes, all from the turtlesim package. The goal of the system is to launch two turtlesim windows, and have one turtle mimic the movements of the other.

When launching the two turtlesim nodes, the only difference between them is their namespace values. Unique namespaces allow the system to start two nodes without node name or topic name conflicts. Both turtles in this system receive commands over the same topic and publish their pose over the same topic. With unique namespaces, messages meant for different turtles can be distinguished.

The final node is also from the turtlesim package, but a different executable: mimic. This node has added configuration details in the form of remappings. mimic's /input/pose topic is remapped to /turtlesim1/turtle1/pose and it's /output/cmd_vel topic to /turtlesim2/turtle1/cmd_vel. This means mimic will subscribe to /turtlesim1/sim's pose topic and republish it for /turtlesim2/sim's velocity command topic to subscribe to. In other words, turtlesim2 will mimic turtlesim1's movements.

Python

These import statements pull in some Python launch modules.

```python
from launch import LaunchDescription
from launch_ros.actions import Node
```

Next, the launch description itself begins:

```python
def generate_launch_description():
    return LaunchDescription([
    ])
```

The first two actions in the launch description launch the two turtlesim windows:

```python
Node(
    package='turtlesim',
    namespace='turtlesim1',
    executable='turtlesim_node',
    name='sim')
Node(
    package='turtlesim',
    namespace='turtlesim2',
    executable='turtlesim_node',
    name='sim')
```

The final action launches the mimic node with the remaps:
Node(
    package='turtlesim',
    executable='mimic',
    name='mimic',
    remappings=[
        ('/input/pose', '/turtlesim1/turtle1/pose'),
        ('/output/cmd_vel', '/turtlesim2/turtle1/cmd_vel'),
    ]
)

XML

The first two actions launch the two turtlesim windows:

```
<node pkg="turtlesim" exec="turtlesim_node" name="sim" namespace="turtlesim1"/>
<node pkg="turtlesim" exec="turtlesim_node" name="sim" namespace="turtlesim2"/>
```

The final action launches the mimic node with the remaps:

```
<node pkg="turtlesim" exec="mimic" name="mimic">
    <remap from="/input/pose" to="/turtlesim1/turtle1/pose"/>
    <remap from="/output/cmd_vel" to="/turtlesim2/turtle1/cmd_vel"/>
</node>
```

YAML

The first two actions launch the two turtlesim windows:

```
- node:
    pkg: "turtlesim"
    exec: "turtlesim_node"
    name: "sim"
    namespace: "turtlesim1"

- node:
    pkg: "turtlesim"
    exec: "turtlesim_node"
    name: "sim"
    namespace: "turtlesim2"
```

The final action launches the mimic node with the remaps:

```
- node:
    pkg: "turtlesim"
    exec: "mimic"
    name: "mimic"
    remap:
        - from: "/input/pose"
            to: "/turtlesim1/turtle1/pose"
        - from: "/output/cmd_vel"
            to: "/turtlesim2/turtle1/cmd_vel"
```
3 ros2 launch

To run the launch file created above, enter into the directory you created earlier and run the following command:

**Python**

```bash
cd launch
ros2 launch turtlesim_mimic_launch.py
```

**XML**

```bash
cd launch
ros2 launch turtlesim_mimic_launch.xml
```

**YAML**

```bash
cd launch
ros2 launch turtlesim_mimic_launch.yaml
```

**Note:** It is possible to launch a launch file directly (as we do above), or provided by a package. When it is provided by a package, the syntax is:

```bash
ros2 launch <package_name> <launch_file_name>
```

You learned about creating packages in *Creating a package*.

**Note:** For packages with launch files, it is a good idea to add an `exec_depend` dependency on the `ros2launch` package in your package's `package.xml`:

```xml
<exec_depend>ros2launch</exec_depend>
```

This helps make sure that the `ros2 launch` command is available after building your package. It also ensures that all launch file formats are recognized.

Two turtlesim windows will open, and you will see the following [INFO] messages telling you which nodes your launch file has started:

```text
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [turtlesim_node-1]: process started with pid [11714]
[INFO] [turtlesim_node-2]: process started with pid [11715]
[INFO] [mimic-3]: process started with pid [11716]
```

To see the system in action, open a new terminal and run the `ros2 topic pub` command on the `/turtlesim1/turtle1/cmd_vel` topic to get the first turtle moving:

```bash
ros2 topic pub -r 1/turtlesim1/turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: -1.8}}"
```

You will see both turtles following the same path.
4 Introspect the system with rqt_graph

While the system is still running, open a new terminal and run `rqt_graph` to get a better idea of the relationship between the nodes in your launch file.

Run the command:

```
rqt_graph
```
A hidden node (the `ros2 topic pub` command you ran) is publishing data to the `/turtlesim1/turtle1/cmd_vel` topic on the left, which the `/turtlesim1/sim` node is subscribed to. The rest of the graph shows what was described earlier: `mimic` is subscribed to `/turtlesim1/sim`’s pose topic, and publishes to `/turtlesim2/sim`’s velocity command topic.

**Summary**

Launch files simplify running complex systems with many nodes and specific configuration details. You can create launch files using Python, XML, or YAML, and run them using the `ros2 launch` command.

**Integrating launch files into ROS 2 packages**

**Goal:** Add a launch file to a ROS 2 package

**Tutorial level:** Intermediate

**Time:** 10 minutes
Prerequisites

You should have gone through the tutorial on how to create a ROS 2 package.
As always, don’t forget to source ROS 2 in every new terminal you open.

Background

In the previous tutorial, we saw how to write a standalone launch file. This tutorial will show how to add a launch file to an existing package, and the conventions typically used.

Tasks

1 Create a package

Create a workspace for the package to live in:

Linux

```
mkdir -p launch_ws/src
cd launch_ws/src
```

macOS

```
mkdir -p launch_ws/src
cd launch_ws/src
```

Windows

```
md launch_ws\src
cd launch_ws\src
```

Python package

```
ros2 pkg create py_launch_example --build-type ament_python
```

C++ package

```
ros2 pkg create cpp_launch_example --build-type ament_cmake
```
2 Creating the structure to hold launch files

By convention, all launch files for a package are stored in the `launch` directory inside of the package. Make sure to create a `launch` directory at the top-level of the package you created above.

Python package

For Python packages, the directory containing your package should look like this:

```bash
src/
  py_launch_example/
  launch/
  package.xml
  py_launch_example/
  resource/
  setup.cfg
  setup.py
  test/
```

In order for colcon to find the launch files, we need to inform Python’s setup tools of our launch files using the `data_files` parameter of `setup`.

Inside our `setup.py` file:

```python
import os
from glob import glob
from setuptools import find_packages, setup

package_name = 'py_launch_example'

setup(
    # Other parameters ...
    data_files=[
        # ... Other data files
        # Include all launch files.
        (os.path.join('share', package_name, 'launch'),
         glob(os.path.join('launch', '*.launch', '*.py', '*.yaml')))
    ]
)
```

C++ package

For C++ packages, we will only be adjusting the `CMakeLists.txt` file by adding:

```cmake
# Install launch files.
install(DIRECTORY
    launch
    DESTINATION share/\${PROJECT_NAME}/
)
```

to the end of the file (but before `ament_package()`).
3 Writing the launch file

Python launch file

Inside your launch directory, create a new launch file called my_script_launch.py. _launch.py is recommended, but not required, as the file suffix for Python launch files. However, the launch file name needs to end with launch.py to be recognized and autocompleted by ros2 launch.

Your launch file should define the generate_launch_description() function which returns a launch.LaunchDescription() to be used by the ros2 launch verb.

```python
import launch
import launch_ros.actions

def generate_launch_description():
    return launch.LaunchDescription([
        launch_ros.actions.Node(
            package='demo_nodes_cpp',
            executable='talker',
            name='talker'),
    ])
```

XML launch file

Inside your launch directory, create a new launch file called my_script_launch.xml. _launch.xml is recommended, but not required, as the file suffix for XML launch files.

```xml
<launch>
    <node pkg="demo_nodes_cpp" exec="talker" name="talker"/>
</launch>
```

YAML launch file

Inside your launch directory, create a new launch file called my_script_launch.yaml. _launch.yaml is recommended, but not required, as the file suffix for YAML launch files.

```yaml
launch:
  - node:
      pkg: "demo_nodes_cpp"
      exec: "talker"
      name: "talker"
```

4 Building and running the launch file

Go to the top-level of the workspace, and build it:

```
colcon build
```

After the colcon build has been successful and you’ve sourced the workspace, you should be able to run the launch file as follows:

Python package

Python launch file
ros2 launch py_launch_example my_script_launch.py

XML launch file

ros2 launch py_launch_example my_script_launch.xml

YAML launch file

ros2 launch py_launch_example my_script_launch.yaml

C++ package

Python launch file

ros2 launch cpp_launch_example my_script_launch.py

XML launch file

ros2 launch cpp_launch_example my_script_launch.xml

YAML launch file

ros2 launch cpp_launch_example my_script_launch.yaml

**Documentation**

The launch documentation provides more details on concepts that are also used in launch_ros.

Additional documentation/examples of launch capabilities are forthcoming. See the source code (https://github.com/ros2/launch and https://github.com/ros2/launch_ros) in the meantime.

**Using substitutions**

**Goal:** Learn about substitutions in ROS 2 launch files.

**Tutorial level:** Intermediate

**Time:** 15 minutes

**Table of Contents**

- Background
- Prerequisites
- Using substitutions
  - 1 Create and setup the package
  - 2 Parent launch file
  - 3 Substitutions example launch file
  - 4 Build the package
- Launching example
Background

Launch files are used to start nodes, services and execute processes. This set of actions may have arguments, which affect their behavior. Substitutions can be used in arguments to provide more flexibility when describing reusable launch files. Substitutions are variables that are only evaluated during execution of the launch description and can be used to acquire specific information like a launch configuration, an environment variable, or to evaluate an arbitrary Python expression.

This tutorial shows usage examples of substitutions in ROS 2 launch files.

Prerequisites

This tutorial uses the turtlesim package. This tutorial also assumes you are familiar with creating packages.

As always, don’t forget to source ROS 2 in every new terminal you open.

Using substitutions

1 Create and setup the package

Create a new package of build_type ament_python called launch_tutorial:

```bash
ros2 pkg create launch_tutorial --build-type ament_python
```

Inside of that package, create a directory called launch:

Linux

```bash
mkdir launch_tutorial/launch
```

macOS

```bash
mkdir launch_tutorial/launch
```

Windows

```bash
md launch_tutorial/launch
```

Finally, make sure to add in changes to the setup.py of the package so that the launch files will be installed:

```python
import os
from glob import glob
from setuptools import find_packages, setup

package_name = 'launch_tutorial'
```

(continues on next page)
setup(
    # Other parameters ...
    data_files=[
        # ... Other data files
        # Include all launch files.
        (os.path.join('share', package_name, 'launch'), glob(os.path.join('launch', '*launch.[pxy][yma]*')))
    ]
)

2 Parent launch file

Let's create a launch file that will call and pass arguments to another launch file. To do this, create an example_main_launch.py file in the launch folder of the launch_tutorial package.

```python
from launch_ros.substitutions import FindPackageShare
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch.substitutions import PathJoinSubstitution, TextSubstitution

def generate_launch_description():
    colors = {
        'background_r': '200'
    }

    return LaunchDescription([
        IncludeLaunchDescription([  
            PythonLaunchDescriptionSource([  
                PathJoinSubstitution([  
                    FindPackageShare('launch_tutorial'),  
                    'launch',  
                    'example_substitutions_launch.py'  
                ])
            ]),
            launch_arguments={
                'turtlesim_ns': 'turtlesim2',  
                'use_provided_red': 'True',  
                'new_background_r': TextSubstitution(text=str(colors['background_r']))
            }.items()
        ])
    
In the example_main_launch.py file, the FindPackageShare substitution is used to find the path to the launch_tutorial package. The PathJoinSubstitution substitution is then used to join the path to that package path with the example_substitutions_launch.py file name.

```
The `launch_arguments` dictionary with `turtlesim_ns` and `use_provided_red` arguments is passed to the `IncludeLaunchDescription` action. The `TextSubstitution` substitution is used to define the `new_background_r` argument with the value of the `background_r` key in the `colors` dictionary.

```python
launch_arguments={
    'turtlesim_ns': 'turtlesim2',
    'use_provided_red': 'True',
    'new_background_r': TextSubstitution(text=str(colors['background_r']))
}.items()
```

### 3 Substitutions example launch file

Now create an `example_substitutions_launch.py` file in the same folder.

```python
from launch_ros.actions import Node
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument, ExecuteProcess, TimerAction
from launch.conditions import IfCondition
from launch.substitutions import LaunchConfiguration, PythonExpression

def generate_launch_description():
    turtlesim_ns = LaunchConfiguration('turtlesim_ns')
    use_provided_red = LaunchConfiguration('use_provided_red')
    new_background_r = LaunchConfiguration('new_background_r')

    turtlesim_ns_launch_arg = DeclareLaunchArgument(
        'turtlesim_ns',
        default_value='turtlesim1'
    )

    use_provided_red_launch_arg = DeclareLaunchArgument(
        'use_provided_red',
        default_value='False'
    )

    new_background_r_launch_arg = DeclareLaunchArgument(
        'new_background_r',
        default_value='200'
    )

    turtlesim_node = Node(
        package='turtlesim',
        namespace=turtlesim_ns,
        executable='turtlesim_node',
        name='sim'
    )

    spawn_turtle = ExecuteProcess(
        command=['ros2', 'run', 'turtlesim2', 'turtlesim_node'],
        output='screen'
    )
```

(continues on next page)
```python
cmd=[[
    'ros2 service call ',
    turtlesim_ns,
    '/spawn ',
    'turtlesim/srv/Spawn ',
    '{x: 2, y: 2, theta: 0.2}''
]],
    shell=True)
change_background_r = ExecuteProcess(
    cmd=[[
    'ros2 param set ',
    turtlesim_ns,
    '/sim background_r ',
    '120'
]],
    shell=True)
change_background_r_conditioned = ExecuteProcess(
    condition=IfCondition(
    PythonExpression([[
        new_background_r,
        ' == 200',
        ' and ',
        use_provided_red
    ])
    ),
    cmd=[[
    'ros2 param set ',
    turtlesim_ns,
    '/sim background_r ',
    new_background_r
]],
    shell=True)
return LaunchDescription([turtlesim_ns_launch_arg,
    use_provided_red_launch_arg,
    new_background_r_launch_arg,
    turtlesim_node,
    spawn_turtle,
    change_background_r,
    TimerAction(
        period=2.0,
        actions=[change_background_r_conditioned],
    )
])
```

In the `example_substitutions_launch.py` file, `turtlesim_ns`, `use_provided_red`, and `new_background_r` launch configurations are defined. They are used to store values of launch arguments in the above variables and to pass them to required actions. These `LaunchConfiguration` substitutions allow us to acquire the value of the launch argument in any part of the launch description.
DeclareLaunchArgument is used to define the launch argument that can be passed from the above launch file or from the console.

```python
# DeclareLaunchArgument for 'turtlesim_ns'
turtlesim_ns = LaunchConfiguration('turtlesim_ns')
use_provided_red = LaunchConfiguration('use_provided_red')
new_background_r = LaunchConfiguration('new_background_r')

turtlesim_ns_launch_arg = DeclareLaunchArgument(
    'turtlesim_ns',
    default_value='turtlesim1'
)
use_provided_red_launch_arg = DeclareLaunchArgument(
    'use_provided_red',
    default_value='False'
)
new_background_r_launch_arg = DeclareLaunchArgument(
    'new_background_r',
    default_value='200'
)
```

The `turtlesim_node` node with the namespace set to `turtlesim_ns` LaunchConfiguration substitution is defined.

```python
# Define the node with namespace
package='turtlesim',
namespace=turtlesim_ns,
executable='turtlesim_node',
name='sim'
)
```

Afterwards, the `ExecuteProcess` action called `spawn_turtle` is defined with the corresponding `cmd` argument. This command makes a call to the spawn service of the turtlesim node.

Additionally, the `LaunchConfiguration` substitution is used to get the value of the `turtlesim_ns` launch argument to construct a command string.

```python
# Define the execute process for spawn_turtle
spawn_turtle = ExecuteProcess(
    cmd=[[
        'ros2 service call ',
        turtlesim_ns,
        '/spawn ',
        'turtlesim/srv/Spawn ',
        '{x: 2, y: 2, theta: 0.2}"
    ]],
    shell=True
)
```

The same approach is used for the `change_background_r` and `change_background_r_conditioned` actions that change the turtlesim background’s red color parameter. The difference is that the `change_background_r_conditioned` action is only executed if the provided `new_background_r` argument equals `200` and the `use_provided_red` launch argument is set to `True`. The evaluation inside the `IfCondition` is done using the `PythonExpression` substitution.

```python
# Define the execute process for change_background_r
change_background_r = ExecuteProcess(
    cmd=[[
```

(continues on next page)
4 Build the package

Go to the root of the workspace, and build the package:

```
colcon build
```

Also remember to source the workspace after building.

**Launching example**

Now you can launch the `example_main_launch.py` file using the `ros2 launch` command.

```
ros2 launch launch_tutorial example_main_launch.py
```

This will do the following:
1. Start a turtlesim node with a blue background
2. Spawn the second turtle
3. Change the color to purple
4. Change the color to pink after two seconds if the provided `background_r` argument is 200 and `use_provided_red` argument is True
Modifying launch arguments

If you want to change the provided launch arguments, you can either update them in `launch_arguments` dictionary in the `example_main_launch.py` or launch the `example_substitutions_launch.py` with preferred arguments. To see arguments that may be given to the launch file, run the following command:

```plaintext
ros2 launch launch_tutorial example_substitutions_launch.py --show-args
```

This will show the arguments that may be given to the launch file and their default values.

Arguments (pass arguments as `<name>:=<value>`):

- `'turtlesim_ns'`:
  - no description given
  - (default: `'turtlesim1'`)

- `'use_provided_red'`:
  - no description given
  - (default: `'False'`)

- `'new_background_r'`:
  - no description given
  - (default: `'200'`)

Now you can pass the desired arguments to the launch file as follows:

```plaintext
ros2 launch launch_tutorial example_substitutions_launch.py turtlesim_ns='turtlesim3' use_provided_red='True' new_background_r=200
```

Documentation

The launch documentation provides detailed information about available substitutions.

Summary

In this tutorial, you learned about using substitutions in launch files. You learned about their possibilities and capabilities to create reusable launch files.

You can now learn more about using event handlers in launch files which are used to define a complex set of rules which can be used to dynamically modify the launch file.

Using event handlers

Goal: Learn about event handlers in ROS 2 launch files

Tutorial level: Intermediate

Time: 15 minutes
## Background

Launch in ROS 2 is a system that executes and manages user-defined processes. It is responsible for monitoring the state of processes it launched, as well as reporting and reacting to changes in the state of those processes. These changes are called events and can be handled by registering an event handler with the launch system. Event handlers can be registered for specific events and can be useful for monitoring the state of processes. Additionally, they can be used to define a complex set of rules which can be used to dynamically modify the launch file.

This tutorial shows usage examples of event handlers in ROS 2 launch files.

## Prerequisites

This tutorial uses the `turtlesim` package. This tutorial also assumes you have created a new package of build type `ament_python` called `launch_tutorial`.

This tutorial extends the code shown in the `Using substitutions in launch files` tutorial.

## Using event handlers

### 1 Event handlers example launch file

Create a new file called `example_event_handlers_launch.py` file in the `launch` folder of the `launch_tutorial` package.

```python
from launch_ros.actions import Node
from launch import LaunchDescription
from launch.actions import (DeclareLaunchArgument, EmitEvent, ExecuteProcess,
                            LogInfo, RegisterEventHandler, TimerAction)
from launch.conditions import IfCondition
from launch.event_handlers import (OnExecutionComplete, OnProcessExit,
from launch.events import Shutdown
from launch.substitutions import (EnvironmentVariable, FindExecutable,
                                   LaunchConfiguration, LocalSubstitution,
                                   PythonExpression)
```

(continues on next page)
def generate_launch_description():
    turtlesim_ns = LaunchConfiguration('turtlesim_ns')
    use_provided_red = LaunchConfiguration('use_provided_red')
    new_background_r = LaunchConfiguration('new_background_r')

    turtlesim_ns_launch_arg = DeclareLaunchArgument('turtlesim_ns',
                                                    default_value='turtlesim1')
    use_provided_red_launch_arg = DeclareLaunchArgument('use_provided_red',
                                                         default_value='False')
    new_background_r_launch_arg = DeclareLaunchArgument('new_background_r',
                                                         default_value='200')

    turtlesim_node = Node(
        package='turtlesim',
        namespace=turtlesim_ns,
        executable='turtlesim_node',
        name='sim')

    spawn_turtle = ExecuteProcess(
        cmd=[[FindExecutable(name='ros2'),
             ' service call ',
             turtlesim_ns,
             '/spawn ',
             'turtlesim/srv/Spawn ',
             '{x: 2, y: 2, theta: 0.2}''
             ],
             shell=True)
    )

    change_background_r = ExecuteProcess(
        cmd=[[FindExecutable(name='ros2'),
             ' param set ',
             turtlesim_ns,
             '/sim background_r ',
             '120'
             ],
             shell=True)
    )

    change_background_r_conditioned = ExecuteProcess(
        condition=IfCondition(
            PythonExpression([new_background_r,
                              ' == 200',
                              ' and '],
                              default_value='False',
                              default_value='False')
            )
        ,
        cmd=[[FindExecutable(name='ros2'),
             ' param set ',
             turtlesim_ns,
             '/sim background_r ',
             '120'
             ],
             shell=True)
    )
use_provided_red
),
),
cmd=[[
   FindExecutable(name='ros2'),
   ' param set ',
   turtlesim_ns,
   '/sim background_r ',
   new_background_r
]],
shell=True
)
return LaunchDescription([
  turtlesim_ns_launch_arg,
  use_provided_red_launch_arg,
  new_background_r_launch_arg,
  turtlesim_node,
  RegisterEventHandler(
    OnProcessStart(
      target_action=turtlesim_node,
      on_start=[
        LogInfo(msg='Turtlesim started, spawning turtle'),
        spawn_turtle
      ]
    ),
  ),
  RegisterEventHandler(
    OnProcessIO(
      target_action=spawn_turtle,
      on_stdout=lambda event: LogInfo(msg='Spawn request says "{}"'.format(event.text.decode().strip()))
    ),
  ),
  RegisterEventHandler(
    OnExecutionComplete(
      target_action=spawn_turtle,
      on_completion=[
        LogInfo(msg='Spawn finished'),
        change_background_r,
        TimerAction(
          period=2.0,
          actions=[change_background_r_conditioned],
        )
      ]
    ),
  ),
  RegisterEventHandler(
    OnProcessExit(
      target_action=turtlesim_node,
      on_exit=
    )
  ),
])
(continues on previous page)
on_exit=[
    LogInfo(msg=(EnvironmentVariable(name='USER'),
        ' closed the turtlesim window')),
    EmitEvent(event=Shutdown(
        reason='Window closed'))
],
),
RegisterEventHandler(
    OnShutdown(
        on_shutdown=[LogInfo(
            msg=['Launch was asked to shutdown: ',
                LocalSubstitution('event.reason')]
        )]
),
),
]

RegisterEventHandler actions for the OnProcessStart, OnProcessIO, OnExecutionComplete, OnProcessExit, and OnShutdown events were defined in the launch description.

The OnProcessStart event handler is used to register a callback function that is executed when the turtlesim node starts. It logs a message to the console and executes the spawn_turtle action when the turtlesim node starts.

RegisterEventHandler(
    OnProcessStart(
        target_action=turtlesim_node,
        on_start=[
            LogInfo(msg='Turtlesim started, spawning turtle'),
            spawn_turtle
        ]
    ),
),

The OnProcessIO event handler is used to register a callback function that is executed when the spawn_turtle action writes to its standard output. It logs the result of the spawn request.

RegisterEventHandler(
    OnProcessIO(
        target_action=spawn_turtle,
        on_stdout=lambda event: LogInfo(
            msg='Spawn request says "{}".format(''
                event.text.decode().strip()'))
    ),
),

The OnExecutionComplete event handler is used to register a callback function that is executed when the spawn_turtle action completes. It logs a message to the console and executes the change_background_r and change_background_r_conditioned actions when the spawn action completes.

RegisterEventHandler(
    OnExecutionComplete(
        on_exit=[
            LogInfo(msg=(EnvironmentVariable(name='USER'),
                'closed the turtlesim window')),
            EmitEvent(event=Shutdown(
                reason='Window closed'))
        ]
    ),
),
]

(continues on next page)
target_action=spawn_turtle,
on_completion=[
    LogInfo(msg='Spawn finished'),
    change_background_r,
    TimerAction(
        period=2.0,
        actions=[change_background_r_conditioned],
    )
],
),

The OnProcessExit event handler is used to register a callback function that is executed when the turtlesim node exits. It logs a message to the console and executes the EmitEvent action to emit a Shutdown event when the turtlesim node exits. It means that the launch process will shutdown when the turtlesim window is closed.

RegisterEventHandler(
    OnProcessExit(
        target_action=turtlesim_node,
        on_exit=[
            LogInfo(msg=(EnvironmentVariable(name='USER',
                'closed the turtlesim window'))),
            EmitEvent(event=Shutdown(
                reason='Window closed'))
        ]
    ),
),

Finally, the OnShutdown event handler is used to register a callback function that is executed when the launch file is asked to shutdown. It logs a message to the console why the launch file is asked to shutdown. It logs the message with a reason for shutdown like the closure of turtlesim window or ctrl-c signal made by the user.

RegisterEventHandler(
    OnShutdown(
        on_shutdown=[LogInfo(
            msg=['Launch was asked to shutdown: ',
                 LocalSubstitution('event.reason')]
        )]
    ),
),

**Build the package**

Go to the root of the workspace, and build the package:

```
colcon build
```

Also remember to source the workspace after building.
**Launching example**

Now you can launch the `example_event_handlers_launch.py` file using the `ros2 launch` command.

```
ros2 launch launch_tutorial example_event_handlers_launch.py turtlesim_ns='turtlesim3' 
   --use_provided_red='True' new_background_r=200
```

This will do the following:

1. Start a turtlesim node with a blue background
2. Spawn the second turtle
3. Change the color to purple
4. Change the color to pink after two seconds if the provided `background_r` argument is 200 and `use_provided_red` argument is `True`
5. Shutdown the launch file when the turtlesim window is closed

Additionally, it will log messages to the console when:

1. The turtlesim node starts
2. The spawn action is executed
3. The `change_background_r` action is executed
4. The `change_background_r_conditioned` action is executed
5. The turtlesim node exits
6. The launch process is asked to shutdown.

**Documentation**

The launch documentation provides detailed information about available event handlers.

**Summary**

In this tutorial, you learned about using event handlers in launch files. You learned about their syntax and usage examples to define a complex set of rules to dynamically modify launch files.

**Managing large projects**

**Goal:** Learn best practices of managing large projects using ROS 2 launch files.
**Tutorial level:** Intermediate
**Time:** 20 minutes
Background

This tutorial describes some tips for writing launch files for large projects. The focus is on how to structure launch files so they may be reused as much as possible in different situations. Additionally, it covers usage examples of different ROS 2 launch tools, like parameters, YAML files, remappings, namespaces, default arguments, and RViz configs.

Prerequisites

This tutorial uses the turtlesim and turtle_tf2_py packages. This tutorial also assumes you have created a new package of build type ament_python called launch_tutorial.

Introduction

Large applications on a robot typically involve several interconnected nodes, each of which can have many parameters. Simulation of multiple turtles in the turtle simulator can serve as a good example. The turtle simulation consists of multiple turtle nodes, the world configuration, and the TF broadcaster and listener nodes. Between all of the nodes, there are a large number of ROS parameters that affect the behavior and appearance of these nodes. ROS 2 launch files allow us to start all nodes and set corresponding parameters in one place. By the end of a tutorial, you will build the launch_turtlesim_launch.py launch file in the launch_tutorial package. This launch file will bring up different nodes responsible for the simulation of two turtlesim simulations, starting TF broadcasters and listener, loading parameters, and launching an RViz configuration. In this tutorial, we’ll go over this launch file and all related features used.
Writing launch files

1 Top-level organization

One of the aims in the process of writing launch files should be making them as reusable as possible. This could be done by clustering related nodes and configurations into separate launch files. Afterwards, a top-level launch file dedicated to a specific configuration could be written. This would allow moving between identical robots to be done without changing the launch files at all. Even a change such as moving from a real robot to a simulated one can be done with only a few changes.

We will now go over the top-level launch file structure that makes this possible. Firstly, we will create a launch file that will call separate launch files. To do this, let’s create a launch_turtlesim_launch.py file in the /launch folder of our launch_tutorial package.

```python
import os

from ament_index_python.packages import get_package_share_directory

from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource

def generate_launch_description():
    turtlesim_world_1 = IncludeLaunchDescription(
        PythonLaunchDescriptionSource([os.path.join(
            get_package_share_directory('launch_tutorial'), 'launch'),
            '/turtlesim_world_1_launch.py'])
    )

    turtlesim_world_2 = IncludeLaunchDescription(
        PythonLaunchDescriptionSource([os.path.join(
            get_package_share_directory('launch_tutorial'), 'launch'),
            '/turtlesim_world_2_launch.py'])
    )

    broadcaster_listener_nodes = IncludeLaunchDescription(
        PythonLaunchDescriptionSource([os.path.join(
            get_package_share_directory('launch_tutorial'), 'launch'),
            '/broadcaster_listener_launch.py'])),
        launch_arguments={'target_frame': 'carrot1'}.items()
    )

    mimic_node = IncludeLaunchDescription(
        PythonLaunchDescriptionSource([os.path.join(
            get_package_share_directory('launch_tutorial'), 'launch'),
            '/mimic_launch.py'])
    )

    fixed_frame_node = IncludeLaunchDescription(
        PythonLaunchDescriptionSource([os.path.join(
            get_package_share_directory('launch_tutorial'), 'launch'),
            '/fixed_broadcaster_launch.py'])
    )

    rviz_node = IncludeLaunchDescription(
        PythonLaunchDescriptionSource([os.path.join(
            get_package_share_directory('launch_tutorial'), 'launch'),
            '/rviz_launch.py'])
    )
```

(continues on next page)
This launch file includes a set of other launch files. Each of these included launch files contains nodes, parameters, and possibly, nested includes, which pertain to one part of the system. To be exact, we launch two turtlesim simulation worlds, TF broadcaster, TF listener, mimic, fixed frame broadcaster, and RViz nodes.

Note: Design Tip: Top-level launch files should be short, consist of includes to other files corresponding to subcomponents of the application, and commonly changed parameters.

Writing launch files in the following manner makes it easy to swap out one piece of the system, as we’ll see later. However, there are cases when some nodes or launch files have to be launched separately due to performance and usage reasons.

Note: Design tip: Be aware of the tradeoffs when deciding how many top-level launch files your application requires.

## 2 Parameters

### 2.1 Setting parameters in the launch file

We will begin by writing a launch file that will start our first turtlesim simulation. First, create a new file called turtlesim_world_1_launch.py.

```python
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch.substitutions import LaunchConfiguration, TextSubstitution
from launch_ros.actions import Node

def generate_launch_description():
    background_r_launch_arg = DeclareLaunchArgument(
        'background_r', default_value=TextSubstitution(text='0')
    )
    background_g_launch_arg = DeclareLaunchArgument(
        'background_g', default_value=TextSubstitution(text='84')
    )
    background_b_launch_arg = DeclareLaunchArgument(
        'background_b', default_value=TextSubstitution(text='122')
    )
```
This launch file starts the `turtlesim_node` node, which starts the turtlesim simulation, with simulation configuration parameters that are defined and passed to the nodes.

### 2.2 Loading parameters from YAML file

In the second launch, we will start a second turtlesim simulation with a different configuration. Now create a `turtlesim_world_2_launch.py` file.

```python
import os

from ament_index_python.packages import get_package_share_directory

from launch import LaunchDescription
from launch_ros.actions import Node
def generate_launch_description():
    config = os.path.join(
        get_package_share_directory('launch_tutorial'),
        'config',
        'turtlesim.yaml'
    )
    return LaunchDescription([
        Node(
            package='turtlesim',
            executable='turtlesim_node',
            namespace='turtlesim2',
            name='sim',
            parameters=[config]
        )
    ])
```

This launch file will launch the same `turtlesim_node` with parameter values that are loaded directly from the YAML
configuration file. Defining arguments and parameters in YAML files make it easy to store and load a large number of variables. In addition, YAML files can be easily exported from the current ros2 param list. To learn how to do that, refer to the Understand parameters tutorial.

Let’s now create a configuration file, turtlesim.yaml, in the /config folder of our package, which will be loaded by our launch file.

```
/turtlesim2/sim:
  ros__parameters:
    background_b: 255
    background_g: 86
    background_r: 150
```

If we now start the turtlesim_world_2_launch.py launch file, we will start the turtlesim_node with preconfigured background colors.

To learn more about using parameters and using YAML files, take a look at the Understand parameters tutorial.

### 2.3 Using wildcards in YAML files

There are cases when we want to set the same parameters in more than one node. These nodes could have different namespaces or names but still have the same parameters. Defining separate YAML files that explicitly define namespaces and node names is not efficient. A solution is to use wildcard characters, which act as substitutions for unknown characters in a text value, to apply parameters to several different nodes.

Now let’s create a new turtlesim_world_3_launch.py file similar to turtlesim_world_2_launch.py to include one more turtlesim_node node.

```
...)
  Node(
    package='turtlesim',
    executable='turtlesim_node',
    namespace='turtlesim3',
    name='sim',
    parameters=[config]
  )
```

Loading the same YAML file, however, will not affect the appearance of the third turtlesim world. The reason is that its parameters are stored under another namespace as shown below:

```
/turtlesim3/sim:
  background_b
  background_g
  background_r
```

Therefore, instead of creating a new configuration for the same node that use the same parameters, we can use wildcards syntax. /** will assign all the parameters in every node, despite differences in node names and namespaces.

We will now update the turtlesim.yaml, in the /config folder in the following manner:

```
/**:
  ros__parameters:
    background_b: 255
    background_g: 86
    background_r: 150
```
Now include the `turtlesim_world_3_launch.py` launch description in our main launch file. Using that configuration file in our launch descriptions will assign `background_b`, `background_g`, and `background_r` parameters to specified values in `turtlesim3/sim` and `turtlesim2/sim` nodes.

### 3 Namespaces

As you may have noticed, we have defined the namespace for the `turtle` world in the `turtlesim_world_2_launch.py` file. Unique namespaces allow the system to start two similar nodes without node name or topic name conflicts.

```python
namespace='turtlesim2',
```

However, if the launch file contains a large number of nodes, defining namespaces for each of them can become tedious. To solve that issue, the `PushROSNamespace` action can be used to define the global namespace for each launch file description. Every nested node will inherit that namespace automatically.

To do that, firstly, we need to remove the `namespace='turtlesim2'` line from the `turtlesim_world_2_launch.py` file. Afterwards, we need to update the `launch_turtlesim_launch.py` to include the following lines:

```python
from launch_ros.actions import PushROSNamespace
...

turtlesim_world_2_with_namespace = GroupAction(
    actions=[
        PushROSNamespace('turtlesim2'),
        turtlesim_world_2,
    ]
)
```

Finally, we replace the `turtlesim_world_2` to `turtlesim_world_2_with_namespace` in the `return LaunchDescription` statement. As a result, each node in the `turtlesim_world_2_launch.py` launch description will have a `turtlesim2` namespace.

### 4 Reusing nodes

Now create a `broadcaster_listener_launch.py` file.

```python
from launch import LaunchDescription
from launch_ros.actions import Node
from launch_ros.substitutions import LaunchConfiguration
from launch.substitutions import LaunchConfiguration

from launch_ros.actions import PushROSNamespace

from launch_ros.actions import Node

from launch_ros.substitutions import Path

from launch_ros.descriptions import LaunchConfiguration

from launch_ros.descriptions import Node

from launch_ros.descriptions import PushROSNamespace

def generate_launch_description():
    return LaunchDescription(
        [  
```

(continues on next page)
In this file, we have declared the `target_frame` launch argument with a default value of `turtle1`. The default value means that the launch file can receive an argument to forward to its nodes, or in case the argument is not provided, it will pass the default value to its nodes.

Afterwards, we use the `turtle_tf2_broadcaster` node two times using different names and parameters during launch. This allows us to duplicate the same node without conflicts.

We also start a `turtle_tf2_listener` node and set its `target_frame` parameter that we declared and acquired above.

### 5 Parameter overrides

Recall that we called the `broadcaster_listener_launch.py` file in our top-level launch file. In addition to that, we have passed it `target_frame` launch argument as shown below:

```python
broadcaster_listener_nodes = IncludeLaunchDescription(
    PythonLaunchDescriptionSource([os.path.join(        get_package_share_directory('launch_tutorial'), 'launch'),        '/broadcaster_listener_launch.py'])),
    launch_arguments={'target_frame': 'carrot1'}.items(),
)
```

This syntax allows us to change the default goal target frame to `carrot1`. If you would like `turtle2` to follow `turtle1` instead of the `carrot1`, just remove the line that defines `launch_arguments`. This will assign `target_frame` its
default value, which is `turtle1`.

### 6 Remapping

Now create a `mimic_launch.py` file.

```python
def generate_launch_description():
    return LaunchDescription([
        Node(
            package='turtlesim',
            executable='mimic',
            name='mimic',
            remappings=[
                ('/input/pose', '/turtle2/pose'),
                ('/output/cmd_vel', '/turtlesim2/turtle1/cmd_vel'),
            ]
        )
    ])
```

This launch file will start the `mimic` node, which will give commands to one turtlesim to follow the other. The node is designed to receive the target pose on the topic `/input/pose`. In our case, we want to remap the target pose from `/turtle2/pose` topic. Finally, we remap the `/output/cmd_vel` topic to `/turtlesim2/turtle1/cmd_vel`. This way `turtle1` in our `turtlesim2` simulation world will follow `turtle2` in our initial `turtlesim` world.

### 7 Config files

Let's now create a file called `turtlesim_rviz_launch.py`.

```python
import os

from ament_index_python.packages import get_package_share_directory

from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    rviz_config = os.path.join(
        get_package_share_directory('turtle_tf2_py'),
        'rviz',
        'turtle_rviz.rviz'
    )

    return LaunchDescription([
        Node(
            package='rviz2',
            executable='rviz2',
            name='rviz2',
        )
    ])
```

(continues on next page)
This launch file will start the RViz with the configuration file defined in the turtle_tf2_py package. This RViz configuration will set the world frame, enable TF visualization, and start RViz with a top-down view.

8 Environment Variables

Let’s now create the last launch file called fixed_broadcaster_launch.py in our package.

```python
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch.substitutions import EnvironmentVariable, LaunchConfiguration
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        DeclareLaunchArgument('node_prefix',
            default_value=[EnvironmentVariable('USER'), '_'],
            description='prefix for node name'),
        Node(
            package='turtle_tf2_py',
            executable='fixed_frame_tf2_broadcaster',
            name=[LaunchConfiguration('node_prefix'), 'fixed_broadcaster'],
        )
    ])
```

This launch file shows the way environment variables can be called inside the launch files. Environment variables can be used to define or push namespaces for distinguishing nodes on different computers or robots.

Running launch files

1 Update setup.py

Open setup.py and add the following lines so that the launch files from the launch/ folder and configuration file from the config/ would be installed. The data_files field should now look like this:

```python
import os
from glob import glob
from setuptools import setup
...

data_files=[
    ...
    (os.path.join('share', package_name, 'launch'),
        glob(os.path.join('launch', '*launch.[pxy][yma]*'))),
    ...
]
2 Build and run

To finally see the result of our code, build the package and launch the top-level launch file using the following command:

```
ros2 launch launch_tutorial launch_turtlesim_launch.py
```

You will now see the two turtlesim simulations started. There are two turtles in the first one and one in the second one. In the first simulation, turtle2 is spawned in the bottom-left part of the world. Its aim is to reach the carrot1 frame which is five meters away on the x-axis relative to the turtle1 frame.

The turtlesim2/turtle1 in the second is designed to mimic the behavior of the turtle2.

If you want to control the turtle1, run the teleop node.

```
ros2 run turtlesim turtle_teleop_key
```

As a result, you will see a similar picture:

![Turtle Simulations](image)

In addition to that, the RViz should have started. It will show all turtle frames relative to the world frame, whose origin is at the bottom-left corner.
Summary

In this tutorial, you learned about various tips and practices of managing large projects using ROS 2 launch files.

1. *Creating a launch file.*

   Learn how to create a launch file that will start up nodes and their configurations all at once.

2. *Launching and monitoring multiple nodes.*

   Get a more advanced overview of how launch files work.

3. *Using substitutions.*

   Use substitutions to provide more flexibility when describing reusable launch files.

4. *Using event handlers.*

   Use event handlers to monitor the state of processes or to define a complex set of rules that can be used to dynamically modify the launch file.

5. *Managing large projects.*

   Structure launch files for large projects so they may be reused as much as possible in different situations. See usage examples of different launch tools like parameters, YAML files, remappings, namespaces, default arguments, and RViz configs.

*Note:* If you are coming from ROS 1, you can use the *ROS Launch Migration guide* to help you migrate your launch files to ROS 2.
tf2

Many of the tf2 tutorials are available for both C++ and Python. The tutorials are streamlined to complete either the C++ track or the Python track. If you want to learn both C++ and Python, you should go through the tutorials once for C++ and once for Python.

## Introducing tf2

**Goal:** Run a turtlesim demo and see some of the power of tf2 in a multi-robot example using turtlesim.

**Tutorial level:** Intermediate

**Time:** 10 minutes

### Installing the demo

Let’s start by installing the demo package and its dependencies.

**Linux Binaries**

```bash
sudo apt-get install ros-iron-turtle-tf2-py ros-iron-tf2-tools ros-iron-tf-transformations
```

**From Source**

```bash
# Clone and build the geometry_tutorials repo using the branch that matches your installation
git clone https://github.com/ros/geometry_tutorials.git -b ros2
```
Running the demo

Now that we've installed the turtle_tf2_py tutorial package let's run the demo. First, open a new terminal and source your ROS 2 installation so that ros2 commands will work. Then run the following command:

```
ros2 launch turtle_tf2_py turtle_tf2_demo.launch.py
```

You will see the turtlesim start with two turtles.

In the second terminal window type the following command:

```
ros2 run turtlesim turtle_teleop_key
```

Once the turtlesim is started you can drive the central turtle around in the turtlesim using the keyboard arrow keys, select the second terminal window so that your keystrokes will be captured to drive the turtle.
You can see that one turtle continuously moves to follow the turtle you are driving around.

**What is happening?**

This demo is using the tf2 library to create three coordinate frames: a *world* frame, a *turtle1* frame, and a *turtle2* frame. This tutorial uses a tf2 broadcaster to publish the turtle coordinate frames and a tf2 listener to compute the difference in the turtle frames and move one turtle to follow the other.
tf2 tools

Now let’s look at how tf2 is being used to create this demo. We can use tf2_tools to look at what tf2 is doing behind the scenes.

1 Using view_frames

view_frames creates a diagram of the frames being broadcast by tf2 over ROS.

```
ros2 run tf2_tools view_frames
```

You will see:

```
Listening to tf data during 5 seconds...
Generating graph in frames.pdf file...
```

Here a tf2 listener is listening to the frames that are being broadcast over ROS and drawing a tree of how the frames are connected. To view the tree, open the resulting frames.pdf with your favorite PDF viewer.

Here we can see three frames that are broadcasted by tf2: world, turtle1, and turtle2. The world here is the parent of the turtle1 and turtle2 frames. view_frames also report some diagnostic information about when the oldest and most recent frame transforms were received and how fast the tf2 frame is published to tf2 for debugging purposes.

2 Using tf2_echo

tf2_echo reports the transform between any two frames broadcasted over ROS.

Usage:

```
ros2 run tf2_ros tf2_echo [source_frame] [target_frame]
```

Let’s look at the transform of the turtle2 frame with respect to turtle1 frame which is equivalent to:
You will see the transform displayed as the `tf2_echo` listener receives the frames broadcasted over ROS 2.

At time 1683385337.850619099
- Translation: [2.157, 0.901, 0.000]
- Rotation: in Quaternion [0.000, 0.000, 0.172, 0.985]
- Rotation: in RPY (radian) [0.000, -0.000, 0.345]
- Rotation: in RPY (degree) [0.000, -0.000, 19.760]
- Matrix:
  
  \[
  \begin{bmatrix}
    0.941 & -0.338 & 0.000 & 2.157 \\
    0.338 & 0.941 & 0.000 & 0.901 \\
    0.000 & 0.000 & 1.000 & 0.000 \\
    0.000 & 0.000 & 0.000 & 1.000 
  \end{bmatrix}
  \]

At time 1683385338.841997774
- Translation: [1.256, 0.216, 0.000]
- Rotation: in Quaternion [0.000, 0.000, -0.016, 1.000]
- Rotation: in RPY (radian) [0.000, 0.000, -0.032]
- Rotation: in RPY (degree) [0.000, 0.000, -1.839]
- Matrix:
  
  \[
  \begin{bmatrix}
    0.999 & 0.032 & 0.000 & 1.256 \\
    -0.032 & 0.999 & -0.000 & 0.216 \\
    -0.000 & 0.000 & 1.000 & 0.000 \\
    0.000 & 0.000 & 0.000 & 1.000 
  \end{bmatrix}
  \]

As you drive your turtle around you will see the transform change as the two turtles move relative to each other.

**rviz and tf2**

`rviz` is a visualization tool that is useful for examining tf2 frames. Let’s look at our turtle frames using rviz. Let’s start rviz with the `turtle_rviz.rviz` configuration file using the `-d` option:

`ros2 run rviz2 rviz2 -d $(ros2 pkg prefix --share turtle_tf2_py)/rviz/turtle_rviz.rviz`
In the side bar you will see the frames broadcasted by tf2. As you drive the turtle around you will see the frames move in rviz.

**Writing a static broadcaster (Python)**

**Goal:** Learn how to broadcast static coordinate frames to tf2.

**Tutorial level:** Intermediate

**Time:** 15 minutes

**Contents**

- **Background**
- **Prerequisites**
- **Tasks**
  - 1 Create a package
  - 2 Write the static broadcaster node
  - 3 Build
  - 4 Run
- **The proper way to publish static transforms**
- **Summary**
Background

Publishing static transforms is useful to define the relationship between a robot base and its sensors or non-moving parts. For example, it is easiest to reason about laser scan measurements in a frame at the center of the laser scanner.

This is a standalone tutorial covering the basics of static transforms, which consists of two parts. In the first part we will write code to publish static transforms to tf2. In the second part we will explain how to use the commandline `static_transform_publisher` executable tool in tf2_ros.

In the next two tutorials we will write the code to reproduce the demo from the Introduction to tf2 tutorial. After that, the following tutorials focus on extending the demo with more advanced tf2 features.

Prerequisites

In previous tutorials, you learned how to create a workspace and create a package.

Tasks

1 Create a package

First we will create a package that will be used for this tutorial and the following ones. The package called learning_tf2_py will depend on geometry_msgs, python3-numpy, rclpy, tf2_ros_py, and turtlesim. Code for this tutorial is stored here.

Open a new terminal and source your ROS 2 installation so that ros2 commands will work. Navigate to workspace's src folder and create a new package:

```bash
ros2 pkg create --build-type ament_python learning_tf2_py
```

Your terminal will return a message verifying the creation of your package `learning_tf2_py` and all its necessary files and folders.

2 Write the static broadcaster node

Let's first create the source files. Inside the src/learning_tf2_py/learning_tf2_py directory download the example static broadcaster code by entering the following command:

Linux

```bash
wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_py/turtle_tf2_py/static_turtle_tf2_broadcaster.py
```

macOS

```bash
wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_py/turtle_tf2_py/static_turtle_tf2_broadcaster.py
```

Windows

In a Windows command line prompt:

```bash
curl -sk https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_py/turtle_tf2_py/static_turtle_tf2_broadcaster.py -o static_turtle_tf2_broadcaster.py
```
Or in powershell:

```bash
curl https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_py/turtle_tf2_py/static_turtle_tf2_broadcaster.py -o static_turtle_tf2_broadcaster.py
```

Open the file using your preferred text editor.

```python
import math
import sys

from geometry_msgs.msg import TransformStamped

import numpy as np

import rclpy
from rclpy.node import Node

from tf2_ros.static_transform_broadcaster import StaticTransformBroadcaster

def quaternion_from_euler(ai, aj, ak):
    ai /= 2.0
    aj /= 2.0
    ak /= 2.0
    ci = math.cos(ai)
    si = math.sin(ai)
    cj = math.cos(aj)
    sj = math.sin(aj)
    ck = math.cos(ak)
    sk = math.sin(ak)
    cc = ci*ck
    cs = ci*sk
    sc = si*ck
    ss = si*sk

    q = np.empty((4, ))
    q[0] = cj*sc - sj*cs
    q[1] = cj*ss + sj*cc
    q[2] = cj*cs - sj*sc
    q[3] = cj*cc + sj*ss

    return q

class StaticFramePublisher(Node):
    ""
    Broadcast transforms that never change.

    This example publishes transforms from `world` to a static turtle frame.
    The transforms are only published once at startup, and are constant for all time.
    ""

    def __init__(self, transformation):
```
(continues on next page)
super().__init__('static_turtle_tf2_broadcaster')

self.tf_static_broadcaster = StaticTransformBroadcaster(self)

# Publish static transforms once at startup
self.make_transforms(transformation)

def make_transforms(self, transformation):
    t = TransformStamped()
    t.header.stamp = self.get_clock().now().to_msg()
    t.header.frame_id = 'world'
    t.child_frame_id = transformation[1]
    t.transform.translation.x = float(transformation[2])
    t.transform.translation.y = float(transformation[3])
    t.transform.translation.z = float(transformation[4])
    quat = quaternion_from_euler(float(transformation[5]), float(transformation[6]), float(transformation[7]))
    t.transform.rotation.x = quat[0]
    t.transform.rotation.y = quat[1]
    t.transform.rotation.z = quat[2]
    t.transform.rotation.w = quat[3]
    self.tf_static_broadcaster.sendTransform(t)

def main():
    logger = rclpy.logging.get_logger('logger')
    
    # obtain parameters from command line arguments
    if len(sys.argv) != 8:
        logger.info('Invalid number of parameters. Usage:

        $ ros2 run learning_tf2_py static_turtle_tf2_broadcaster
        'child_frame_name x y z roll pitch yaw')
        sys.exit(1)

    if sys.argv[1] == 'world':
        logger.info('Your static turtle name cannot be "world"')
        sys.exit(2)

    # pass parameters and initialize node
    rclpy.init()
    node = StaticFramePublisher(sys.argv)
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    rclpy.shutdown()
2.1 Examine the code

Now let’s look at the code that is relevant to publishing the static turtle pose to tf2. The first lines import required packages. First we import the `TransformStamped` from the `geometry_msgs`, which provides us a template for the message that we will publish to the transformation tree.

```python
from geometry_msgs.msg import TransformStamped
```

Afterward, rclpy is imported so its Node class can be used.

```python
import rclpy
from rclpy.node import Node
```

The `tf2_ros` package provides a `StaticTransformBroadcaster` to make the publishing of static transforms easy. To use the `StaticTransformBroadcaster`, we need to import it from the `tf2_ros` module.

```python
from tf2_ros.static_transform_broadcaster import StaticTransformBroadcaster
```

The `StaticFramePublisher` class constructor initializes the node with the name `static_turtle_tf2_broadcaster`. Then, `StaticTransformBroadcaster` is created, which will send one static transformation upon the startup.

```python
self.tf_static_broadcaster = StaticTransformBroadcaster(self)
self.make_transforms(transformation)
```

Here we create a `TransformStamped` object, which will be the message we will send over once populated. Before passing the actual transform values we need to give it the appropriate metadata.

1. We need to give the transform being published a timestamp and we’ll just stamp it with the current time, `self.get_clock().now()`
2. Then we need to set the name of the parent frame of the link we’re creating, in this case `world`
3. Finally, we need to set the name of the child frame of the link we’re creating

```python
t = TransformStamped()
t.header.stamp = self.get_clock().now().to_msg()
t.header.frame_id = 'world'
t.child_frame_id = transformation[1]
```

Here we populate the 6D pose (translation and rotation) of the turtle.

```python
t.transform.translation.x = float(transformation[2])
t.transform.translation.y = float(transformation[3])
t.transform.translation.z = float(transformation[4])
quat = quaternion_from_euler(
    float(transformation[5]), float(transformation[6]), float(transformation[7]))
t.transform.rotation.x = quat[0]
t.transform.rotation.y = quat[1]
t.transform.rotation.z = quat[2]
t.transform.rotation.w = quat[3]
```

Finally, we broadcast static transform using the `sendTransform()` function.
self.tf_static_broadcaster.sendTransform(t)

2.2 Add dependencies

Navigate one level back to the src/learning_tf2_py directory, where the setup.py, setup.cfg, and package.xml files have been created for you.

Open package.xml with your text editor.

As mentioned in the Create a package tutorial, make sure to fill in the <description>, <maintainer> and <license> tags:

```xml
<description>Learning tf2 with rclpy</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache License 2.0</license>
```

After the lines above, add the following dependencies corresponding to your node’s import statements:

```xml
<exec_depend>geometry_msgs</exec_depend>
<exec_depend>python3-numpy</exec_depend>
<exec_depend>rclpy</exec_depend>
<exec_depend>tf2_ros_py</exec_depend>
<exec_depend>turtlesim</exec_depend>
```

This declares the required geometry_msgs, python3-numpy, rclpy, tf2_ros_py, and turtlesim dependencies when its code is executed.

Make sure to save the file.

2.3 Add an entry point

To allow the ros2 run command to run your node, you must add the entry point to setup.py (located in the src/learning_tf2_py directory).

Add the following line between the 'console_scripts': brackets:

```python
'static_turtle_tf2_broadcaster = learning_tf2_py.static_turtle_tf2_broadcaster:main',
```

3 Build

It’s good practice to run rosdep in the root of your workspace to check for missing dependencies before building:

**Linux**

```bash
rosdep install -i --from-path src --rosdistro iron -y
```

**macOS**

rosdep only runs on Linux, so you will need to install geometry_msgs and turtlesim dependencies yourself

**Windows**

rosdep only runs on Linux, so you will need to install geometry_msgs and turtlesim dependencies yourself

Still in the root of your workspace, build your new package:
Linux

```
colcon build --packages-select learning_tf2_py
```

macOS

```
colcon build --packages-select learning_tf2_py
```

Windows

```
colcon build --merge-install --packages-select learning_tf2_py
```

Open a new terminal, navigate to the root of your workspace, and source the setup files:

Linux

```
. install/setup.bash
```

macOS

```
. install/setup.bash
```

Windows

```
# CMD
call install\setup.bat
# Powershell
.\install\setup.ps1
```

### 4 Run

Now run the `static_turtle_tf2_broadcaster` node:

```
ros2 run learning_tf2_py static_turtle_tf2_broadcaster mystaticturtle 0 0 1 0 0 0
```

This sets a turtle pose broadcast for `mystaticturtle` to float 1 meter above the ground.

We can now check that the static transform has been published by echoing the `tf_static` topic

```
ros2 topic echo /tf_static
```

If everything went well you should see a single static transform

```
transforms:
  - header:
      stamp:
        sec: 1622908754
        nanosec: 208515730
      frame_id: world
    child_frame_id: mystaticturtle
  transform:
    translation:
      x: 0.0
```
(continues on next page)
The proper way to publish static transforms

This tutorial aimed to show how `StaticTransformBroadcaster` can be used to publish static transforms. In your real development process you shouldn't have to write this code yourself and should use the dedicated `tf2_ros` tool to do so. `tf2_ros` provides an executable named `static_transform_publisher` that can be used either as a commandline tool or a node that you can add to your launchfiles.

Publish a static coordinate transform to tf2 using an x/y/z offset in meters and roll/pitch/yaw in radians. In our case, roll/pitch/yaw refers to rotation about the x/y/z-axis, respectively.

```
ros2 run tf2_ros static_transform_publisher --x x --y y --z z --yaw yaw --pitch pitch --
  --roll roll --frame-id frame_id --child-frame-id child_frame_id
```

Publish a static coordinate transform to tf2 using an x/y/z offset in meters and quaternion.

```
ros2 run tf2_ros static_transform_publisher --x x --y y --z z --qx qx --qy qy --qz qz --
  --qw qw --frame-id frame_id --child-frame-id child_frame_id
```

`static_transform_publisher` is designed both as a command-line tool for manual use, as well as for use within launch files for setting static transforms. For example:

```
from launch import LaunchDescription
from launch_ros.actions import Node
def generate_launch_description():
    return LaunchDescription([
        Node(
            package='tf2_ros',
            executable='static_transform_publisher',
            arguments = ['-x', '0', '-y', '0', '-z', '0', '--yaw', '0', '--pitch', '0', '--roll', '0', '--frame-id', 'world', '--child-frame-id', 'mystaticturtle'],
        )
    ])
```

Note that all arguments except for `--frame-id` and `--child-frame-id` are optional; if a particular option isn't specified, then the identity will be assumed.
Summary

In this tutorial you learned how static transforms are useful to define static relationships between frames, like mystaticturtle in relation to the world frame. In addition, you learned how static transforms can be useful for understanding sensor data, such as from laser scanners, by relating the data to a common coordinate frame. Finally, you wrote your own node to publish static transforms to tf2 and learned how to publish required static transformations using static_transform_publisher executable and launch files.

Writing a static broadcaster (C++)

Goal: Learn how to broadcast static coordinate frames to tf2.

Tutorial level: Intermediate

Time: 15 minutes

Contents

• Background
• Prerequisites
• Tasks
  – 1 Create a package
  – 2 Write the static broadcaster node
  – 3 Build
  – 4 Run
• The proper way to publish static transforms
• Summary

Background

Publishing static transforms is useful to define the relationship between a robot base and its sensors or non-moving parts. For example, it is easiest to reason about laser scan measurements in a frame at the center of the laser scanner.

This is a standalone tutorial covering the basics of static transforms, which consists of two parts. In the first part we will write code to publish static transforms to tf2. In the second part we will explain how to use the commandline static_transform_publisher executable tool in tf2_ros.

In the next two tutorials we will write the code to reproduce the demo from the Introduction to tf2 tutorial. After that, the following tutorials focus on extending the demo with more advanced tf2 features.
Prerequisites

In previous tutorials, you learned how to create a workspace and create a package.

Tasks

1 Create a package

First we will create a package that will be used for this tutorial and the following ones. The package called learning_tf2_cpp will depend on geometry_msgs, rclcpp, tf2, tf2_ros, and turtlesim. Code for this tutorial is stored here.

Open a new terminal and source your ROS 2 installation so that ros2 commands will work. Navigate to workspace’s src folder and create a new package:

```
ros2 pkg create --build-type ament_cmake --dependencies geometry_msgs rclcpp tf2 tf2_ros turtlesim -- learning_tf2_cpp
```

Your terminal will return a message verifying the creation of your package learning_tf2_cpp and all its necessary files and folders.

2 Write the static broadcaster node

Let’s first create the source files. Inside the src/learning_tf2_cpp/src directory download the example static broadcaster code by entering the following command:

Linux

```
wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/static_turtle_tf2_broadcaster.cpp
```

macOS

```
wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/static_turtle_tf2_broadcaster.cpp
```

Windows

In a Windows command line prompt:

```
curl -sk https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/static_turtle_tf2_broadcaster.cpp -o static_turtle_tf2_broadcaster.cpp
```

Or in powershell:

```
curl https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/static_turtle_tf2_broadcaster.cpp -o static_turtle_tf2_broadcaster.cpp
```

Open the file using your preferred text editor.

```
#include <memory>
#include "geometry_msgs/msg/transform_stamped.hpp"
```

(continues on next page)
```cpp
#include "rclcpp/rclcpp.hpp"
#include "tf2/LinearMath/Quaternion.h"
#include "tf2_ros/static_transform_broadcaster.h"

class StaticFramePublisher : public rclcpp::Node
{
public:
    explicit StaticFramePublisher(char * transformation[]) : Node("static_turtle_tf2_broadcaster")
    {
        tf_static_broadcaster_ = std::make_shared<tf2_ros::StaticTransformBroadcaster>(this);

        // Publish static transforms once at startup
        this->make_transforms(transformation);
    }

private:
    void make_transforms(char * transformation[])
    {
        geometry_msgs::msg::TransformStamped t;

        t.header.stamp = this->get_clock()->now();
        t.header.frame_id = "world";
        t.child_frame_id = transformation[1];

        t.transform.translation.x = atof(transformation[2]);
        t.transform.translation.y = atof(transformation[3]);
        t.transform.translation.z = atof(transformation[4]);
        tf2::Quaternion q;
        q.setRPY(
            atof(transformation[5]),
            atof(transformation[6]),
            atof(transformation[7]));
        t.transform.rotation.x = q.x();
        t.transform.rotation.y = q.y();
        t.transform.rotation.z = q.z();
        t.transform.rotation.w = q.w();

        tf_static_broadcaster_ -> sendTransform(t);
    }

    std::shared_ptr<tf2_ros::StaticTransformBroadcaster> tf_static_broadcaster_;
};

int main(int argc, char * argv[])
{
    auto logger = rclcpp::get_logger("logger");

    // Obtain parameters from command line arguments
    if (argc != 8) {
        RCLCPP_INFO(
            logger, "Invalid number of parameters\usage: ")
    } else {
        // ...
"$ ros2 run learning_tf2_cpp static_turtle_tf2_broadcaster "
  "child_frame_name x y z roll pitch yaw";
  return 1;
}

// As the parent frame of the transform is 'world', it is
// necessary to check that the frame name passed is different
if (strcmp(argv[1], "world") == 0) {
  RCLCPP_INFO(logger, "Your static turtle name cannot be 'world'");
  return 1;
}

// Pass parameters and initialize node
rclcpp::init(argc, argv);
rclcpp::spin(std::make_shared<StaticFramePublisher>(argv));
rclcpp::shutdown();
return 0;
}

2.1 Examine the code

Now let's look at the code that is relevant to publishing the static turtle pose to tf2. The first lines include the required
header files. First we include geometry_msgs/msg/transform_stamped.hpp to access the TransformStamped
message type, which we will publish to the transformation tree.

```cpp
#include "geometry_msgs/msg/transform_stamped.hpp"
```

Afterward, rclcpp is included so its rclcpp::Node class can be used.

```cpp
#include "rclcpp/rclcpp.hpp"
```

`tf2::Quaternion` is a class for a quaternion that provides convenient functions for converting Euler angles
to quaternions and vice versa. We also include tf2_ros/static_transform_broadcaster.h to use the
StaticTransformBroadcaster to make the publishing of static transforms easy.

```cpp
#include "tf2/LinearMath/Quaternion.h"
#include "tf2_ros/static_transform_broadcaster.h"
```

The StaticFramePublisher class constructor initializes the node with the name
static_turtle_tf2_broadcaster. Then, StaticTransformBroadcaster is created, which will send one
static transformation upon the startup.

```cpp
tf_static_broadcaster_ = std::make_shared<tf2_ros::StaticTransformBroadcaster>(this);
this->make_transforms(transformation);
```

Here we create a TransformStamped object, which will be the message we will send once populated. Before
passing the actual transform values we need to give it the appropriate metadata.

1. We need to give the transform being published a timestamp and we'll just stamp it with the current time,
   ```cpp
this->get_clock()->now()
   ```
2. Then we need to set the name of the parent frame of the link we're creating, in this case world
3. Finally, we need to set the name of the child frame of the link we’re creating

```cpp
geometry_msgs::msg::TransformStamped t;

// Set the transformation stamp
t.header.stamp = this->get_clock()->now();

// Set the names of the parent frame and the child frame
// Here we populate the 6D pose (translation and rotation) of the turtle.
// 6D pose is defined by translation and rotation, where translation is a 3D vector
// and rotation is represented by a quaternion.

// Translation
// x, y, and z are the components of the 3D vector
// atof is used to convert the string to a float
// q is a tf2::Quaternion object
// q.setRPY is used to set the rotation in roll-pitch-yaw (R-P-Y) order
// The rotation transformation is applied after the translation transformation

float x = atof(transformation[2]);
float y = atof(transformation[3]);
float z = atof(transformation[4]);

tf2::Quaternion q;
q.setRPY(
  atof(transformation[5]),
  atof(transformation[6]),
  atof(transformation[7]));

// Rotation
// x, y, and z are the components of the quaternion
// tf2::Quaternion::x() method returns the x component of the quaternion
// Similarly, tf2::Quaternion::y(), tf2::Quaternion::z(), and tf2::Quaternion::w() return the y, z, and w components

// Finally, broadcast the static transform using the sendTransform() function.

tf_static_broadcaster_->sendTransform(t);
```

---

2.2 Add dependencies

Navigate one level back to the `src/learning_tf2_cpp` directory, where the `CMakeLists.txt` and `package.xml` files have been created for you.

Open `package.xml` with your text editor.

As mentioned in the `Create a package` tutorial, make sure to fill in the `<description>`, `<maintainer>` and `<license>` tags:

```xml
<description>Learning tf2 with rclcpp</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache License 2.0</license>
```

Make sure to save the file.

2.3 CMakeLists.txt

Add the executable to the `CMakeLists.txt` and name it `static_turtle_tf2_broadcaster`, which you’ll use later with `ros2 run`.

```cpp
add_executable(static_turtle_tf2_broadcaster src/static_turtle_tf2_broadcaster.cpp)
ament_target_dependencies(
  static_turtle_tf2_broadcaster
  geometry_msgs
  rclcpp
)
```
Finally, add the `install(TARGETS...)` section so `ros2 run` can find your executable:

```cpp
install(TARGETS
    static_turtle_tf2_broadcaster
    DESTINATION lib/${PROJECT_NAME})
```

## 3 Build

It's good practice to run `rosdep` in the root of your workspace to check for missing dependencies before building:

Linux

```
rosdep install -i --from-path src --rosdistro iron -y
```

macOS

`rosdep` only runs on Linux, so you will need to install `geometry_msgs` and `turtlesim` dependencies yourself

Windows

`rosdep` only runs on Linux, so you will need to install `geometry_msgs` and `turtlesim` dependencies yourself

Still in the root of your workspace, build your new package:

Linux

```
colcon build --packages-select learning_tf2_cpp
```

macOS

```
colcon build --packages-select learning_tf2_cpp
```

Windows

```
colcon build --merge-install --packages-select learning_tf2_cpp
```

Open a new terminal, navigate to the root of your workspace, and source the setup files:

Linux

```
. install/setup.bash
```

macOS

```
. install/setup.bash
```

Windows

```
# CMD
call install\setup.bat
```
# Powershell

`.\install\setup.ps1`

## 4 Run

Now run the `static_turtle_tf2_broadcaster` node:

```bash
ros2 run learning_tf2_cpp static_turtle_tf2_broadcaster mystaticturtle 0 0 1 0 0 0
```

This sets a turtle pose broadcast for `mystaticturtle` to float 1 meter above the ground.

We can now check that the static transform has been published by echoing the `tf_static` topic

```bash
ros2 topic echo /tf_static
```

If everything went well you should see a single static transform

```
transforms:
- header:
  stamp:
    sec: 1622908754
    nanosec: 20851730
  frame_id: world
  child_frame_id: mystaticturtle
  transform:
    translation:
      x: 0.0
      y: 0.0
      z: 1.0
    rotation:
      x: 0.0
      y: 0.0
      z: 0.0
      w: 1.0
```

### The proper way to publish static transforms

This tutorial aimed to show how `StaticTransformBroadcaster` can be used to publish static transforms. In your real development process you shouldn’t have to write this code yourself and should use the dedicated `tf2_ros` tool to do so. `tf2_ros` provides an executable named `static_transform_publisher` that can be used either as a commandline tool or a node that you can add to your launchfiles.

Publish a static coordinate transform to `tf2` using an x/y/z offset in meters and roll/pitch/yaw in radians. In our case, roll/pitch/yaw refers to rotation about the x/y/z-axis, respectively.

```bash
ros2 run tf2_ros static_transform_publisher --x x --y y --z z --yaw yaw --pitch pitch --
  --roll roll --frame-id frame_id --child-frame-id child_frame_id
```

Publish a static coordinate transform to `tf2` using an x/y/z offset in meters and quaternion.

```bash
ros2 run tf2_ros static_transform_publisher --x x --y y --z z --qx qx --qy qy --qz qz --
  --qw qw --frame-id frame_id --child-frame-id child_frame_id
```

(continues on next page)
static_transform_publisher is designed both as a command-line tool for manual use, as well as for use within launch files for setting static transforms. For example:

```python
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([Node(
        package='tf2_ros',
        executable='static_transform_publisher',
        arguments = ['-x', '0', '-y', '0', '-z', '1', '--yaw', '0', '--pitch', '0', '--roll', '0', '--frame-id', 'world', '--child-frame-id', 'mystaticturtle'],
    ))
```

Note that all arguments except for --frame-id and --child-frame-id are optional; if a particular option isn't specified, then the identity will be assumed.

**Summary**

In this tutorial you learned how static transforms are useful to define static relationships between frames, like mystaticturtle in relation to the world frame. In addition, you learned how static transforms can be useful for understanding sensor data, such as from laser scanners, by relating the data to a common coordinate frame. Finally, you wrote your own node to publish static transforms to tf2 and learned how to publish required static transformations using static_transform_publisher executable and launch files.

**Writing a broadcaster (Python)**

**Goal:** Learn how to broadcast the state of a robot to tf2.

**Tutorial level:** Intermediate

**Time:** 15 minutes
Background

In the next two tutorials we will write the code to reproduce the demo from the *Introduction to tf2* tutorial. After that, following tutorials focus on extending the demo with more advanced tf2 features, including the usage of timeouts in transformation lookups and time travel.

Prerequisites

This tutorial assumes you have a working knowledge of ROS 2 and you have completed the *Introduction to tf2 tutorial*. In previous tutorials, you learned how to *create a workspace* and *create a package*. You also have created the *learning_tf2_py package*, which is where we will continue working from.

Tasks

1 Write the broadcaster node

Let's first create the source files. Go to the *learning_tf2_py* package we created in the previous tutorial. Inside the *src/learning_tf2_py/learning_tf2_py* directory download the example broadcaster code by entering the following command:

Linux

```bash
wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_py/turtle_tf2_broadcaster.py
```

macOS

```bash
wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_py/turtle_tf2_broadcaster.py
```

Windows

In a Windows command line prompt:

```bash
curl -sk https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_py/turtle_tf2_broadcaster.py -o turtle_tf2_broadcaster.py
```

Or in powershell:

```bash
curl https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_py/turtle_tf2_broadcaster.py -o turtle_tf2_broadcaster.py
```

Open the file using your preferred text editor.

```python
import math

from geometry_msgs.msg import TransformStamped

import numpy as np

import rclpy

from rclpy.node import Node
```

(continues on next page)
from tf2_ros import TransformBroadcaster
from turtlesim.msg import Pose

def quaternion_from_euler(ai, aj, ak):
    ai /= 2.0
    aj /= 2.0
    ak /= 2.0
    ci, ci = math.cos(ai)
    si, si = math.sin(ai)
    cj, cj = math.cos(aj)
    sj, sj = math.sin(aj)
    ck, ck = math.cos(ak)
    sk, sk = math.sin(ak)
    cc = ci*ck
    cs = ci*sk
    sc = si*ck
    ss = si*sk

    q = np.empty((4, ))
    q[0] = cj*sc - sj*cs
    q[1] = cj*ss + sj*cc
    q[2] = cj*cs - sj*sc
    q[3] = cj*cc + sj*ss

    return q

class FramePublisher(Node):
    def __init__(self):
        super().__init__('
turtle_tf2_frame_publisher'')

        # Declare and acquire 'turtlename' parameter
        self.turtlename = self.declare_parameter('
turtle'.get_parameter_value().string_value

        # Initialize the transform broadcaster
        self.tf_broadcaster = TransformBroadcaster(self)

        # Subscribe to a turtle1/2/pose topic and call handle_turtle_pose
        # callback function on each message
        self.subscription = self.create_subscription(
            Pose,
            f'/{self.turtlename}/pose',
            self.handle_turtle_pose,
            1)

        self.subscription # prevent unused variable warning

    def handle_turtle_pose(self, msg):
(continues on next page)
t = TransformStamped()

    # Read message content and assign it to
    # corresponding tf variables
    t.header.stamp = self.get_clock().now().to_msg()
    t.header.frame_id = 'world'
    t.child_frame_id = self.turtlename

    # Turtle only exists in 2D, thus we get x and y translation
    # coordinates from the message and set the z coordinate to 0
    t.transform.translation.x = msg.x
    t.transform.translation.y = msg.y
    t.transform.translation.z = 0.0

    # For the same reason, turtle can only rotate around one axis
    # and this why we set rotation in x and y to 0 and obtain
    # rotation in z axis from the message
    q = quaternion_from_euler(0, 0, msg.theta)
    t.transform.rotation.x = q[0]
    t.transform.rotation.y = q[1]
    t.transform.rotation.z = q[2]
    t.transform.rotation.w = q[3]

    # Send the transformation
self.tf_broadcaster.sendTransform(t)

def main():
    rclpy.init()
    node = FramePublisher()  
    try:
        rclpy.spin(node)
    except KeyboardInterupt:
        pass
    rclpy.shutdown()

1.1 Examine the code

Now, let's take a look at the code that is relevant to publishing the turtle pose to tf2. Firstly, we define and acquire a
single parameter turtlename, which specifies a turtle name, e.g. turtle1 or turtle2.

self.turtlename = self.declare_parameter(
    'turtlename', 'turtle').get_parameter_value().string_value

Afterward, the node subscribes to topic turtleX/pose and runs function handle_turtle_pose on every incoming
message.

self.subscription = self.create_subscription(
    Pose,
    f'/{self.turtlename}/pose',
    (continues on next page)
Now, we create a TransformStamped object and give it the appropriate metadata.

1. We need to give the transform being published a timestamp, and we'll just stamp it with the current time by calling `self.get_clock().now()`. This will return the current time used by the Node.
2. Then we need to set the name of the parent frame of the link we’re creating, in this case `world`.
3. Finally, we need to set the name of the child node of the link we’re creating, in this case this is the name of the turtle itself.

The handler function for the turtle pose message broadcasts this turtle’s translation and rotation, and publishes it as a transform from frame `world` to frame `turtleX`.

```python
self.handle_turtle_pose,

1)
```

Here we copy the information from the 3D turtle pose into the 3D transform.

```python
t = TransformStamped()

# Read message content and assign it to
# corresponding tf variables
t.header.stamp = self.get_clock().now().to_msg()
t.header.frame_id = 'world'
t.child_frame_id = self.turtlename
```

Finally we take the transform that we constructed and pass it to the `sendTransform` method of the `TransformBroadcaster` that will take care of broadcasting.

```python
# Send the transformation
self.tf_broadcaster.sendTransform(t)
```

Note: You can also publish static transforms with the same pattern by instantiating a `tf2_ros.StaticTransformBroadcaster` instead of a `tf2_ros.TransformBroadcaster`. The static transforms will be published on the `/tf_static` topic and will be sent only when required, not periodically. For more details see [here](https://...).
1.2 Add an entry point

To allow the ros2 run command to run your node, you must add the entry point to setup.py (located in the src/learning_tf2_py directory).

Finally, add the following line between the 'console_scripts': brackets:

```
'turtle_tf2_broadcaster = learning_tf2_py.turtle_tf2_broadcaster:main',
```

2 Write the launch file

Now create a launch file for this demo. With your text editor, create a new file called turtle_tf2_demo_launch.py in the launch folder, and add the following lines:

```python
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='turtlesim',
            executable='turtlesim_node',
            name='sim',
        ),
        Node(
            package='learning_tf2_py',
            executable='turtle_tf2_broadcaster',
            name='broadcaster1',
            parameters=[
                {'turtlename': 'turtle1'}
            ],
        ),
    ])
```

2.1 Examine the code

First we import required modules from the launch and launch_ros packages. It should be noted that launch is a generic launching framework (not ROS 2 specific) and launch_ros has ROS 2 specific things, like nodes that we import here.

```python
from launch import LaunchDescription
from launch_ros.actions import Node

Now we run our nodes that start the turtlesim simulation and broadcast turtle1 state to the tf2 using our turtle_tf2_broadcaster node.

Node(
    package='turtlesim',
    executable='turtlesim_node',
    name='sim'
)```

(continues on next page)
2.2 Add dependencies

Navigate one level back to the src/learning_tf2_py directory, where the setup.py, setup.cfg, and package.xml files are located.

Open package.xml with your text editor. Add the following dependencies corresponding to your launch file's import statements:

```xml
<exec_depend>launch</exec_depend>
<exec_depend>launch_ros</exec_depend>
```

This declares the additional required launch and launch_ros dependencies when its code is executed.

Make sure to save the file.

2.3 Update setup.py

Reopen setup.py and add the line so that the launch files from the launch/ folder would be installed. The data_files field should now look like this:

```python
data_files=[
        ...
        (os.path.join('share', package_name, 'launch'),
         glob(os.path.join('launch', '*launch.*'))),
        ],
```

Also add the appropriate imports at the top of the file:

```python
import os
from glob import glob
```

You can learn more about creating launch files in this tutorial.
3 Build

Run `rosdep` in the root of your workspace to check for missing dependencies.

Linux

```
rosdep install -i --from-path src --rosdistro iron -y
```

macOS

`rosdep` only runs on Linux, so you will need to install `geometry_msgs` and `turtlesim` dependencies yourself.

Windows

`rosdep` only runs on Linux, so you will need to install `geometry_msgs` and `turtlesim` dependencies yourself.

Still in the root of your workspace, build your package:

Linux

```
colcon build --packages-select learning_tf2_py
```

macOS

```
colcon build --packages-select learning_tf2_py
```

Windows

```
colcon build --merge-install --packages-select learning_tf2_py
```

Open a new terminal, navigate to the root of your workspace, and source the setup files:

Linux

```
. install/setup.bash
```

macOS

```
. install/setup.bash
```

Windows

```
# CMD
call install\setup.bat
# Powershell
.\install\setup.ps1
```
4 Run

Now run the launch file that will start the turtlesim simulation node and `turtle_tf2_broadcaster` node:

```
ros2 launch learning_tf2_py turtle_tf2_demo_launch.py
```

In the second terminal window type the following command:

```
ros2 run turtlesim turtle_teleop_key
```

You will now see that the turtlesim simulation have started with one turtle that you can control.

Now, use the `tf2_echo` tool to check if the turtle pose is actually getting broadcast to tf2:

```
ros2 run tf2_ros tf2_echo world turtle1
```

This should show you the pose of the first turtle. Drive around the turtle using the arrow keys (make sure your `turtle_teleop_key` terminal window is active, not your simulator window). In your console output you will see something similar to this:

```
At time 1625137663.912474878
  Translation: [5.276, 7.930, 0.000]
```

(continues on next page)
- Rotation: in Quaternion [0.000, 0.000, 0.934, -0.357]  
At time 1625137664.950813527  
- Translation: [3.750, 6.563, 0.000]  
- Rotation: in Quaternion [0.000, 0.000, 0.934, -0.357]  
At time 1625137665.906280726  
- Translation: [2.320, 5.282, 0.000]  
- Rotation: in Quaternion [0.000, 0.000, 0.934, -0.357]  
At time 1625137666.850775673  
- Translation: [2.153, 5.133, 0.000]  
- Rotation: in Quaternion [0.000, 0.000, -0.365, 0.931]  

If you run `tf2_echo` for the transform between the `world` and `turtle2`, you should not see a transform, because the second turtle is not there yet. However, as soon as we add the second turtle in the next tutorial, the pose of `turtle2` will be broadcast to tf2.

**Summary**

In this tutorial you learned how to broadcast the pose of the robot (position and orientation of the turtle) to tf2 and how to use the `tf2_echo` tool. To actually use the transforms broadcasted to tf2, you should move on to the next tutorial about creating a `tf2` listener.

**Writing a broadcaster (C++)**

**Goal:** Learn how to broadcast the state of a robot to tf2.  
**Tutorial level:** Intermediate  
**Time:** 15 minutes

**Contents**

- Background  
- Prerequisites  
- Tasks  
  - 1 Write the broadcaster node  
  - 2 Write the launch file  
  - 3 Build  
  - 4 Run  
- Summary
Background

In the next two tutorials we will write the code to reproduce the demo from the *Introduction to tf2* tutorial. After that, following tutorials focus on extending the demo with more advanced tf2 features, including the usage of timeouts in transformation lookups and time travel.

Prerequisites

This tutorial assumes you have a working knowledge of ROS 2 and you have completed the *Introduction to tf2* tutorial and *tf2 static broadcaster tutorial* (C++). In previous tutorials, you learned how to create a workspace and create a package. You also have created the `learning_tf2_cpp` package, which is where we will continue working from.

Tasks

1 Write the broadcaster node

Let’s first create the source files. Go to the `learning_tf2_cpp` package we created in the previous tutorial. Inside the `src` directory download the example broadcaster code by entering the following command:

Linux

```bash
wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/
   → turtle_tf2_broadcaster.cpp
```

macOS

```bash
wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/
   → turtle_tf2_broadcaster.cpp
```

Windows

In a Windows command line prompt:

```bash
curl -sk https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/
   → src/turtle_tf2_broadcaster.cpp -o turtle_tf2_broadcaster.cpp
```

Or in powershell:

```bash
curl https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/
   → turtle_tf2_broadcaster.cpp -o turtle_tf2_broadcaster.cpp
```

Open the file using your preferred text editor.

```cpp
#include <functional>
#include <memory>
#include <sstream>
#include <string>
#include "geometry_msgs/msg/transform_stamped.hpp"
#include "rclcpp/rclcpp.hpp"
#include "tf2/LinearMath/Quaternion.h"
#include "tf2_ros/transform_broadcaster.h"
```

(continues on next page)
```cpp
#include "turtlesim/msg/pose.hpp"

class FramePublisher : public rclcpp::Node
{
public:
    FramePublisher()
        : Node("turtle_tf2_frame_publisher")
    {
        // Declare and acquire `turtlename` parameter
        turtlename_ = this->declare_parameter<std::string>("turtlename", "turtle");

        // Initialize the transform broadcaster
        tf_broadcaster_ = std::make_unique<tf2_ros::TransformBroadcaster>(^this);

        // Subscribe to a turtle{1}{2}/pose topic and call handle_turtle_pose
        // callback function on each message
        std::ostringstream stream;
        stream << "/" << turtlename_.c_str() << "/pose";
        std::string topic_name = stream.str();
        subscription_ = this->create_subscription<turtlesim::msg::Pose>(
            topic_name, 10,
            std::bind(&FramePublisher::handle_turtle_pose, this, std::placeholders::_1));
    }

private:
    void handle_turtle_pose(const std::shared_ptr<turtlesim::msg::Pose> msg)
    {
        geometry_msgs::msg::TransformStamped t;

        // Read message content and assign it to corresponding tf variables
        t.header.stamp = this->get_clock()->now();
        t.header.frame_id = "world";
        t.child_frame_id = turtlename_.c_str();

        // Turtle only exists in 2D, thus we get x and y translation
        // coordinates from the message and set the z coordinate to 0
        t.transform.translation.x = msg->x;
        t.transform.translation.y = msg->y;
        t.transform.translation.z = 0.0;

        // For the same reason, turtle can only rotate around one axis
        // and why we set rotation in x and y to 0 and obtain
        // rotation in z axis from the message
        tf2::Quaternion q;
        q.setRPY(0, 0, msg->theta);
        t.transform.rotation.x = q.x();
        t.transform.rotation.y = q.y();
        t.transform.rotation.z = q.z();
        t.transform.rotation.w = q.w();
    }
};
```
// Send the transformation
    tf_broadcaster_->sendTransform(t);
}

rclcpp::Subscription<turtlesim::msg::Pose>::SharedPtr subscription_
    = this->create_subscription<turtlesim::msg::Pose>(
        std::bind(&FramePublisher::handle_turtle_pose,
            this, _1));

1.1 Examine the code

Now, let's take a look at the code that is relevant to publishing the turtle pose to tf2. Firstly, we define and acquire a single parameter turtlename, which specifies a turtle name, e.g. turtle1 or turtle2.

turtlename_ = this->declare_parameter<std::string>("turtlename", "turtle");

Afterward, the node subscribes to topic turtleX/pose and runs function handle_turtle_pose on every incoming message.

subscription_ = this->create_subscription<turtlesim::msg::Pose>(
    topic_name, 10,
    std::bind(&FramePublisher::handle_turtle_pose, this, _1));

Now, we create a TransformStamped object and give it the appropriate metadata.

1. We need to give the transform being published a timestamp, and we'll just stamp it with the current time by calling this->get_clock()->now(). This will return the current time used by the Node.
2. Then we need to set the name of the parent frame of the link we're creating, in this case world.
3. Finally, we need to set the name of the child node of the link we're creating, in this case this is the name of the turtle itself.

The handler function for the turtle pose message broadcasts this turtle’s translation and rotation, and publishes it as a transform from frame world to frame turtleX.

gEometry_msgs::msg::TransformStamped t;

// Read message content and assign it to corresponding tf variables
t.header.stamp = this->get_clock()->now();
t.header.frame_id = "world";
t.child_frame_id = turtlename_.c_str();

Here we copy the information from the 3D turtle pose into the 3D transform.
// Turtle only exists in 2D, thus we get x and y translation
// coordinates from the message and set the z coordinate to 0
  t.transform.translation.x = msg->x;
  t.transform.translation.y = msg->y;
  t.transform.translation.z = 0.0;

// For the same reason, turtle can only rotate around one axis
// and this why we set rotation in x and y to 0 and obtain
// rotation in z axis from the message
  tf2::Quaternion q;
  q.setRPY(0, 0, msg->theta);
  t.transform.rotation.x = q.x();
  t.transform.rotation.y = q.y();
  t.transform.rotation.z = q.z();
  t.transform.rotation.w = q.w();

Finally we take the transform that we constructed and pass it to the sendTransform method of the TransformBroadcaster that will take care of broadcasting.

// Send the transformation
  tf_broadcaster_->sendTransform(t);

1.2 CMakeLists.txt

Navigate one level back to the learning_tf2_cpp directory, where the CMakeLists.txt and package.xml files are located.

Now open the CMakeLists.txt add the executable and name it turtle_tf2_broadcaster, which you’ll use later with ros2 run.

add_executable(turtle_tf2_broadcaster src/turtle_tf2_broadcaster.cpp)
ament_target_dependencies(
  turtle_tf2_broadcaster
  geometry_msgs
  rclcpp
  tf2
  tf2_ros
  turtlesim
)

Finally, add the install(TARGETS...) section so ros2 run can find your executable:

install(TARGETS
  turtle_tf2_broadcaster
  DESTINATION lib/${PROJECT_NAME})
2 Write the launch file

Now create a launch file for this demo. With your text editor, create a new file called `turtle_tf2_demo_launch.py` in the `launch` folder, and add the following lines:

```python
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='turtlesim',
            executable='turtlesim_node',
            name='sim',
        ),
        Node(
            package='learning_tf2_cpp',
            executable='turtle_tf2_broadcaster',
            name='broadcaster1',
            parameters=[
                {'turtlename': 'turtle1'}
            ],
        ),
    ])
```

2.1 Examine the code

First we import required modules from the `launch` and `launch_ros` packages. It should be noted that `launch` is a generic launching framework (not ROS 2 specific) and `launch_ros` has ROS 2 specific things, like nodes that we import here.

```python
from launch import LaunchDescription
from launch_ros.actions import Node
```

Now we run our nodes that start the turtlesim simulation and broadcast `turtle1` state to the tf2 using our `turtle_tf2_broadcaster` node.

```python
Node(
    package='turtlesim',
    executable='turtlesim_node',
    name='sim',
),
Node(
    package='learning_tf2_cpp',
    executable='turtle_tf2_broadcaster',
    name='broadcaster1',
    parameters=[
        {'turtlename': 'turtle1'}
    ],
)
```
2.2 Add dependencies

Navigate one level back to the learning_tf2_cpp directory, where the CMakeLists.txt and package.xml files are located.

Open package.xml with your text editor. Add the following dependencies corresponding to your launch file's import statements:

```xml
<exec_depend>launch</exec_depend>
<exec_depend>launch_ros</exec_depend>
```

This declares the additional required launch and launch_ros dependencies when its code is executed.

Make sure to save the file.

2.3 CMakeLists.txt

Reopen CMakeLists.txt and add the line so that the launch files from the launch/ folder would be installed.

```markdown
install(DIRECTORY launch
       DESTINATION share/${PROJECT_NAME})
```

You can learn more about creating launch files in this tutorial.

3 Build

Run rosdep in the root of your workspace to check for missing dependencies.

Linux

```bash
rosdep install -i --from-path src --rosdistro iron -y
```

macOS

rosdep only runs on Linux, so you will need to install geometry_msgs and turtlesim dependencies yourself

Windows

rosdep only runs on Linux, so you will need to install geometry_msgs and turtlesim dependencies yourself

From the root of your workspace, build your updated package:

Linux

```bash
colcon build --packages-select learning_tf2_cpp
```

macOS

```bash
colcon build --packages-select learning_tf2_cpp
```

Windows

```bash
colcon build --merge-install --packages-select learning_tf2_cpp
```

Open a new terminal, navigate to the root of your workspace, and source the setup files:

Linux
macOS

```
. install/setup.bash
```

Windows

```
# CMD
call install\setup.bat

# Powershell
.\install\setup.ps1
```

## 4 Run

Now run the launch file that will start the turtlesim simulation node and `turtle_tf2_broadcaster` node:

```
ros2 launch learning_tf2_cpp turtle_tf2_demo.launch.py
```

In the second terminal window type the following command:

```
ros2 run turtlesim turtle_teleop_key
```

You will now see that the turtlesim simulation have started with one turtle that you can control.
Now, use the `tf2_echo` tool to check if the turtle pose is actually getting broadcast to tf2:

```
ros2 run tf2_ros tf2_echo world turtle1
```

This should show you the pose of the first turtle. Drive around the turtle using the arrow keys (make sure your `turtle_teleop_key` terminal window is active, not your simulator window). In your console output you will see something similar to this:

```
At time 1625137663.912474878
- Translation: [5.276, 7.930, 0.000]
- Rotation: in Quaternion [0.000, 0.000, 0.934, -0.357]
At time 1625137664.950813527
- Translation: [3.750, 6.563, 0.000]
- Rotation: in Quaternion [0.000, 0.000, 0.934, -0.357]
At time 1625137665.906280726
- Translation: [2.320, 5.282, 0.000]
- Rotation: in Quaternion [0.000, 0.000, 0.934, -0.357]
At time 1625137666.850775673
- Translation: [2.153, 5.133, 0.000]
- Rotation: in Quaternion [0.000, 0.000, -0.365, 0.931]
```

If you run `tf2_echo` for the transform between the `world` and `turtle2`, you should not see a transform, because the
second turtle is not there yet. However, as soon as we add the second turtle in the next tutorial, the pose of turtle2 will be broadcast to tf2.

**Summary**

In this tutorial you learned how to broadcast the pose of the robot (position and orientation of the turtle) to tf2 and how to use the `tf2_echo` tool. To actually use the transforms broadcasted to tf2, you should move on to the next tutorial about creating a *tf2 listener*.

**Writing a listener (Python)**

**Goal:** Learn how to use tf2 to get access to frame transformations.

**Tutorial level:** Intermediate

**Time:** 10 minutes

**Contents**

- **Background**
- **Prerequisites**
- **Tasks**
  - 1 Write the listener node
  - 2 Update the launch file
  - 3 Build
  - 4 Run
- **Summary**

**Background**

In previous tutorials we created a tf2 broadcaster to publish the pose of a turtle to tf2.

In this tutorial we’ll create a tf2 listener to start using tf2.

**Prerequisites**

This tutorial assumes you have completed the *tf2 broadcaster tutorial (Python)*. In the previous tutorial, we created a `learning_tf2_py` package, which is where we will continue working from.
Tasks

1 Write the listener node

Let’s first create the source files. Go to the `learning_tf2_py` package we created in the previous tutorial. Inside the `src/learning_tf2_py/learning_tf2_py` directory download the example listener code by entering the following command:

**Linux**

```
wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_py/turtle_tf2_py/turtle_tf2_listener.py
```

**macOS**

```
wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_py/turtle_tf2_py/turtle_tf2_listener.py
```

**Windows**

In a Windows command line prompt:

```
curl -sk https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_py/turtle_tf2_py/turtle_tf2_listener.py -o turtle_tf2_listener.py
```

Or in powershell:

```
curl https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_py/turtle_tf2_py/turtle_tf2_listener.py -o turtle_tf2_listener.py
```

Open the file using your preferred text editor.

```python
import math
from geometry_msgs.msg import Twist
import rclpy
from rclpy.node import Node
from tf2_ros import TransformException
from tf2_ros.buffer import Buffer
from tf2_ros.transform_listener import TransformListener
from turtlesim.srv import Spawn

class FrameListener(Node):
    def __init__(self):
        super().__init__('turtle_tf2_frame_listener')

        # Declare and acquire `target_frame` parameter
        self.target_frame = self.declare_parameter('target_frame', 'turtle1').get_parameter_value().string_value
```

(continues on next page)
self.tf_buffer = Buffer()
self.tf_listener = TransformListener(self.tf_buffer, self)

# Create a client to spawn a turtle
self.spawner = self.create_client(Spawn, 'spawn')
# Boolean values to store the information
# if the service for spawning turtle is available
self.turtle_spawning_service_ready = False
# if the turtle was successfully spawned
self.turtle_spawned = False

# Create turtle2 velocity publisher
self.publisher = self.create_publisher(Twist, 'turtle2/cmd_vel', 1)

# Call on_timer function every second
self.timer = self.create_timer(1.0, self.on_timer)

def on_timer(self):
    # Store frame names in variables that will be used to
    # compute transformations
    from_frame_rel = self.target_frame
    to_frame_rel = 'turtle2'

    if self.turtle_spawning_service_ready:
        if self.turtle_spawned:
            # Look up for the transformation between target_frame and turtle2 frames
            # and send velocity commands for turtle2 to reach target_frame
            try:
                t = self.tf_buffer.lookup_transform(
                    to_frame_rel,
                    from_frame_rel,
                    rclpy.time.Time())
            except TransformException as ex:
                self.get_logger().info(f'Could not transform {to_frame_rel} to {from_frame_rel}: {ex}'),
            return

            msg = Twist()
            scale_rotation_rate = 1.0
            msg.angular.z = scale_rotation_rate * math.atan2(  
                t.transform.translation.y,  
                t.transform.translation.x)

            scale_forward_speed = 0.5
            msg.linear.x = scale_forward_speed * math.sqrt(  
                t.transform.translation.x ** 2 +  
                t.transform.translation.y ** 2)

            self.publisher.publish(msg)
        else:
            if self.result.done():
(continues on next page)
self.get_logger().info(f'Successfully spawned {self.result.result().name}')
self.turtle_spawned = True
else:
    self.get_logger().info('Spawn is not finished')
else:
    if self.spawner.service_is_ready():
        # Initialize request with turtle name and coordinates
        # Note that x, y and theta are defined as floats in turtlesim/srv/Spawn
        request = Spawn.Request()
        request.name = 'turtle2'
        request.x = float(4)
        request.y = float(2)
        request.theta = float(0)
        # Call request
        self.result = self.spawner.call_async(request)
        self.turtle_spawning_service_ready = True
    else:
        # Check if the service is ready
        self.get_logger().info('Service is not ready')

def main():
    rclpy.init()
    node = FrameListener()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    rclpy.shutdown()

1.1 Examine the code

To understand how the service behind spawning turtle works, please refer to writing a simple service and client (Python) tutorial.

Now, let’s take a look at the code that is relevant to get access to frame transformations. The tf2_ros package provides an implementation of a TransformListener to help make the task of receiving transforms easier.

```python
from tf2_ros.transform_listener import TransformListener
```

Here, we create a TransformListener object. Once the listener is created, it starts receiving tf2 transformations over the wire, and buffers them for up to 10 seconds.

```python
self.tf_listener = TransformListener(self.tf_buffer, self)
```

Finally, we query the listener for a specific transformation. We call lookup_transform method with following arguments:

1. Target frame
2. Source frame
3. The time at which we want to transform

Providing `rclpy.time.Time()` will just get us the latest available transform. All this is wrapped in a try-except block to handle possible exceptions.

```python
t = self.tf_buffer.lookup_transform(  
    to_frame_rel,  
    from_frame_rel,  
    rclpy.time.Time())
```

### 1.2 Add an entry point

To allow the `ros2 run` command to run your node, you must add the entry point to `setup.py` (located in the `src/learning_tf2_py` directory).

```python
'turtle_tf2_listener = learning_tf2_py.turtle_tf2_listener:main',
```

### 2 Update the launch file

Open the launch file called `turtle_tf2_demo_launch.py` with your text editor, add two new nodes to the launch description, add a launch argument, and add the imports. The resulting file should look like:

```python
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch.substitutions import LaunchConfiguration
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([  
        Node(  
            package='turtlesim',  
            executable='turtlesim_node',  
            name='sim'  
        ),  
        Node(  
            package='learning_tf2_py',  
            executable='turtle_tf2_broadcaster',  
            name='broadcaster1',  
            parameters=[  
                {'turtlename': 'turtle1'}  
            ]  
        ),  
        DeclareLaunchArgument(  
            'target_frame', default_value='turtle1',  
            description='Target frame name.'  
        ),  
        Node(  
            package='learning_tf2_py',  
            executable='turtle_tf2_broadcaster',
```
This will declare a `target_frame` launch argument, start a broadcaster for second turtle that we will spawn and listener that will subscribe to those transformations.

### 3 Build

Run `rosdep` in the root of your workspace to check for missing dependencies.

**Linux**

```bash
rosdep install -i --from-path src --rosdistro iron -y
```

**macOS**

`rosdep` only runs on Linux, so you will need to install `geometry_msgs` and `turtlesim` dependencies yourself.

**Windows**

`rosdep` only runs on Linux, so you will need to install `geometry_msgs` and `turtlesim` dependencies yourself.

Still in the root of your workspace, build your package:

**Linux**

```bash
colcon build --packages-select learning_tf2_py
```

**macOS**

```bash
colcon build --packages-select learning_tf2_py
```

**Windows**

```bash
colcon build --merge-install --packages-select learning_tf2_py
```

Open a new terminal, navigate to the root of your workspace, and source the setup files:

**Linux**

```bash
.
install/setup.bash
```

**macOS**
4 Run

Now you’re ready to start your full turtle demo:

```bash
ros2 launch learning_tf2_py turtle_tf2_demo_launch.py
```

You should see the turtle sim with two turtles. In the second terminal window type the following command:

```bash
ros2 run turtlesim turtle_teleop_key
```

To see if things work, simply drive around the first turtle using the arrow keys (make sure your terminal window is active, not your simulator window), and you’ll see the second turtle following the first one!

Summary

In this tutorial you learned how to use tf2 to get access to frame transformations. You also have finished writing your own turtlesim demo that you first tried in Introduction to tf2 tutorial.

Writing a listener (C++)

**Goal:** Learn how to use tf2 to get access to frame transformations.

**Tutorial level:** Intermediate

**Time:** 10 minutes

**Contents**

- **Background**
- **Prerequisites**
- **Tasks**
  - 1 Write the listener node
  - 2 Update the launch file
  - 3 Build
  - 4 Run
- **Summary**
Background

In previous tutorials we created a tf2 broadcaster to publish the pose of a turtle to tf2.

In this tutorial we’ll create a tf2 listener to start using tf2.

Prerequisites

This tutorial assumes you have completed the tf2 static broadcaster tutorial (C++) and the tf2 broadcaster tutorial (C++). In the previous tutorial, we created a learning_tf2_cpp package, which is where we will continue working from.

Tasks

1 Write the listener node

Let’s first create the source files. Go to the learning_tf2_cpp package we created in the previous tutorial. Inside the src directory download the example listener code by entering the following command:

Linux

```bash
wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/turtle_tf2_listener.cpp
```

macOS

```bash
wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/turtle_tf2_listener.cpp
```

Windows

In a Windows command line prompt:

```bash
curl -sk https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/turtle_tf2_listener.cpp -o turtle_tf2_listener.cpp
```

Or in powershell:

```bash
curl https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/turtle_tf2_listener.cpp -o turtle_tf2_listener.cpp
```

Open the file using your preferred text editor.

```cpp
#include <chrono>
#include <functional>
#include <memory>
#include <string>
#include "geometry_msgs/msg/transform_stamped.hpp"
#include "geometry_msgs/msg/twist.hpp"
#include "rclcpp/rclcpp.hpp"
#include "tf2/exceptions.h"
#include "tf2_ros/transform_listener.h"
```

(continues on next page)
```cpp
#include "tf2_ros/buffer.h"
#include "turtlesim/srv/spawn.hpp"

using namespace std::chrono_literals;

class FrameListener : public rclcpp::Node {
public:
    FrameListener()
    : Node("turtle_tf2_frame_listener"),
      turtle_spawning_service_ready_(false),
      turtle_spawned_(false)
    {
        // Declare and acquire 'target_frame' parameter
        target_frame_ = this->declare_parameter<std::string>("target_frame", "turtle1");

        tf_buffer_ = std::make_unique<tf2_ros::Buffer>(this->get_clock());
        tf_listener_ = std::make_shared<tf2_ros::TransformListener>(*tf_buffer_);

        // Create a client to spawn a turtle
        spawner_ = this->create_client<turtlesim::srv::Spawn>("spawn");

        // Create turtle2 velocity publisher
        publisher_ = this->create_publisher<geometry_msgs::msg::Twist>("turtle2/cmd_vel", 1);

        // Call on_timer function every second
        timer_ = this->create_wall_timer(1s, std::bind(&FrameListener::on_timer, this));
    }

private:
    void on_timer() {
        std::string fromFrameRel = target_frame_.c_str();
        std::string toFrameRel = "turtle2";

        if (turtle_spawning_service_ready_) {
            if (turtle_spawned_) {
                geometry_msgs::msg::TransformStamped t;

                // Look up for the transformation between target_frame and turtle2 frames
                // and send velocity commands for turtle2 to reach target_frame
                try {
                    t = tf_buffer_->lookupTransform(
                        toFrameRel, fromFrameRel,
                        tf2::TimePointZero);
                }
            }
        }
    }
};
```
} catch (const tf2::TransformException & ex) {
    RCLCPP_INFO(
        this->get_logger(), "Could not transform %s to %s: %s",
        toFrameRel.c_str(), fromFrameRel.c_str(), ex.what());
    return;
}

geometry_msgs::msg::Twist msg;
static const double scaleRotationRate = 1.0;
msg.angular.z = scaleRotationRate * atan2(t.transform.translation.y,
    t.transform.translation.x);

static const double scaleForwardSpeed = 0.5;
msg.linear.x = scaleForwardSpeed * sqrt(pow(t.transform.translation.x, 2) +
    pow(t.transform.translation.y, 2));

publisher_->publish(msg);

else {
    RCLCPP_INFO(
        this->get_logger(), "Successfully spawned");
turtle_spawned_ = true;
}
}
else {
    // Check if the service is ready
    if (spawner_->service_is_ready()) {
        // Initialize request with turtle name and coordinates
        // Note that x, y and theta are defined as floats in turtlesim/srv/Spawn
        auto request = std::make_shared<turtlesim::srv::Spawn::Request>();
        request->x = 4.0;
        request->y = 2.0;
        request->theta = 0.0;
        request->name = "turtle2";

        // Call request
        using ServiceResponseFuture = rclcpp::Client<turtlesim::srv::Spawn>::SharedFuture;
        auto response_received_callback = [this](ServiceResponseFuture future) {
            auto result = future.get();
            if (strcmp(result->name.c_str(), "turtle2") == 0) {
                turtle_spawning_service_ready_ = true;
            } else {
                RCLCPP_ERROR(
                    this->get_logger(), "Service callback result mismatch");
            }
        };
        auto result = spawner_->async_send_request(request, response_received_callback);
    } else {
        RCLCPP_INFO(
            this->get_logger(), "Service is not ready");
    }
}
// Boolean values to store the information
// if the service for spawning turtle is available
bool turtle_spawning_service_ready_;  
// if the turtle was successfully spawned
bool turtle_spawned_;  
rclcpp::Client<turtlesim::srv::Spawn>::SharedPtr spawner_{nullptr};  
rclcpp::TimerBase::SharedPtr timer_{nullptr};  
rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr publisher_{nullptr};  
std::shared_ptr<tf2_ros::TransformListener> tf_listener_{nullptr};  
std::unique_ptr<tf2_ros::Buffer> tf_buffer_;  
std::string target_frame_;  
};  

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);  
rclcpp::spin(std::make_shared<FrameListener>());  
rclcpp::shutdown();  
  return 0;
}

1.1 Examine the code

To understand how the service behind spawning turtle works, please refer to writing a simple service and client (C++) tutorial.

Now, let’s take a look at the code that is relevant to get access to frame transformations. The tf2_ros contains a TransformListener header file implementation that makes the task of receiving transforms easier.

```
#include "tf2_ros/transform_listener.h"
```

Here, we create a TransformListener object. Once the listener is created, it starts receiving tf2 transformations over the wire, and buffers them for up to 10 seconds.

```
tf_listener_ =
  std::make_shared<tf2_ros::TransformListener>(*tf_buffer_);
```

Finally, we query the listener for a specific transformation. We call lookup_transform method with following arguments:

1. Target frame
2. Source frame
3. The time at which we want to transform

Providing tf2::TimePointZero() will just get us the latest available transform. All this is wrapped in a try-catch block to handle possible exceptions.

```
t = tf_buffer_""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
```
1.2 CMakeLists.txt

Navigate one level back to the learning_tf2_cpp directory, where the CMakeLists.txt and package.xml files are located.

Now open the CMakeLists.txt add the executable and name it turtle_tf2_listener, which you’ll use later with ros2 run.

```cmake
add_executable(turtle_tf2_listener src/turtle_tf2_listener.cpp)
ament_target_dependencies(
  turtle_tf2_listener
  geometry_msgs
  rclcpp
  tf2
  tf2_ros
  turtlesim
)
```

Finally, add the install(TARGETS...) section so ros2 run can find your executable:

```cmake
install(TARGETS
turtle_tf2_listener
  DESTINATION lib/${PROJECT_NAME})
```

2 Update the launch file

Open the launch file called turtle_tf2_demo_launch.py with your text editor, add two new nodes to the launch description, add a launch argument, and add the imports. The resulting file should look like:

```python
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch_ros.actions import Node

from launch_ros.substitutions import FindPackageShare

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='turtlesim',
            executable='turtlesim_node',
            name='sim',
        ),
        Node(
            package='learning_tf2_cpp',
            executable='turtle_tf2_broadcaster',
            name='broadcaster1',
            parameters=[
                {'turtlename': 'turtle1'}
            ],
        ),
        DeclareLaunchArgument(
```
This will declare a `target_frame` launch argument, start a broadcaster for second turtle that we will spawn and listener that will subscribe to those transformations.

### 3 Build

Run `rosdep` in the root of your workspace to check for missing dependencies.

#### Linux

```
rosdep install -i --from-path src --rosdistro iron -y
```

#### macOS

Rosdep only runs on Linux, so you will need to install `geometry_msgs` and `turtlesim` dependencies yourself.

#### Windows

Rosdep only runs on Linux, so you will need to install `geometry_msgs` and `turtlesim` dependencies yourself.

From the root of your workspace, build your updated package:

#### Linux

```
colcon build --packages-select learning_tf2_cpp
```

#### macOS

```
colcon build --packages-select learning_tf2_cpp
```

#### Windows

```
colcon build --merge-install --packages-select learning_tf2_cpp
```

Open a new terminal, navigate to the root of your workspace, and source the setup files:
Linux

```bash
. install/setup.bash
```

macOS

```bash
. install/setup.bash
```

Windows

```bash
# CMD
call install\setup.bat

# Powershell
.\install\setup.ps1
```

### 4 Run

Now you’re ready to start your full turtle demo:

```bash
ros2 launch learning_tf2_cpp turtle_tf2_demo_launch.py
```

You should see the turtle sim with two turtles. In the second terminal window type the following command:

```bash
ros2 run turtlesim turtle_teleop_key
```

To see if things work, simply drive around the first turtle using the arrow keys (make sure your terminal window is active, not your simulator window), and you’ll see the second turtle following the first one!

### Summary

In this tutorial you learned how to use tf2 to get access to frame transformations. You also have finished writing your own turtlesim demo that you first tried in *Introduction to tf2* tutorial.

### Adding a frame (Python)

**Goal:** Learn how to add an extra frame to tf2.

**Tutorial level:** Intermediate

**Time:** 15 minutes

**Contents**

- **Background**
- **tf2 tree**
- **Tasks**
  - 1 Write the fixed frame broadcaster
    - 1.1 Examine the code
Background

In previous tutorials, we recreated the turtle demo by writing a tf2 broadcaster and a tf2 listener. This tutorial will teach you how to add extra fixed and dynamic frames to the transformation tree. In fact, adding a frame in tf2 is very similar to creating the tf2 broadcaster, but this example will show you some additional features of tf2.

For many tasks related to transformations, it is easier to think inside a local frame. For example, it is easiest to reason about laser scan measurements in a frame at the center of the laser scanner. tf2 allows you to define a local frame for each sensor, link, or joint in your system. When transforming from one frame to another, tf2 will take care of all the hidden intermediate frame transformations that are introduced.

tf2 tree

tf2 builds up a tree structure of frames and, thus, does not allow a closed loop in the frame structure. This means that a frame only has one single parent, but it can have multiple children. Currently, our tf2 tree contains three frames: world, turtle1 and turtle2. The two turtle frames are children of the world frame. If we want to add a new frame to tf2, one of the three existing frames needs to be the parent frame, and the new one will become its child frame.
Tasks

1 Write the fixed frame broadcaster

In our turtle example, we’ll add a new frame carrot1, which will be the child of the turtle1. This frame will serve as the goal for the second turtle.

Let’s first create the source files. Go to the learning_tf2_py package we created in the previous tutorials. Download the fixed frame broadcaster code by entering the following command:

Linux

```
wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_py/turtle_tf2_py/fixed_frame_tf2_broadcaster.py
```

macOS

```
wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_py/turtle_tf2_py/fixed_frame_tf2_broadcaster.py
```

Windows

In a Windows command line prompt:

```
curl -sk https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_py/turtle_tf2_py/fixed_frame_tf2_broadcaster.py -o fixed_frame_tf2_broadcaster.py
```

Or in powershell:

```
curl https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_py/turtle_tf2_py/fixed_frame_tf2_broadcaster.py -o fixed_frame_tf2_broadcaster.py
```

Now open the file called fixed_frame_tf2_broadcaster.py.
from geometry_msgs.msg import TransformStamped

import rclpy
from rclpy.node import Node

from tf2_ros import TransformBroadcaster

class FixedFrameBroadcaster(Node):
    def __init__(self):
        super().__init__('fixed_frame_tf2_broadcaster')
        self.tf_broadcaster = TransformBroadcaster(self)
        self.timer = self.create_timer(0.1, self.broadcast_timer_callback)

    def broadcast_timer_callback(self):
        t = TransformStamped()

        t.header.stamp = self.get_clock().now().to_msg()
        t.header.frame_id = 'turtle1'
        t.child_frame_id = 'carrot1'
        t.transform.translation.x = 0.0
        t.transform.translation.y = 2.0
        t.transform.translation.z = 0.0
        t.transform.rotation.x = 0.0
        t.transform.rotation.y = 0.0
        t.transform.rotation.z = 0.0
        t.transform.rotation.w = 1.0

        self.tf_broadcaster.sendTransform(t)

def main():
    rclpy.init()
    node = FixedFrameBroadcaster()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass

    rclpy.shutdown()

The code is very similar to the tf2 broadcaster tutorial example and the only difference is that the transform here does not change over time.
1.1 Examine the code

Let’s take a look at the key lines in this piece of code. Here we create a new transform, from the parent turtle1 to the new child carrot1. The carrot1 frame is 2 meters offset in y axis in terms of the turtle1 frame.

```python
t = TransformStamped()
t.header.stamp = self.get_clock().now().to_msg()
t.header.frame_id = 'turtle1'
t.child_frame_id = 'carrot1'
t.transform.translation.x = 0.0
t.transform.translation.y = 2.0
t.transform.translation.z = 0.0
```

1.2 Add an entry point

To allow the ros2 run command to run your node, you must add the entry point to setup.py (located in the src/learning_tf2_py directory).

Finally, add the following line between the 'console_scripts': brackets:

```python
'fixed_frame_tf2_broadcaster = learning_tf2_py.fixed_frame_tf2_broadcaster:main',
```

1.3 Write the launch file

Now let’s create a launch file for this example. With your text editor, create a new file called launch/turtle_tf2_fixed_frame_demo_launch.py, and add the following lines:

```python
import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch_ros.actions import Node

def generate_launch_description():
    demo_nodes = IncludeLaunchDescription(
        PythonLaunchDescriptionSource([os.path.join(
            get_package_share_directory('learning_tf2_py'), 'launch',
            '/turtle_tf2_demo_launch.py')]),
    )

    return LaunchDescription([
        demo_nodes,
        Node(
            package='learning_tf2_py',
            executable='fixed_frame_tf2_broadcaster',
        )
    ])
```
This launch file imports the required packages and then creates a `demo_nodes` variable that will store nodes that we created in the previous tutorial’s launch file.

The last part of the code will add our fixed carrot1 frame to the turtlesim world using our `fixed_frame_tf2_broadcaster` node.

```
Node(
    package='learning_tf2_py',
    executable='fixed_frame_tf2_broadcaster',
    name='fixed_broadcaster',
),
```

### 1.4 Build

Run `rosdep` in the root of your workspace to check for missing dependencies.

Linux

```
rosdep install -i --from-path src --rosdistro iron -y
```

macOS

`rosdep` only runs on Linux, so you will need to install `geometry_msgs` and `turtlesim` dependencies yourself

Windows

`rosdep` only runs on Linux, so you will need to install `geometry_msgs` and `turtlesim` dependencies yourself

Still in the root of your workspace, build your package:

Linux

```
colcon build --packages-select learning_tf2_py
```

macOS

```
colcon build --packages-select learning_tf2_py
```

Windows

```
colcon build --merge-install --packages-select learning_tf2_py
```

Open a new terminal, navigate to the root of your workspace, and source the setup files:

Linux

```
. install/setup.bash
```

macOS

```
. install/setup.bash
```

Windows
1.5 Run

Now you are ready to run the launch file:

```bash
ros2 launch learning_tf2_py turtle_tf2_fixed_frame_demo_launch.py
```

You should notice that the new carrot1 frame appeared in the transformation tree.

If you drive the first turtle around, you should notice that the behavior didn’t change from the previous tutorial, even though we added a new frame. That’s because adding an extra frame does not affect the other frames and our listener is still using the previously defined frames.

Therefore if we want our second turtle to follow the carrot instead of the first turtle, we need to change value of the `target_frame`. This can be done two ways. One way is to pass the `target_frame` argument to the launch file directly from the console:

```bash
ros2 launch learning_tf2_py turtle_tf2_fixed_frame_demo_launch.py target_frame:=carrot1
```

The second way is to update the launch file. To do so, open the `turtle_tf2_fixed_frame_demo_launch.py` file, and add the `target_frame`: `carrot1` parameter via `launch_arguments` argument.
```python
def generate_launch_description():
    demo_nodes = IncludeLaunchDescription(
        ...
        launch_arguments={'target_frame': 'carrot1'}.items(),
    )
```

Now just rebuild the package, restart the turtle_tf2_fixed_frame_demo_launch.py, and you’ll see the second turtle following the carrot instead of the first turtle!

2 Write the dynamic frame broadcaster

The extra frame we published in this tutorial is a fixed frame that doesn’t change over time in relation to the parent frame. However, if you want to publish a moving frame you can code the broadcaster to change the frame over time. Let’s change our carrot1 frame so that it changes relative to turtle1 frame over time. Now download the dynamic frame broadcaster code by entering the following command:

Linux

```
wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_py/turtle_tf2_py/dynamic_frame_tf2_broadcaster.py
```
macOS

```bash
wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_py/turtle_tf2_py/dynamic_frame_tf2_broadcaster.py
```

Windows

In a Windows command line prompt:

```bash
curl -sk https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_py/turtle_tf2_py/dynamic_frame_tf2_broadcaster.py -o dynamic_frame_tf2_broadcaster.py
```

Or in powershell:

```bash
curl https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_py/turtle_tf2_py/dynamic_frame_tf2_broadcaster.py -o dynamic_frame_tf2_broadcaster.py
```

Now open the file called `dynamic_frame_tf2_broadcaster.py`:

```python
import math
from geometry_msgs.msg import TransformStamped
import rclpy
from rclpy.node import Node
from tf2_ros import TransformBroadcaster

class DynamicFrameBroadcaster(Node):
    def __init__(self):
        super().__init__('dynamic_frame_tf2_broadcaster')
        self.tf_broadcaster = TransformBroadcaster(self)
        self.timer = self.create_timer(0.1, self.broadcast_timer_callback)

    def broadcast_timer_callback(self):
        seconds, _ = self.get_clock().now().seconds_nanoseconds()
        x = seconds * math.pi

        t = TransformStamped()
        t.header.stamp = self.get_clock().now().to_msg()
        t.header.frame_id = 'turtle1'
        t.child_frame_id = 'carrot1'
        t.transform.translation.x = 10 * math.sin(x)
        t.transform.translation.y = 10 * math.cos(x)
        t.transform.translation.z = 0.0
        t.transform.rotation.x = 0.0
        t.transform.rotation.y = 0.0
        t.transform.rotation.z = 0.0
        t.transform.rotation.w = 1.0

        self.tf_broadcaster.sendTransform(t)
```

(continues on next page)
def main():
    rclpy.init()
    node = DynamicFrameBroadcaster()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    rclpy.shutdown()

2.1 Examine the code

Instead of a fixed definition of our x and y offsets, we are using the \( \sin() \) and \( \cos() \) functions on the current time so that the offset of \texttt{carrot1} is constantly changing.

\[
\text{seconds}, _ = \text{self.get_clock().now().seconds.nanoseconds()}
\]
\[
x = \text{seconds} \times \text{math.pi}
\]
\[
\ldots
\]
\[
t.\text{transform.translation.x} = 10 \times \text{math.sin}(x)
\]
\[
t.\text{transform.translation.y} = 10 \times \text{math.cos}(x)
\]

2.2 Add an entry point

To allow the \texttt{ros2 run} command to run your node, you must add the entry point to \texttt{setup.py} (located in the \texttt{src/learning_tf2_py} directory).

Finally, add the following line between the \texttt{'console_scripts':} brackets:

```
'dynamic_frame_tf2_broadcaster = learning_tf2_py.dynamic_frame_tf2_broadcaster:main',
```

2.3 Write the launch file

To test this code, create a new launch file \texttt{launch/turtle_tf2_dynamic_frame_demo_launch.py} and paste the following code:

```python
import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch_ros.actions import Node

def generate_launch_description():
    demo_nodes = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(os.path.join(
```

(continues on next page)
get_package_share_directory('learning_tf2_py', 'launch'),
        '/turtle_tf2_demo_launch.py'),
    launch_arguments={'target_frame': 'carrot1'}.items(),
)

return LaunchDescription([
    demo_nodes,
    Node(
        package='learning_tf2_py',
        executable='dynamic_frame_tf2_broadcaster',
        name='dynamic_broadcaster',
    ),
])

2.4 Build

Run rosdep in the root of your workspace to check for missing dependencies.

Linux

rosdep install -i --from-path src --rosdistro iron -y

macOS

rosdep only runs on Linux, so you will need to install geometry_msgs and turtlesim dependencies yourself

Windows

rosdep only runs on Linux, so you will need to install geometry_msgs and turtlesim dependencies yourself

Still in the root of your workspace, build your package:

Linux

colcon build --packages-select learning_tf2_py

macOS

colcon build --packages-select learning_tf2_py

Windows

colcon build --merge-install --packages-select learning_tf2_py

Open a new terminal, navigate to the root of your workspace, and source the setup files:

Linux

. install/setup.bash

macOS

. install/setup.bash

Windows
1.5 Run

Now you are ready to run the launch file:

```bash
ros2 launch learning_tf2_py turtle_tf2_dynamic_frame_demo_launch.py
```

You should see that the second turtle is following the carrot's position that is constantly changing.
Summary

In this tutorial, you learned about the tf2 transformation tree, its structure, and its features. You also learned that it is easiest to think inside a local frame, and learned to add extra fixed and dynamic frames for that local frame.

Adding a frame (C++)

Goal: Learn how to add an extra frame to tf2.

Tutorial level: Intermediate

Time: 15 minutes

Contents

- Background
- tf2 tree
- Tasks
  - 1 Write the fixed frame broadcaster
    - 1.1 Examine the code
    - 1.2 CMakeLists.txt
    - 1.3 Write the launch file
    - 1.4 Build
    - 1.5 Run
  - 2 Write the dynamic frame broadcaster
    - 2.1 Examine the code
    - 2.2 CMakeLists.txt
    - 2.3 Write the launch file
    - 2.4 Build
    - 2.5 Run

- Summary

Background

In previous tutorials, we recreated the turtle demo by writing a tf2 broadcaster and a tf2 listener. This tutorial will teach you how to add extra fixed and dynamic frames to the transformation tree. In fact, adding a frame in tf2 is very similar to creating the tf2 broadcaster, but this example will show you some additional features of tf2.

For many tasks related to transformations, it is easier to think inside a local frame. For example, it is easiest to reason about laser scan measurements in a frame at the center of the laser scanner. tf2 allows you to define a local frame for each sensor, link, or joint in your system. When transforming from one frame to another, tf2 will take care of all the hidden intermediate frame transformations that are introduced.
tf2 tree

tf2 builds up a tree structure of frames and, thus, does not allow a closed loop in the frame structure. This means that a frame only has one single parent, but it can have multiple children. Currently, our tf2 tree contains three frames: world, turtle1 and turtle2. The two turtle frames are children of the world frame. If we want to add a new frame to tf2, one of the three existing frames needs to be the parent frame, and the new one will become its child frame.

![View Frames Result]

<table>
<thead>
<tr>
<th>World</th>
<th>Turtle2</th>
<th>Turtle1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broadcaster: default authority</td>
<td>Broadcaster: default authority</td>
<td>Broadcaster: default authority</td>
</tr>
<tr>
<td>Average rate: 62.682</td>
<td>Average rate: 62.881</td>
<td>Average rate: 62.881</td>
</tr>
<tr>
<td>Buffer length: 5.073</td>
<td>Buffer length: 5.073</td>
<td>Buffer length: 5.073</td>
</tr>
<tr>
<td>Most recent transform: 1622031689.094395</td>
<td>Most recent transform: 1622031689.094407</td>
<td>Most recent transform: 1622031689.094407</td>
</tr>
<tr>
<td>Oldest transform: 1622031684.02114</td>
<td>Oldest transform: 1622031684.02114</td>
<td>Oldest transform: 1622031684.02114</td>
</tr>
</tbody>
</table>

Tasks

1 Write the fixed frame broadcaster

In our turtle example, we’ll add a new frame carrot1, which will be the child of the turtle1. This frame will serve as the goal for the second turtle.

Let’s first create the source files. Go to the learning_tf2_cpp package we created in the previous tutorials. Download the fixed frame broadcaster code by entering the following command:

Linux

```
wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/fixed_frame_tf2_broadcaster.cpp
```

macOS

```
wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/fixed_frame_tf2_broadcaster.cpp
```

Windows

```
curl -sk https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/fixed_frame_tf2_broadcaster.cpp -o fixed_frame_tf2_broadcaster.cpp
```
Or in powershell:

```
curl https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/ fixed_frame_tf2_broadcaster.cpp -o fixed_frame_tf2_broadcaster.cpp
```

Now open the file called `fixed_frame_tf2_broadcaster.cpp`.

```cpp
#include <chrono>
#include <functional>
#include <memory>
#include "geometry_msgs/msg/transform_stamped.hpp"
#include "rclcpp/rclcpp.hpp"
#include "tf2_ros/transform_broadcaster.h"

using namespace std::chrono_literals;

class FixedFrameBroadcaster : public rclcpp::Node
{
public:
    FixedFrameBroadcaster()
    : Node("fixed_frame_tf2_broadcaster")
    {
        tf_broadcaster_ = std::make_shared<tf2_ros::TransformBroadcaster>(this);
        timer_ = this->create_wall_timer(100ms, std::bind(&FixedFrameBroadcaster::broadcast_timer_callback, this));
    }

private:
    void broadcast_timer_callback()
    {
        geometry_msgs::msg::TransformStamped t;
        t.header.stamp = this->get_clock()->now();
        t.header.frame_id = "turtle1";
        t.child_frame_id = "carrot1";
        t.transform.translation.x = 0.0;
        t.transform.translation.y = 2.0;
        t.transform.translation.z = 0.0;
        t.transform.rotation.x = 0.0;
        t.transform.rotation.y = 0.0;
        t.transform.rotation.z = 0.0;
        t.transform.rotation.w = 1.0;
        tf_broadcaster_->sendTransform(t);
    }

    rclcpp::TimerBase::SharedPtr timer_
    std::shared_ptr<tf2_ros::TransformBroadcaster> tf_broadcaster_
};

int main(int argc, char * argv[])
{
    rclcpp::init(argc, argv);
}```
```
rclcpp::spin(std::make_shared<FixedFrameBroadcaster>())
rclcpp::shutdown();
return 0;
```

The code is very similar to the tf2 broadcaster tutorial example and the only difference is that the transform here does not change over time.

### 1.1 Examine the code

Let's take a look at the key lines in this piece of code. Here we create a new transform, from the parent turtle1 to the new child carrot1. The carrot1 frame is 2 meters offset in y axis in terms of the turtle1 frame.

```
geometry_msgs::msg::TransformStamped t;
t.header.stamp = this->get_clock()->now();
t.header.frame_id = "turtle1";
t.child_frame_id = "carrot1";
t.transform.translation.x = 0.0;
t.transform.translation.y = 2.0;
t.transform.translation.z = 0.0;
```

### 1.2 CMakeLists.txt

Navigate one level back to the learning_tf2_cpp directory, where the CMakeLists.txt and package.xml files are located.

Now open the CMakeLists.txt add the executable and name it fixed_frame_tf2_broadcaster.

```
add_executable(fixed_frame_tf2_broadcaster src/fixed_frame_tf2_broadcaster.cpp)
ament_target_dependencies(
   fixed_frame_tf2_broadcaster
   geometry_msgs
   rclcpp
   tf2_ros
)
```

Finally, add the install(TARGETS...) section so ros2 run can find your executable:

```
install(TARGETS
   fixed_frame_tf2_broadcaster
   DESTINATION lib/${PROJECT_NAME})
```
1.3 Write the launch file

Now let's create a launch file for this example. With your text editor, create a new file called `turtle_tf2_fixed_frame_demo_launch.py`, and add the following lines:

```python
import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch_ros.actions import Node

def generate_launch_description():
    demo_nodes = IncludeLaunchDescription(
        PythonLaunchDescriptionSource([os.path.join(
            get_package_share_directory('learning_tf2_cpp'), 'launch',
            '/turtle_tf2_demo_launch.py')]),
    )

    return LaunchDescription([
        demo_nodes,
        Node(
            package='learning_tf2_cpp',
            executable='fixed_frame_tf2_broadcaster',
            name='fixed_broadcaster',
        ),
    ])
```

This launch file imports the required packages and then creates a `demo_nodes` variable that will store nodes that we created in the previous tutorial's launch file.

The last part of the code will add our fixed `carrot1` frame to the turtlesim world using our `fixed_frame_tf2_broadcaster` node.

```python
Node(
    package='learning_tf2_cpp',
    executable='fixed_frame_tf2_broadcaster',
    name='fixed_broadcaster',
),
```
1.4 Build

Run `rosdep` in the root of your workspace to check for missing dependencies.

Linux

```
rosdep install -i --from-path src --rosdistro iron -y
```

macOS
rosdep only runs on Linux, so you will need to install `geometry_msgs` and `turtlesim` dependencies yourself

Windows
rosdep only runs on Linux, so you will need to install `geometry_msgs` and `turtlesim` dependencies yourself

From the root of your workspace, build your updated package:

Linux

```
colcon build --packages-select learning_tf2_cpp
```

macOS

```
colcon build --packages-select learning_tf2_cpp
```

Windows

```
colcon build --merge-install --packages-select learning_tf2_cpp
```

Open a new terminal, navigate to the root of your workspace, and source the setup files:

Linux

```
. install/setup.bash
```

macOS

```
. install/setup.bash
```

Windows

```
# CMD
call install\setup.bat

# Powershell
.\install\setup.ps1
```
1.5 Run

Now you can start the turtle broadcaster demo:

```
ros2 launch learning_tf2_cpp turtle_tf2_fixed_frame_demo_launch.py
```

You should notice that the new `carrot1` frame appeared in the transformation tree.

If you drive the first turtle around, you should notice that the behavior didn’t change from the previous tutorial, even though we added a new frame. That’s because adding an extra frame does not affect the other frames and our listener is still using the previously defined frames.

Therefore if we want our second turtle to follow the carrot instead of the first turtle, we need to change value of the `target_frame`. This can be done two ways. One way is to pass the `target_frame` argument to the launch file directly from the console:

```
ros2 launch learning_tf2_cpp turtle_tf2_fixed_frame_demo_launch.py target_frame:=carrot1
```

The second way is to update the launch file. To do so, open the `turtle_tf2_fixed_frame_demo_launch.py` file, and add the `target_frame`: 'carrot1' parameter via `launch_arguments` argument.

```python
def generate_launch_description():
    demo_nodes = IncludeLaunchDescription(
        ...
        launch_arguments={'target_frame': 'carrot1'}.items(),
    )
```

Now rebuild the package, restart the `turtle_tf2_fixed_frame_demo_launch.py`, and you’ll see the second turtle following the carrot instead of the first turtle!
2 Write the dynamic frame broadcaster

The extra frame we published in this tutorial is a fixed frame that doesn’t change over time in relation to the parent frame. However, if you want to publish a moving frame you can code the broadcaster to change the frame over time. Let’s change our carrot1 frame so that it changes relative to turtle1 frame over time. Now download the dynamic frame broadcaster code by entering the following command:

**Linux**

```
wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/...dynamic_frame_tf2_broadcaster.cpp
```

**macOS**

```
wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/...dynamic_frame_tf2_broadcaster.cpp
```

**Windows**

In a Windows command line prompt:
curl -sk https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/_src/dynamic_frame_tf2_broadcaster.cpp -o dynamic_frame_tf2_broadcaster.cpp

Or in powershell:

curl https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/_dynamic_frame_tf2_broadcaster.cpp -o dynamic_frame_tf2_broadcaster.cpp

Now open the file called dynamic_frame_tf2_broadcaster.cpp:

```cpp
#include <chrono>
#include <functional>
#include <memory>
#include "geometry_msgs/msg/transform_stamped.hpp"
#include "rclcpp/rclcpp.hpp"
#include "tf2_ros/transform_broadcaster.h"

using namespace std::chrono_literals;

const double PI = 3.141592653589793238463;

class DynamicFrameBroadcaster : public rclcpp::Node {
public:
  DynamicFrameBroadcaster() : Node("dynamic_frame_tf2_broadcaster")
  {
    tf_broadcaster_ = std::make_shared<tf2_ros::TransformBroadcaster>(this);
    timer_ = this->create_wall_timer(100ms, std::bind(&DynamicFrameBroadcaster::broadcast_timer_callback, this));
  }

private:
  void broadcast_timer_callback()
  {
    rclcpp::Time now = this->get_clock()->now();
    double x = now.seconds() * PI;
    geometry_msgs::msg::TransformStamped t;
    t.header.stamp = now;
    t.header.frame_id = "turtle1";
    t.child_frame_id = "carrot1";
    t.transform.translation.x = 10 * sin(x);
    t.transform.translation.y = 10 * cos(x);
    t.transform.translation.z = 0.0;
    t.transform.rotation.x = 0.0;
    t.transform.rotation.y = 0.0;
    t.transform.rotation.z = 0.0;
    t.transform.rotation.w = 1.0;
    tf_broadcaster_->sendTransform(t);
  }
};
```

(continues on next page)
```cpp
rclcpp::TimerBase::SharedPtr timer_; 
std::shared_ptr<tf2_ros::TransformBroadcaster> tf_broadcaster_; 
};

int main(int argc, char * argv[])
{
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<DynamicFrameBroadcaster>();
    rclcpp::shutdown();
    return 0;
}
```

### 2.1 Examine the code

Instead of a fixed definition of our x and y offsets, we are using the \(\sin()\) and \(\cos()\) functions on the current time so that the offset of `carrot1` is constantly changing.

```cpp
double x = now.seconds() * PI;
...
t.transform.translation.x = 10 * \sin(x);
t.transform.translation.y = 10 * \cos(x);
```

### 2.2 CMakeLists.txt

Navigate one level back to the `learning_tf2_cpp` directory, where the `CMakeLists.txt` and `package.xml` files are located.

Now open the `CMakeLists.txt` add the executable and name it `dynamic_frame_tf2_broadcaster`.

```cpp
add_executable(dynamic_frame_tf2_broadcaster src/dynamic_frame_tf2_broadcaster.cpp)
ament_target_dependencies(
    dynamic_frame_tf2_broadcaster
    geometry_msgs
    rclcpp
    tf2_ros
)
```

Finally, add the `install(TARGETS...)` section so `ros2 run` can find your executable:

```cpp
install(TARGETS
    dynamic_frame_tf2_broadcaster
    DESTINATION lib/${PROJECT_NAME})
```

---

4.8. ROS 2 Documentation
2.3 Write the launch file

To test this code, create a new launch file `turtle_tf2_dynamic_frame_demo_launch.py` and paste the following code:

```python
import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch_ros.actions import Node

def generate_launch_description():
    demo_nodes = IncludeLaunchDescription(
        PythonLaunchDescriptionSource([os.path.join(
            get_package_share_directory('learning_tf2_cpp'), 'launch'),
            '/turtle_tf2_demo_launch.py']),
        launch_arguments={'target_frame': 'carrot1'}.items(),
    )
    return LaunchDescription([
        demo_nodes,
        Node(
            package='learning_tf2_cpp',
            executable='dynamic_frame_tf2_broadcaster',
            name='dynamic_broadcaster',
        ),
    ])

2.4 Build

Run `rosdep` in the root of your workspace to check for missing dependencies.

Linux

```
rosdep install -i --from-path src --rosdistro iron -y
```

macOS

`rosdep` only runs on Linux, so you will need to install `geometry_msgs` and `turtlesim` dependencies yourself

Windows

`rosdep` only runs on Linux, so you will need to install `geometry_msgs` and `turtlesim` dependencies yourself

From the root of your workspace, build your updated package:

Linux

```
colcon build --packages-select learning_tf2_cpp
```

macOS
```sh
colcon build --packages-select learning_tf2_cpp
```

Windows

```sh
colcon build --merge-install --packages-select learning_tf2_cpp
```

Open a new terminal, navigate to the root of your workspace, and source the setup files:

Linux

```sh
. install/setup.bash
```

macOS

```sh
. install/setup.bash
```

Windows

```sh
# CMD
call install\setup.bat

# Powershell
disable_name_resolution
.\install\setup.ps1
```

### 2.5 Run

Now you can start the dynamic frame demo:

```sh
ros2 launch learning_tf2_cpp turtle_tf2_dynamic_frame_demo_launch.py
```

You should see that the second turtle is following the carrot's position that is constantly changing.
Summary

In this tutorial, you learned about the tf2 transformation tree, its structure, and its features. You also learned that it is easiest to think inside a local frame, and learned to add extra fixed and dynamic frames for that local frame.

Using time (Python)

Goal: Learn to use the timeout in `lookup_transform` function to wait for a transform to be available on the tf2 tree.

Tutorial level: Intermediate

Time: 10 minutes

Contents

- Background
- Tasks
  - Update the listener node
---

## 2 Fix the listener node

### Summary

**Background**

In previous tutorials, we recreated the turtle demo by writing a tf2 broadcaster and a tf2 listener. We also learned how to add a new frame to the transformation tree. Now we will learn more about the timeout argument which makes the `lookup_transform` wait for the specified transform for up to the specified duration before throwing an exception. This tool can be useful to listen for transforms that are published at varying rates or those incoming source with unreliable networking and non negligible latency. This tutorial will teach you how use the timeout in `lookup_transform` function to wait for a transform to be available on the tf2 tree.

### Tasks

1. Update the listener node

   Edit `turtle_tf2_listener.py` and remove the `timeout=Duration(seconds=1.0)` parameter that is passed to the `lookup_transform()` call on line 76. It should look like shown below:

   ```python
   trans = self._tf_buffer.lookup_transform(
       to_frame_rel,
       from_frame_rel,
       now)
   ```

   Moreover, import additional exceptions that we will handle in the beginning of the file:

   ```python
   from tf2_ros import LookupException, ConnectivityException, ExtrapolationException
   ```

   Edit the exception handling on line 81 by adding newly imported exceptions and raise statement to see the exception:

   ```python
   except (LookupException, ConnectivityException, ExtrapolationException):
       self.get_logger().info('transform not ready')
       raise
   return
   ```

   If you now try to run the launch file, you will notice that it is failing:

   ```bash
   ros2 launch learning_tf2_py turtle_tf2_demo_launch.py
   ```

2. Fix the listener node

   You now should notice that `lookup_transform()` is failing. It tells you that the frame does not exist or that the data is in the future. To fix this, edit your code on line 76 as shown below (return the timeout parameter):

   ```python
   trans = self._tf_buffer.lookup_transform(
       to_frame_rel,
       from_frame_rel,
       now,
       timeout=rclpy.duration.Duration(seconds=1.0))
   ```

---
The `lookup_transform` can take four arguments, where the last one is an optional timeout. It will block for up to that duration waiting for it to timeout.

**Note:** Once this change is made, remove the `raise` line from the `except()` block that we added above or the code will continue to fail.

You can now run the launch file.

```bash
ros2 launch learning_tf2_py turtle_tf2_demo_launch.py
```

You should notice that `lookup_transform()` will actually block until the transform between the two turtles becomes available (this will usually take a few milli-seconds). Once the timeout has been reached (one second in this case), an exception will be raised only if the transform is still not available.

**Summary**

In this tutorial you learned more about the `lookup_transform` function and its timeout features. You also learned how to catch and handle additional exceptions that can be thrown by tf2.

**Using time (C++)**

**Goal:** Learn how to get a transform at a specific time and wait for a transform to be available on the tf2 tree using `lookupTransform()` function.

**Tutorial level:** Intermediate

**Time:** 10 minutes

**Background**

In previous tutorials, we recreated the turtle demo by writing a `tf2 broadcaster` and a `tf2 listener`. We also learned how to `add a new frame to the transformation tree` and learned how tf2 keeps track of a tree of coordinate frames. This tree changes over time, and tf2 stores a time snapshot for every transform (for up to 10 seconds by default). Until now we used the `lookupTransform()` function to get access to the latest available transforms in that tf2 tree, without knowing at what time that transform was recorded. This tutorial will teach you how to get a transform at a specific time.
Tasks

1 tf2 and time

So let’s go back to where we ended in the adding a frame tutorial. Go to learning_tf2_cpp package. Open turtle_tf2_listener.cpp and take a look at the lookupTransform() call:

```cpp
try {
    t = tf_buffer_->lookupTransform(
        toFrameRel,
        fromFrameRel,
        tf2::TimePointZero);
} catch (const tf2::TransformException & ex) {
```

You can see that we specified a time equal to 0 by calling tf2::TimePointZero.

**Note:** The tf2 package has its own time type tf2::TimePoint, which is different from rclcpp::Time. Many APIs in the package tf2_ros automatically convert between rclcpp::Time and tf2::TimePoint.

```cpp
rclcpp::Time(0, 0, this->get_clock()->get_clock_type())
```

could have been used here, but it would have been converted to tf2::TimePointZero anyways.

For tf2, time 0 means “the latest available” transform in the buffer. Now, change this line to get the transform at the current time:

```cpp
rclcpp::Time now = this->get_clock()->now();
try {
    t = tf_buffer_->lookupTransform(
        toFrameRel, fromFrameRel,
        now);
} catch (const tf2::TransformException & ex) {
```

Now build the package and try to run the launch file.

```
ros2 launch learning_tf2_cpp turtle_tf2_demo_launch.py
```

You will notice that it fails and outputs something similar to this:

```
[INFO] [1629873136.345688064] [listener]: Could not transform turtle1 to turtle2: Lookup would require extrapolation into the future. Requested time 1629873136.345539 but the latest data is at time 1629873136.338804, when looking up transform from frame [turtle1] to frame [turtle2]
```

It tells you that the frame does not exist or that the data is in the future.

To understand why is this happening we need to understand how buffers work. Firstly, each listener has a buffer where it stores all the coordinate transforms coming from the different tf2 broadcasters. Secondly, when a broadcaster sends out a transform, it takes some time before that transform gets into the buffer (usually a couple of milliseconds). As a result, when you request a frame transform at time “now”, you should wait a few milliseconds for that information to arrive.
2 Wait for transforms

tf2 provides a nice tool that will wait until a transform becomes available. You use this by adding a timeout parameter to `lookupTransform()`. To fix this, edit your code as shown below (add the last timeout parameter):

```cpp
rclcpp::Time now = this->get_clock()->now();
try {
    t = tf_buffer_->lookupTransform(
        toFrameRel,  // Output frame
        fromFrameRel, // Source frame
        now,          // Timestamp
        50ms);        // Optional timeout
} catch (const tf2::TransformException & ex) {
```

The `lookupTransform()` can take four arguments, where the last one is an optional timeout. It will block for up to that duration waiting for it to timeout.

3 Checking the results

You can now build the package and run the launch file.

```
ros2 launch learning_tf2_CPP turtle_tf2_demo_launch.py
```

You should notice that `lookupTransform()` will actually block until the transform between the two turtles becomes available (this will usually take a few milliseconds). Once the timeout has been reached (fifty milliseconds in this case), an exception will be raised only if the transform is still not available.

Summary

In this tutorial, you learned how to acquire a transform at a specific timestamp and how to wait for a transform to be available on the tf2 tree when using the `lookupTransform()` function.

Traveling in time (Python)

**Goal:** Learn about advanced time travel features of tf2.

**Tutorial level:** Intermediate

**Time:** 10 minutes
Background

In the previous tutorial, we discussed the basics of tf2 and time. This tutorial will take us one step further and expose a powerful tf2 trick: the time travel. In short, one of the key features of tf2 library is that it is able to transform data in time as well as in space.

This tf2 time travel feature can be useful for various tasks, like monitoring the pose of the robot for a long period of time or building a follower robot that will follow the “steps” of the leader. We will use that time travel feature to look up transforms back in time and program turtle2 to follow 5 seconds behind carrot1.

Time travel

First, let’s go back to where we ended in the previous tutorial Using time. Go to your learning_tf2_py package.

Now, instead of making the second turtle go to where the carrot is now, we will make the second turtle go to where the first carrot was 5 seconds ago. Edit the lookup_transform() call in turtle_tf2_listener.py file to

```python
when = self.get_clock().now() - rclpy.time.Duration(seconds=5.0)
try:
    t = self.tf_buffer.lookup_transform(
        to_frame_rel,
        from_frame_rel,
        when,
        timeout=rclpy.duration.Duration(seconds=0.05))
except TransformException as ex:
```

Now if you run this, during the first 5 seconds, the second turtle would not know where to go because we do not yet have a 5-second history of poses of the carrot. But what happens after these 5 seconds? Build the package as usual then let’s just give it a try:

```
ros2 launch learning_tf2_py turtle_tf2_fixed_frame_demo_launch.py
```
You should now notice that your turtle is driving around uncontrollably like in this screenshot. Let’s try to understand reason behind that behavior.

1. In our code we asked tf2 the following question: “What was the pose of carrot1 5 seconds ago, relative to turtle2 5 seconds ago?”. This means we are controlling the second turtle based on where it was 5 seconds ago as well as where the first carrot was 5 seconds ago.

2. However, what we really want to ask is: “What was the pose of carrot1 5 seconds ago, relative to the current position of the turtle2?”.

**Advanced API for lookup_transform()**

To ask the tf2 that particular question, we will use an advanced API that gives us the power to say explicitly when to acquire the specified transformations. This is done by calling the `lookup_transform_full()` method with additional parameters. Your code now would look like this:

```python
when = self.get_clock().now() - rclpy.time.Duration(seconds=5.0)
try:
    t = self.tf_buffer.lookup_transform_full(
        target_frame=to_frame_rel,
        target_time=rclpy.time.Time(),
    )
```

(continues on next page)
source_frame=from_frame_rel,
source_time=when,
fixed_frame='world',
timeout=rclpy.duration.Duration(seconds=0.05))
except TransformException as ex:

The advanced API for `lookup_transform_full()` takes six arguments:

1. Target frame
2. The time to transform to
3. Source frame
4. The time at which source frame will be evaluated
5. Frame that does not change over time, in this case the `world` frame
6. Time to wait for the target frame to become available

To sum up, tf2 does the following in the background. In the past, it computes the transform from the `carrot1` to the `world`. In the `world` frame, tf2 time travels from the past to now. And at the current time, tf2 computes the transform from the `world` to the `turtle2`.

**Checking the results**

Build the package as usual then let’s run the simulation again, this time with the advanced time-travel API:

```
ros2 launch learning_tf2_py turtle_tf2_fixed_frame_demo_launch.py
```
And yes, the second turtle is directed to where the first carrot was 5 seconds ago!

**Summary**

In this tutorial, you have seen one of the advanced features of tf2. You learned that tf2 can transform data in time and learned how to do that with turtlesim example. tf2 allowed you to go back in time and make frame transformations between old and current poses of turtles by using the advanced `lookup_transform_full()` API.

**Traveling in time (C++)**

**Goal:** Learn about advanced time travel features of tf2.

**Tutorial level:** Intermediate

**Time:** 10 minutes

**Contents**

- Background
Background

In the previous tutorial, we discussed the basics of tf2 and time. This tutorial will take us one step further and expose a powerful tf2 trick: time travel. In short, one of the key features of tf2 library is that it is able to transform data in time as well as in space.

This tf2 time travel feature can be useful for various tasks, like monitoring the pose of the robot for a long period of time or building a follower robot that will follow the “steps” of the leader. We will use that time travel feature to look up transforms back in time and program turtle2 to follow 5 seconds behind carrot1.

Time travel

First, let’s go back to where we ended in the previous tutorial Using time. Go to your learning_tf2_cpp package.

Now, instead of making the second turtle go to where the carrot is now, we will make the second turtle go to where the first carrot was 5 seconds ago. Edit the lookupTransform() call in turtle_tf2_listener.cpp file to

```cpp
rclcpp::Time when = this->get_clock()->now() - rclcpp::Duration(5, 0);
try {
    t = tf_buffer_->lookupTransform(
        toFrameRel,
        fromFrameRel,
        when,
        50ms);
} catch (const tf2::TransformException & ex) {
```

Now if you run this, during the first 5 seconds, the second turtle would not know where to go because we do not yet have a 5-second history of poses of the carrot. But what happens after these 5 seconds? Build the package then let’s just give it a try:

```bash
ros2 launch learning_tf2_cpp turtle_tf2_fixed_frame_demo_launch.py
```
You should now notice that your turtle is driving around uncontrollably like in this screenshot. Let’s try to understand the reason behind that behavior.

1. In our code we asked tf2 the following question: “What was the pose of carrot1 5 seconds ago, relative to turtle2 5 seconds ago?”. This means we are controlling the second turtle based on where it was 5 seconds ago as well as where the first carrot was 5 seconds ago.

2. However, what we really want to ask is: “What was the pose of carrot1 5 seconds ago, relative to the current position of the turtle2?”.

Advanced API for `lookupTransform()`

To ask the tf2 that particular question, we will use an advanced API that gives us the power to say explicitly when to acquire the specified transformations. This is done by calling the `lookupTransform()` method with additional parameters. Your code now would look like this:

```cpp
rclcpp::Time now = this->get_clock()->now();
    rclcpp::Time when = now - rclcpp::Duration(5, 0);
    try {
      t = tf_buffer_ -> lookupTransform(
        toFrameRel,
```
The advanced API for `lookupTransform()` takes six arguments:

1. Target frame
2. The time to transform to
3. Source frame
4. The time at which source frame will be evaluated
5. Frame that does not change over time, in this case the `world` frame
6. Time to wait for the target frame to become available

To sum up, tf2 does the following in the background. In the past, it computes the transform from the `carrot1` to the `world`. In the `world` frame, tf2 time travels from the past to now. And at the current time, tf2 computes the transform from the `world` to the `turtle2`.

**Checking the results**

Build the package then let’s run the simulation again, this time with the advanced time-travel API:

```bash
ros2 launch learning_tf2_cpp turtle_tf2_fixed_frame_demo_launch.py
```
And yes, the second turtle is directed to where the first carrot was 5 seconds ago!

Summary

In this tutorial, you have seen one of the advanced features of tf2. You learned that tf2 can transform data in time and learned how to do that with turtlesim example. tf2 allowed you to go back in time and make frame transformations between old and current poses of turtles by using the advanced `lookupTransform()` API.

Debugging

**Goal:** Learn how to use a systematic approach for debugging tf2 related problems.

**Tutorial level:** Intermediate

**Time:** 10 minutes

**Contents**

- Background
Background

This tutorial walks you through the steps to debug a typical tf2 problem. It will also use many of the tf2 debugging tools, such as `tf2_echo`, `tf2_monitor`, and `view_frames`. This tutorial assumes you have completed the `learning tf2` tutorials.

Debugging example

1 Setting and starting the example

For this tutorial we will set up a demo application that has a number of problems. The goal of this tutorial is to apply a systematic approach to find and tackle these problems. First, let’s create the source file.

Go to the `learning_tf2_cpp` package we created in `tf2 tutorials`. Inside the `src` directory make a copy of the source file `turtle_tf2_listener.cpp` and rename it to `turtle_tf2_listener_debug.cpp`.

Open the file using your preferred text editor, and change line 65 from

```cpp
std::string toFrameRel = "turtle2";
```

to

```cpp
std::string toFrameRel = "turtle3";
```

and change `lookupTransform()` call in lines 73-77 from

```cpp
try {
    t = tf_buffer_->lookupTransform(
        toFrameRel, fromFrameRel,
        tf2::TimePointZero);
} catch (const tf2::TransformException & ex) {
```

to

```cpp
try {
    t = tf_buffer_->lookupTransform(
        toFrameRel, fromFrameRel,
        this->now());
} catch (const tf2::TransformException & ex) {
```

And save changes to the file. In order to run this demo, we need to create a launch file `start_tf2_debug_demo_launch.py` in the `launch` subdirectory of package `learning_tf2_cpp`:
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch.substitutions import LaunchConfiguration
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        DeclareLaunchArgument(
            'target_frame', default_value='turtle1',
            description='Target frame name.'
        ),
        Node(
            package='turtlesim',
            executable='turtlesim_node',
            name='sim',
            output='screen'
        ),
        Node(
            package='learning_tf2_cpp',
            executable='turtle_tf2_broadcaster',
            name='broadcaster1',
            parameters=[
                {'turtlename': 'turtle1'}
            ]
        ),
        Node(
            package='learning_tf2_cpp',
            executable='turtle_tf2_broadcaster',
            name='broadcaster2',
            parameters=[
                {'turtlename': 'turtle2'}
            ]
        ),
        Node(
            package='learning_tf2_cpp',
            executable='turtle_tf2_listener_debug',
            name='listener_debug',
            parameters=[
                {'target_frame': LaunchConfiguration('target_frame')}
            ]
        ),
    ])

Don't forget to add the `turtle_tf2_listener_debug` executable to the `CMakeLists.txt` and build the package.

Now let's run it to see what happens:

```
ros2 launch learning_tf2_cpp start_tf2_debug_demo_launch.py
```

You will now see that the turtlesim came up. At the same time, if you run the `turtle_teleop_key` in another terminal window, you can use the arrow keys to drive the `turtle1` around.
You will also notice that there is a second turtle in the lower, left corner. If the demo would be working correctly, this second turtle should be following the turtle you can command with the arrow keys. However, it is not the case because we have to solve some problems first. You should notice the following message:

```
[turtle_tf2_listener_debug-4] [INFO] [1630223454.942322623] [listener_debug]: Could not transform turtle3 to turtle1: "turtle3" passed to lookupTransform argument target_frame does not exist
```

### 2 Finding the tf2 request

Firstly, we need to find out what exactly we are asking tf2 to do. Therefore, we go into the part of the code that is using tf2. Open the `src/turtle_tf2_listener_debug.cpp` file, and take a look at line 65:

```cpp
std::string to_frame_rel = "turtle3";
```

and lines 73-77:

```cpp
try {
    t = tf_buffer_->lookupTransform(
        toFrameRel, fromFrameRel, 
        this->now());
} catch (const tf2::TransformException & ex) {
    
    Here we do the actual request to tf2. The three arguments tell us directly what we are asking tf2: transform from frame turtle3 to frame turtle1 at time now.
    
Now, let's take a look at why this request to tf2 is failing.

### 3 Checking the frames

Firstly, to find out if tf2 knows about our transform between turtle3 and turtle1, we will use `tf2_echo` tool.

```
ros2 run tf2_ros tf2_echo turtle3 turtle1
```

The output tells us that frame turtle3 does not exist:

```
[INFO] [1630223557.477636052] [tf2_echo]: Waiting for transform turtle3 -> turtle1: Invalid frame ID "turtle3" passed to canTransform argument target_frame - frame does not exist
```

Then what frames do exist? If you like to get a graphical representation of this, use `view_frames` tool.

```
ros2 run tf2_tools view_frames
```

Open the generated `frames.pdf` file to see the following output:
So obviously the problem is that we are requesting transform from frame turtle3, which does not exist. To fix this bug, just replace turtle3 with turtle2 in line 65.

And now stop the running demo, build it, and run it again:

```bash
ros2 launch turtle_tf2 start_debug_demo.launch.py
```

And right away we run into the next problem:

```bash
[turtle_tf2_listener_debug-4] [INFO] [1630223704.617382464] [listener_debug]: Could not transform turtle2 to turtle1: Lookup would require extrapolation into the future.Requested time 1630223704.617054 but the latest data is at time 1630223704.616726, when looking up transform from frame [turtle1] to frame [turtle2]
```

### 4 Checking the timestamp

Now that we solved the frame name problem, it is time to look at the timestamps. Remember, we are trying to get the transform between turtle2 and turtle1 at the current time (i.e., now). To get statistics on the timing, call `tf2_monitor` with corresponding frames.

```bash
ros2 run tf2_ros tf2_monitor turtle2 turtle1
```

The result should look something like this:

```
RESULTS: for turtle2 to turtle1
Chain is: turtle1
Net delay avg = 0.00287347: max = 0.0167241

Frames:
Frame: turtle1, published by <no authority available>, Average Delay: 0.00028533, Max Delay: 0.000755072
```

(continues on next page)
The key part here is the delay for the chain from turtle2 to turtle1. The output shows there is an average delay of about 3 milliseconds. This means that tf2 can only transform between the turtles after 3 milliseconds are passed. So, if we would be asking tf2 for the transformation between the turtles 3 milliseconds ago instead of now, tf2 would be able to give us an answer sometimes. Let's test this quickly by changing lines 73-77 to:

```c++
try {
  t = tf_buffer_->lookupTransform(
    toFrameRel, fromFrameRel,
    this->now() - rclcpp::Duration::from_seconds(0.1));
} catch (const tf2::TransformException & ex) {

In the new code we are asking for the transform between the turtles 100 milliseconds ago. It is usual to use a longer periods, just to make sure that the transform will arrive. Stop the demo, build and run:

```c
ros2 launch turtle_tf2 start_debug_demo.launch.py
```n
And you should finally see the turtle move!
That last fix we made is not really what you want to do, it was just to make sure that was our problem. The real fix would look like this:

```cpp
try {
    t = tf_buffer_->lookupTransform(
        toFrameRel, fromFrameRel,
        tf2::TimePointZero);
} catch (const tf2::TransformException & ex) {
```

Or like this:

```cpp
try {
    t = tf_buffer_->lookupTransform(
        toFrameRel, fromFrameRel,
        tf2::TimePoint());
} catch (const tf2::TransformException & ex) {
```

You can learn more about timeouts in the Using time tutorial, and use them as below:

```cpp
try {
    t = tf_buffer_->lookupTransform(
        toFrameRel, fromFrameRel,
        this->now(),
        rclcpp::Duration::from_seconds(0.05));
} catch (const tf2::TransformException & ex) {
```

Summary

In this tutorial you learned how to use a systematic approach for debugging tf2 related problems. You also learned how to use tf2 debugging tools, such as tf2_echo, tf2_monitor, and view_frames to help you debug those tf2 problems.

Quaternion fundamentals

Goal: Learn the basics of quaternion usage in ROS 2.

Tutorial level: Intermediate

Time: 10 minutes

Contents

- Background
- Prerequisites
- Components of a quaternion
- Quaternion types in ROS 2
- Quaternion operations
  - 1 Think in RPY then convert to quaternion
  - 2 Applying a quaternion rotation
Inverting a quaternion

Relative rotations

Summary

Background

A quaternion is a 4-tuple representation of orientation, which is more concise than a rotation matrix. Quaternions are very efficient for analyzing situations where rotations in three dimensions are involved. Quaternions are used widely in robotics, quantum mechanics, computer vision, and 3D animation.

You can learn more about the underlying mathematical concept on Wikipedia. You can also take a look at an explorable video series Visualizing quaternions made by 3blue1brown.

In this tutorial, you will learn how quaternions and conversion methods work in ROS 2.

Prerequisites

However, this is not a hard requirement and you can stick to any other geometric transformation library that suit you best. You can take a look at libraries like transforms3d, scipy.spatial.transform, pytransform3d, numpy-quaternion or blender.mathutils.

Components of a quaternion

ROS 2 uses quaternions to track and apply rotations. A quaternion has 4 components (x, y, z, w). In ROS 2, w is last, but in some libraries like Eigen, w can be placed at the first position. The commonly-used unit quaternion that yields no rotation about the x/y/z axes is (0, 0, 0, 1), and can be created in a following way:

```cpp
#include <tf2/LinearMath/Quaternion.h>
...

tf2::Quaternion q;
// Create a quaternion from roll/pitch/yaw in radians (0, 0, 0)
q.setRPY(0, 0, 0);
// Print the quaternion components (0, 0, 0, 1)
RCLCPP_INFO(this->get_logger(), "%f %f %f %f",
            q.x(), q.y(), q.z(), q.w());
```

The magnitude of a quaternion should always be one. If numerical errors cause a quaternion magnitude other than one, ROS 2 will print warnings. To avoid these warnings, normalize the quaternion:

```cpp
q.normalize();
```
Quaternion types in ROS 2

ROS 2 uses two quaternion datatypes: `tf2::Quaternion` and its equivalent `geometry_msgs::msg::Quaternion`. To convert between them in C++, use the methods of `tf2::geometry_msgs`.

C++

```cpp
#include <tf2_geometry_msgs/tf2_geometry_msgs.hpp>
...

tf2::Quaternion tf2_quat, tf2_quat_from_msg;
tf2_quat.setRPY(roll, pitch, yaw);
// Convert tf2::Quaternion to geometry_msgs::msg::Quaternion
geometry_msgs::msg::Quaternion msg_quat = tf2::toMsg(tf2_quat);

// Convert geometry_msgs::msg::Quaternion to tf2::Quaternion
tf2::convert(msg_quat, tf2_quat_from_msg);
// or
tf2::fromMsg(msg_quat, tf2_quat_from_msg);
```

Python

```python
from geometry_msgs.msg import Quaternion
...

# Create a list of floats, which is compatible with tf2
# Quaternion methods
quat_tf = [0.0, 1.0, 0.0, 0.0]

# Convert a list to geometry_msgs.msg.Quaternion
msg_quat = Quaternion(x=quat_tf[0], y=quat_tf[1], z=quat_tf[2], w=quat_tf[3])
```

Quaternion operations

1 Think in RPY then convert to quaternion

It's easy for us to think of rotations about axes, but hard to think in terms of quaternions. A suggestion is to calculate target rotations in terms of roll (about an X-axis), pitch (about the Y-axis), and yaw (about the Z-axis), and then convert to a quaternion.

```python
# quaternion_from_euler method is available in turtle_tf2_py/turtle_tf2_py/turtle_tf2_...broadcaster.py
q = quaternion_from_euler(1.5707, 0, -1.5707)
print(f'The quaternion representation is x: {q[0]} y: {q[1]} z: {q[2]} w: {q[3]}')
```
2 Applying a quaternion rotation

To apply the rotation of one quaternion to a pose, simply multiply the previous quaternion of the pose by the quaternion representing the desired rotation. The order of this multiplication matters.

C++

```cpp
#include <tf2_geometry_msgs/tf2_geometry_msgs.hpp>
...

tf2::Quaternion q_orig, q_rot, q_new;
q_orig.setRPY(0.0, 0.0, 0.0);
// Rotate the previous pose by 180° about X
q_rot.setRPY(3.14159, 0.0, 0.0);
q_new = q_rot * q_orig;
q_new.normalize();
```

Python

```python
def quaternion_multiply(q0, q1):
    ""
    Multiplies two quaternions.
    ""
    
    Input
    :param q0: A 4 element array containing the first quaternion (q01, q11, q21, q31)
    :param q1: A 4 element array containing the second quaternion (q02, q12, q22, q32)
    
    # Rotating the previous pose by 180° about X
    q_rot = quaternion_from_euler(3.14159, 0, 0)
    q_new = quaternion_multiply(q_rot, q_orig)
```

3 Inverting a quaternion

An easy way to invert a quaternion is to negate the w-component:

```python
```

4 Relative rotations

Say you have two quaternions from the same frame, q_1 and q_2. You want to find the relative rotation, q_r, that converts q_1 to q_2 in a following manner:

```python
q_2 = q_r * q_1
```

You can solve for q_r similarly to solving a matrix equation. Invert q_1 and right-multiply both sides. Again, the order of multiplication is important:

```python
q_r = q_2 * q_1_inverse
```

Here’s an example to get the relative rotation from the previous robot pose to the current robot pose in python:
Output
:return: A 4 element array containing the final quaternion (q03,q13,q23,q33)

```python
# Extract the values from q0
w0 = q0[0]
x0 = q0[1]
y0 = q0[2]
z0 = q0[3]

# Extract the values from q1
w1 = q1[0]
x1 = q1[1]
y1 = q1[2]
z1 = q1[3]

# Compute the product of the two quaternions, term by term
q0q1_w = w0 * w1 - x0 * x1 - y0 * y1 - z0 * z1
q0q1_x = w0 * x1 + x0 * w1 + y0 * z1 - z0 * y1
q0q1_y = w0 * y1 - x0 * z1 + y0 * w1 + z0 * x1
q0q1_z = w0 * z1 + x0 * y1 - y0 * x1 + z0 * w1

# Create a 4 element array containing the final quaternion
final_quaternion = np.array([q0q1_w, q0q1_x, q0q1_y, q0q1_z])

# Return a 4 element array containing the final quaternion (q02,q12,q22,q32)
return final_quaternion
```

```python
q1_inv[0] = prev_pose.pose.orientation.x
q1_inv[1] = prev_pose.pose.orientation.y
q1_inv[2] = prev_pose.pose.orientation.z
q1_inv[3] = -prev_pose.pose.orientation.w  # Negate for inverse

q2[0] = current_pose.pose.orientation.x
q2[1] = current_pose.pose.orientation.y
q2[2] = current_pose.pose.orientation.z
q2[3] = current_pose.pose.orientation.w

qr = quaternion_multiply(q2, q1_inv)
```
Summary

In this tutorial, you learned about the fundamental concepts of a quaternion and its related mathematical operations, like inversion and rotation. You also learned about its usage examples in ROS 2 and conversion methods between two separate Quaternion classes.

Using stamped datatypes with \texttt{tf2\_ros::MessageFilter}

Goal: Learn how to use \texttt{tf2\_ros::MessageFilter} to process stamped datatypes.

Tutorial level: Intermediate

Time: 10 minutes

Contents

\begin{itemize}
\item Background
\item Tasks
\begin{itemize}
\item 1 Write the broadcaster node of \texttt{PointStamped} messages
\begin{itemize}
\item 1.1 Examine the code
\item 1.2 Write the launch file
\item 1.3 Add an entry point
\item 1.4 Build
\end{itemize}
\item 2 Writing the message filter/listener node
\begin{itemize}
\item 2.1 Examine the code
\item 2.2 Add dependencies
\item 2.3 CMakeLists.txt
\item 2.4 Build
\end{itemize}
\item 3 Run
\end{itemize}

\item Summary
\end{itemize}

Background

This tutorial explains how to use sensor data with tf2. Some real-world examples of sensor data are:

\begin{itemize}
\item cameras, both mono and stereo
\item laser scans
\end{itemize}

Suppose that a new turtle named \texttt{turtle3} is created and it doesn’t have good odometry, but there is an overhead camera tracking its position and publishing it as a \texttt{PointStamped} message in relation to the \texttt{world} frame.

\texttt{turtle1} wants to know where \texttt{turtle3} is compared to itself.

To do this \texttt{turtle1} must listen to the topic where \texttt{turtle3}’s pose is being published, wait until transforms into the desired frame are ready, and then do its operations. To make this easier the \texttt{tf2\_ros::MessageFilter} is very useful.
The `tf2_ros::MessageFilter` will take a subscription to any ROS 2 message with a header and cache it until it is possible to transform it into the target frame.

**Tasks**

1. **Write the broadcaster node of PointStamped messages**

For this tutorial we will set up a demo application which has a node (in Python) to broadcast the `PointStamped` position messages of `turtle3`.

First, let’s create the source file.

Go to the `learning_tf2_py` package we created in the previous tutorial. Inside the `src/learning_tf2_py/learning_tf2_py` directory download the example sensor message broadcaster code by entering the following command:

Linux

```
wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_py/turtle_tf2_py/turtle_tf2_message_broadcaster.py
```

macOS

```
wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_py/turtle_tf2_py/turtle_tf2_message_broadcaster.py
```

Windows

In a Windows command line prompt:

```
curl -sk https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_py/turtle_tf2_py/turtle_tf2_message_broadcaster.py -o turtle_tf2_message_broadcaster.py
```

Or in powershell:

```
curl https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_py/turtle_tf2_py/turtle_tf2_message_broadcaster.py -o turtle_tf2_message_broadcaster.py
```

Open the file using your preferred text editor.

```python
from geometry_msgs.msg import PointStamped
from geometry_msgs.msg import Twist

import rclpy
from rclpy.node import Node

from turtlesim.msg import Pose
from turtlesim.srv import Spawn

class PointPublisher(Node):
    
    def __init__(self):
        super().__init__('turtle_tf2_message_broadcaster')
```

(continues on next page)
# Create a client to spawn a turtle
self.spawner = self.create_client(Spawn, 'spawn')
# Boolean values to store the information
# if the service for spawning turtle is available
self.turtle_spawning_service_ready = False
# if the turtle was successfully spawned
self.turtle_spawned = False
# if the topics of turtle3 can be subscribed
self.turtle_pose_cansubscribe = False

self.timer = self.create_timer(1.0, self.on_timer)

def on_timer(self):
    if self.turtle_spawning_service_ready:
        if self.turtle_spawned:
            self.turtle_pose_cansubscribe = True
        else:
            if self.result.done():
                self.get_logger().info(f'Successfully spawned {self.result.result().name}')
                self.turtle_spawned = True
            else:
                self.get_logger().info('Spawn is not finished')
    else:
        if self.spawner.service_is_ready():
            # Initialize request with turtle name and coordinates
            # Note that x, y and theta are defined as floats in turtlesim/srv/Spawn
            request = Spawn.Request()
            request.name = 'turtle3'
            request.x = 4.0
            request.y = 2.0
            request.theta = 0.0
            # Call request
            self.result = self.spawner.call_async(request)
            self.turtle_spawning_service_ready = True
        else:
            # Check if the service is ready
            self.get_logger().info('Service is not ready')

    if self.turtle_pose_cansubscribe:
        self.vel_pub = self.create_publisher(Twist, 'turtle3/cmd_vel', 10)
        self.sub = self.create_subscription(Pose, 'turtle3/pose', self.handle_turtle_pose, 10)
        self.pub = self.create_publisher(PointStamped, 'turtle3/turtle_point_stamped', 10)

    def handle_turtle_pose(self, msg):
        vel_msg = Twist()
        vel_msg.linear.x = 1.0
        vel_msg.angular.z = 1.0
        self.vel_pub.publish(vel_msg)
ps = PointStamped()
ps.header.stamp = self.get_clock().now().to_msg()
ps.header.frame_id = 'world'
ps.point.x = msg.x
ps.point.y = msg.y
ps.point.z = 0.0
self.pub.publish(ps)

def main():
rclpy.init()
node = PointPublisher()
try:
    rclpy.spin(node)
except KeyboardInterrupt:
    pass
rclpy.shutdown()

1.1 Examine the code

Now let's take a look at the code. First, in the on_timer callback function, we spawn the turtle3 by asynchronously calling the Spawn service of turtlesim, and initialize its position at (4, 2, 0), when the turtle spawning service is ready.

```python
# Initialize request with turtle name and coordinates
# Note that x, y and theta are defined as floats in turtlesim/srv/Spawn
request = Spawn.Request()
request.name = 'turtle3'
request.x = 4.0
request.y = 2.0
request.theta = 0.0
# Call request
self.result = self.spawner.call_async(request)
```

Afterward, the node publishes the topic turtle3/cmd_vel, topic turtle3/turtle_point_stamped, and subscribes to topic turtle3/pose and runs callback function handle_turtle_pose on every incoming message.

```python
self.vel_pub = self.create_publisher(Twist, '/turtle3/cmd_vel', 10)
sel.sub = self.create_subscription(Pose, '/turtle3/pose', self.handle_turtle_pose, 10)
sel.pub = self.create_publisher(PointStamped, '/turtle3/turtle_point_stamped', 10)
```

Finally, in the callback function handle_turtle_pose, we initialize the Twist messages of turtle3 and publish them, which will make the turtle3 move along a circle. Then we fill up the PointStamped messages of turtle3 with incoming Pose messages and publish them.

```python
vel_msg = Twist()
vel_msg.linear.x = 1.0
vel_msg.angular.z = 1.0
self.vel_pub.publish(vel_msg)
```
ps = PointStamped()
ps.header.stamp = self.get_clock().now().to_msg()
ps.header.frame_id = 'world'
ps.point.x = msg.x
ps.point.y = msg.y
ps.point.z = 0.0
self.pub.publish(ps)

1.2 Write the launch file

In order to run this demo, we need to create a launch file `turtle_tf2_sensor_message_launch.py` in the launch subdirectory of package `learning_tf2_py`:

```python
from launch import LaunchDescription
from launch_ros.actions import Node
from launch import *
from launch_ros.actions import *

def generate_launch_description():
    return LaunchDescription(
        DeclareLaunchArgument(
            'target_frame', default_value='turtle1',
            description='Target frame name.'
        ),
        Node(
            package='turtlesim',
            executable='turtlesim_node',
            name='sim',
            output='screen'
        ),
        Node(
            package='turtle_tf2_py',
            executable='turtle_tf2_broadcaster',
            name='broadcaster1',
            parameters=[
                {'turtlename': 'turtle1'}
            ]
        ),
        Node(
            package='turtle_tf2_py',
            executable='turtle_tf2_broadcaster',
            name='broadcaster2',
            parameters=[
                {'turtlename': 'turtle3'}
            ]
        ),
        Node(
            package='turtle_tf2_py',
            executable='turtle_message_broadcaster',
            name='message_broadcaster',
        ),
    )
```
1.3 Add an entry point

To allow the `ros2 run` command to run your node, you must add the entry point to `setup.py` (located in the `src/learning_tf2_py` directory).

Add the following line between the 'console_scripts' brackets:

```
'turtle_tf2_message_broadcaster = learning_tf2_py.turtle_tf2_message_broadcaster:main',
```

1.4 Build

Run `rosdep` in the root of your workspace to check for missing dependencies.

Linux

```
rosdep install -i --from-path src --rosdistro iron -y
```

macOS

Rosdep only runs on Linux, so you will need to install `geometry_msgs` and `turtlesim` dependencies yourself.

Windows

Rosdep only runs on Linux, so you will need to install `geometry_msgs` and `turtlesim` dependencies yourself.

And then we can build the package:

Linux

```
colcon build --packages-select learning_tf2_py
```

macOS

```
colcon build --packages-select learning_tf2_py
```

Windows

```
colcon build --merge-install --packages-select learning_tf2_py
```

2 Writing the message filter/listener node

Now, to get the streaming `PointStamped` data of `turtle3` in the frame of `turtle1` reliably, we will create the source file of the message filter/listener node.

Go to the `learning_tf2_cpp` package we created in the previous tutorial. Inside the `src/learning_tf2_cpp/src` directory download file `turtle_tf2_message_filter.cpp` by entering the following command:

Linux

```
wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/turtle_tf2_message_filter.cpp
```
macOS

```
wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/
  turtle_tf2_message_filter.cpp
```

Windows

In a Windows command line prompt:

```
curl -sk wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_  
cpp/src/turtle_tf2_message_filter.cpp -o turtle_tf2_message_filter.cpp
```

Or in powershell:

```
curl https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/
  turtle_tf2_message_filter.cpp -o turtle_tf2_message_filter.cpp
```

Open the file using your preferred text editor.

```
#include <chrono>
#include <memory>
#include <string>
#include "geometry_msgs/msg/point_stamped.hpp"
#include "message_filters/subscriber.h"
#include "rclcpp/rclcpp.hpp"
#include "tf2_ros/buffer.h"
#include "tf2_ros/create_timer_ros.h"
#include "tf2_ros/message_filter.h"
#include "tf2_ros/transform_listener.h"
#ifdef TF2_CPP_HEADERS
  #include "tf2_geometry_msgs/tf2_geometry_msgs.hpp"
#else
  #include "tf2_geometry_msgs/tf2_geometry_msgs.h"
#endif
using namespace std::chrono_literals;

class PoseDrawer : public rclcpp::Node
{
public:
  PoseDrawer()
    : Node("turtle_tf2_pose_drawer")
  {
    // Declare and acquire 'target_frame' parameter
    target_frame_ = this->declare_parameter<std::string>("target_frame", "turtle1");

    std::chrono::duration<int> buffer_timeout(1);

    tf2_buffer_ = std::make_shared<tf2_ros::Buffer>(this->get_clock());
    // Create the timer interface before call to waitForTransform,
    // to avoid a tf2_ros::CreateTimerInterfaceException exception
    auto timer_interface = std::make_shared<tf2_ros::CreateTimerROS>(
      this->get_node_base_interface(),
      this->get_node_timers_interface());

    (continues on next page)
```
tf2_buffer_->setCreateTimerInterface(timer_interface);
  tf2_listener_ =
    std::make_shared<tf2_ros::TransformListener>(*tf2_buffer_);

  point_sub_.subscribe(this, "/turtle3/turtle_point_stamped");
  tf2_filter_ = std::make_shared<tf2_ros::MessageFilter<geometry_
    msgs::msg::PointStamped>>(
    point_sub_, *tf2_buffer_, target_frame_, 100, this->get_node_logging_interface(),
    this->get_node_clock_interface(), buffer_timeout);
  // Register a callback with tf2_ros::MessageFilter to be called when transforms are
  // available
  tf2_filter_->registerCallback(&PoseDrawer::msgCallback, this);
}

private:
  void msgCallback(const geometry_msgs::msg::PointStamped::SharedPtr point_ptr)
  {
    geometry_msgs::msg::PointStamped point_out;
    try {
      tf2_buffer_->transform(*point_ptr, point_out, target_frame_);
      RCLCPP_INFO(
        this->get_logger(), "Point of turtle3 in frame of turtle1: x:%f y:%f z:%f\n",
        point_out.point.x, point_out.point.y, point_out.point.z);
    } catch (const tf2::TransformException & ex) {
      RCLCPP_WARN(
        this->get_logger(), "Failure %s\n", ex.what());
    }
  }

  std::string target_frame_;  
  std::shared_ptr<tf2_ros::Buffer> tf2_buffer_;  
  std::shared_ptr<tf2_ros::TransformListener> tf2_listener_;  
  message_filters::Subscriber<geometry_msgs::msg::PointStamped> point_sub_;  
  std::shared_ptr<tf2_ros::MessageFilter<geometry_msgs::msg::PointStamped>> tf2_filter_;  
};

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<PoseDrawer>());
  rclcpp::shutdown();
  return 0;
}
2.1 Examine the code

First, you must include the `tf2_ros::MessageFilter` headers from the `tf2_ros` package, as well as the previously used `tf2` and `ros2` related headers.

```
#include "geometry_msgs/msg/point_stamped.hpp"
#include "message_filters/subscriber.h"
#include "rclcpp/rclcpp.h"
#include "tf2_ros/buffer.h"
#include "tf2_ros/create_timer_ros.h"
#include "tf2_ros/message_filter.h"
#include "tf2_ros/transform_listener.h"
#ifdef TF2_CPP_HEADERS
    #include "tf2_geometry_msgs/tf2_geometry_msgs.hpp"
#else
    #include "tf2_geometry_msgs/tf2_geometry_msgs.h"
#endif
```

Second, there needs to be persistent instances of `tf2_ros::Buffer`, `tf2_ros::TransformListener` and `tf2_ros::MessageFilter`.

```
std::string target_frame_;  
std::shared_ptr<tf2_ros::Buffer> tf2_buffer_;  
std::shared_ptr<tf2_ros::TransformListener> tf2_listener_;  
message_filters::Subscriber<geometry_msgs::msg::PointStamped> point_sub_;  
std::shared_ptr<tf2_ros::MessageFilter<geometry_msgs::msg::PointStamped>> tf2_filter_;  
```

Third, the ROS 2 `message_filters::Subscriber` must be initialized with the topic. And the `tf2_ros::MessageFilter` must be initialized with that `Subscriber` object. The other arguments of note in the `MessageFilter` constructor are the `target_frame` and the callback function. The target frame is the frame into which it will make sure `canTransform` will succeed. And the callback function is the function that will be called when the data is ready.

```
PoseDrawer()
    : Node("turtle_tf2_pose_drawer")
{
    // Declare and acquire 'target_frame' parameter
    target_frame_ = this->declare_parameter<std::string>("target_frame", "turtle1");

    std::chrono::duration<int> buffer_timeout(1);

    tf2_buffer_ = std::make_shared<tf2_ros::Buffer>(this->get_clock());
    // Create the timer interface before call to waitForTransform.
    // to avoid a tf2_ros::CreateTimerInterfaceException exception
    auto timer_interface = std::make_shared<tf2_ros::CreateTimerROS>(
        this->get_node_base_interface(),
        this->get_node_timers_interface());
    tf2_buffer_->setCreateTimerInterface(timer_interface);
    tf2_listener_ = std::make_shared<tf2_ros::TransformListener>(*tf2_buffer_);

    point_sub_.subscribe(this, "/turtle3/turtle_point_stamped");
    tf2_filter_ = std::make_shared<tf2_ros::MessageFilter<geometry_msgs::msg::PointStamped>>(
            (continues on next page))
And last, the callback method will call `tf2_buffer_->transform` when the data is ready and print output to the console.

```cpp
private:
    void msgCallback(const geometry_msgs::msg::PointStamped::SharedPtr point_ptr)
    {
        geometry_msgs::msg::PointStamped point_out;
        try {
            tf2_buffer_->transform(*point_ptr, point_out, target_frame_);
            RCLCPP_INFO(this->get_logger(), "Point of turtle3 in frame of turtle1: x:%f y:%f z:%f\n",
                        point_out.point.x, point_out.point.y, point_out.point.z);
        } catch (const tf2::TransformException & ex) {
            RCLCPP_WARN(this->get_logger(), "Failure %s\n", ex.what());
        }
    }
```

### 2.2 Add dependencies

Before building the package `learning_tf2_cpp`, please add two another dependencies in the `package.xml` file of this package:

```xml
<depend>message Filters</depend>
<depend>tf2_geometry_msgs</depend>
```

### 2.3 CMakeLists.txt

And in the `CMakeLists.txt` file, add two lines below the existing dependencies:

```cmake
find_package(message_filters REQUIRED)
find_package(tf2_geometry_msgs REQUIRED)
```

The lines below will deal with differences between ROS distributions:

```cmake
if(TARGET tf2_geometry_msgs::tf2_geometry_msgs)
    get_target_property(_include_dirs tf2_geometry_msgs::tf2_geometry_msgs INTERFACE_INCLUDE_DIRS)
else()
    set(_include_dirs ${tf2_geometry_msgs_INCLUDE_DIRS})
```

(continues on next page)
endif()

find_file(TF2_CPP_HEADERS
  NAMES tf2_geometry_msgs.hpp
  PATHS ${_include_dirs}
  NO_CACHE
  PATH_SUFFIXES tf2_geometry_msgs
)

After that, add the executable and name it turtle_tf2_message_filter, which you'll use later with ros2 run.

add_executable(turtle_tf2_message_filter src/turtle_tf2_message_filter.cpp)
ament_target_dependencies(
  turtle_tf2_message_filter
  geometry_msgs
  message_filters
  rclcpp
  tf2
  tf2_geometry_msgs
  tf2_ros
)

if(EXISTS ${TF2_CPP_HEADERS})
  target_compile_definitions(turtle_tf2_message_filter PUBLIC -DTF2_CPP_HEADERS)
endif()

Finally, add the install(TARGETS...) section (below other existing nodes) so ros2 run can find your executable:

install(TARGETS
  turtle_tf2_message_filter
  DESTINATION lib/${PROJECT_NAME})

2.4 Build

Run rosdep in the root of your workspace to check for missing dependencies.

Linux

rosdep install -i --from-path src --rosdistro iron -y

macOS

rosdep only runs on Linux, so you will need to install geometry_msgs and turtlesim dependencies yourself

Windows

rosdep only runs on Linux, so you will need to install geometry_msgs and turtlesim dependencies yourself

Now open a new terminal, navigate to the root of your workspace, and rebuild the package with command:

Linux

colcon build --packages-select learning_tf2_cpp

macOS
Vulcanexus Documentation, Release 1.0.0

```
colcon build --packages-select learning_tf2_cpp
```

Windows
```
colcon build --merge-install --packages-select learning_tf2_cpp
```

Open a new terminal, navigate to the root of your workspace, and source the setup files:

Linux
```
. install/setup.bash
```

macOS
```
. install/setup.bash
```

Windows
```
# CMD
call install\setup.bat
```
```
# PowerShell
.
\install\setup.ps1
```

3 Run

First we need to run several nodes (including the broadcaster node of PointStamped messages) by launching the launch file `turtle_tf2_sensor_message_launch.py`:

```
ros2 launch learning_tf2_py turtle_tf2_sensor_message_launch.py
```

This will bring up the turtlesim window with two turtles, where turtle3 is moving along a circle, while turtle1 isn’t moving at first. But you can run the `turtle_teleop_key` node in another terminal to drive turtle1 to move:

```
ros2 run turtlesim turtle_teleop_key
```

Now if you echo the topic `turtle3/turtle_pointStamped`:

```
ros2 topic echo /turtle3/turtle_point_stamped
```

Then there will be output like this:
When the demo is running, open another terminal and run the message filter/listener node:

`ros2 run learning_tf2_cpp turtle_tf2_message_filter`

If it's running correctly you should see streaming data like this:

```
[INFO] [1630016162.006173900] [turtle_tf2_pose_drawer]: Point of turtle3 in frame of turtle1: x:-6.493231 y:-2.961614 z:0.000000

[INFO] [1630016162.006291983] [turtle_tf2_pose_drawer]: Point of turtle3 in frame of turtle1: x:-6.472169 y:-3.004742 z:0.000000

[INFO] [1630016162.006326234] [turtle_tf2_pose_drawer]: Point of turtle3 in frame of turtle1: x:-6.479420 y:-2.990479 z:0.000000

[INFO] [1630016162.006355644] [turtle_tf2_pose_drawer]: Point of turtle3 in frame of turtle1: x:-6.486441 y:-2.976102 z:0.000000
```
Summary

In this tutorial you learned how to use sensor data/messages in tf2. Specifically speaking, you learned how to publish PointStamped messages on a topic, and how to listen to the topic and transform the frame of PointStamped messages with tf2_ros::MessageFilter.

Workspace setup

If you have not yet created a workspace in which to complete the tutorials, follow this tutorial.

Learning tf2

1. **Introduction to tf2.**
   This tutorial will give you a good idea of what tf2 can do for you. It shows off some of the tf2 power in a multi-robot example using turtlesim. This also introduces using tf2_echo, view_frames, and rviz.

2. Writing a static broadcaster *(Python) (C++)*.  
   This tutorial teaches you how to broadcast static coordinate frames to tf2.

3. Writing a broadcaster *(Python) (C++)*.  
   This tutorial teaches you how to broadcast the state of a robot to tf2.

4. Writing a listener *(Python) (C++)*.  
   This tutorial teaches you how to use tf2 to get access to frame transformations.

5. Adding a frame *(Python) (C++)*.  
   This tutorial teaches you how to add an extra fixed frame to tf2.

6. Using time *(Python) (C++)*.  
   This tutorial teaches you to use the timeout in lookup_transform function to wait for a transform to be available on the tf2 tree.

7. Traveling in time *(Python) (C++)*.  
   This tutorial teaches you about advanced time travel features of tf2.

Debugging tf2

1. **Quaternion fundamentals.**
   This tutorial teaches you basics of quaternion usage in ROS 2.

2. **Debugging tf2 problems.**
   This tutorial teaches you about a systematic approach for debugging tf2 related problems.
Testing

Why automatic tests?

Here are some of the many good reasons why should we have automated tests:

- You can make incremental updates to your code more quickly. ROS has hundreds of packages with many inter-dependencies, so it can be hard to anticipate the problems a small change might cause. If your change passes the unit tests, you can be more confident that you haven't introduced problems — or at least the problems aren't your fault.

- You can refactor your code with greater confidence. Passing the unit tests verifies that you haven't introduced any bugs while refactoring. This gives you this wonderful freedom from change fear!

- It leads to better designed code. Unit tests force you to write your code so that it can be more easily tested. This often means keeping your underlying functions and framework separate, which is one of our design goals with ROS code.

- They prevent recurring bugs (bug regressions). It's a good practice to write a unit test for every bug you fix. In fact, write the unit test before you fix the bug. This will help you to precisely, or even deterministically, reproduce the bug, and much more precisely understand what the problem is. As a result, you will also create a better patch, which you can then test with your regression test to verify that the bug is fixed. That way the bug won't accidentally get reintroduced if the code gets modified later on. It also means that it will be easier to convince the reviewer of the patch that the problem is solved, and the contribution is of high quality.

- Other people can work on your code more easily (an automatic form of documentation). It can be hard to figure out whether or not you've broken someone else's code when you make a change. The unit tests are a tool for other developers to validate their changes. Automatic tests document your coding decisions, and communicate to other developers automatically about their violation. Thus tests become documentation for your code — a documentation that does not need to be read for the most time, and when it does need to be inspected the test system will precisely indicate what to read (which tests fail). By writing automatic tests you make other contributors faster. This improves the entire ROS project.

- It is much easier to become a contributor to ROS if we have automated unit tests. It is very difficult for new external developers to contribute to your components. When they make changes to code, they are often doing it in the blind, driven by a lot of guesswork. By providing a harness of automated tests, you help them in the task. They get immediate feedback for their changes. It becomes easier to contribute to a project, and new contributors to join more easily. Also their first contributions are of higher quality, which decreases the workload on maintainers. A win-win!

- Automatic tests simplify maintainership. Especially for mature packages, which change more slowly, and mostly need to be updated to new dependencies, an automatic test suite helps to very quickly establish whether the package still works. This makes it much easier to decide whether the package is still supported or not.

- Automatic tests amplify the value of Continuous Integration. Regression tests, along with normal scenario-based requirements tests, contribute to overall body of automated tests for your component. Your component is better tested against evolution of other APIs that it depends on (CI servers will tell you better and more precisely what problems develop in your code).

Perhaps the most important benefit of writing tests is that tests make you a good citizen. Tests influence quality in the long term. It is a well accepted practice in many open-source projects. By writing regressions tests, you are contributing
to long term quality of the ROS ecosystem.

**Is this all coming for free?**

Of course, there is never free lunch. To get the benefits of testing, some investment is necessary.

- You need to develop a test, which sometimes may be difficult or costly. Sometimes it might also be nontrivial, as the test should be automatic. Things get particularly hairy if your tests should involve special hardware (they should not: try to use simulation, mock the hardware, or narrow down the test to a smaller software problem) or require external environment, for instance human operators.
- Regression tests and other automatic tests need to be maintained. When the design of the component changes, a lot of tests become invalidated (for instance they no longer compile, or throw runtime exceptions related to the API design). These tests fail not only because the redesign re-introduced bugs but also because they need to be updated to the new design. Occasionally, with bigger redesigns, old regression tests should be dropped.
- Large bodies of tests can take a long time to run, which can increase Continuous Integration server costs.

**Available Tutorials:**

**Running Tests in ROS 2 from the Command Line**

**Build and run your tests**

To compile and run the tests, simply run the `test` verb from `colcon`.

```
colon test --ctest-args tests [package_selection_args]
```

(where `package_selection_args` are optional package selection arguments for `colcon` to limit which packages are built and run)

*Sourcing the workspace* before testing should not be necessary. `colcon` `test` makes sure that the tests run with the right environment, have access to their dependencies, etc.

**Examine Test Results**

To see the results, simply run the `test-result` verb from `colcon`.

```
colon test-result --all
```

To see the exact test cases which fail, use the `--verbose` flag:

```
colon test-result --all --verbose
```
Writing Basic Tests with C++ with GTest

Starting point: we’ll assume you have a basic ament_cmake package set up already and you want to add some tests to it.

In this tutorial, we’ll be using gtest.

Package Setup

Source Code

We’ll start off with our code in a file called test/tutorial_test.cpp

```
#include <gtest/gtest.h>

TEST(package_name, a_first_test)
{
  ASSERT_EQ(4, 2 + 2);
}

int main(int argc, char** argv)
{
  testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

package.xml

Add the following line to package.xml

```
<test_depend>ament_cmake_gtest</test_depend>
```

CMakeLists.txt

```
if(BUILD_TESTING)
  find_package(ament_cmake_gtest REQUIRED)
  ament_add_gtest(${PROJECT_NAME}_tutorial_test test/tutorial_test.cpp)
  target_include_directories(${PROJECT_NAME}_tutorial_test PUBLIC
    ${<BUILD_INTERFACE}:${CMAKE_CURRENT_SOURCE_DIR}/include>
    ${<INSTALL_INTERFACE:include>}
  )
  ament_target_dependencies(${PROJECT_NAME}_tutorial_test
    std_msgs
  )
  target_link_libraries(${PROJECT_NAME}_tutorial_test name_of_local_library)
endif()
```

The testing code is wrapped in the if/endif block to avoid building tests where possible. ament_add_gtest functions much like add_executable so you’ll need to call target_include_directories, ament_target_dependencies and target_link_libraries as you normally would.
Running Tests

See the tutorial on how to run tests from the command line for more information on running the tests and inspecting the test results.

Writing Basic Tests with Python

Starting point: we’ll assume you have a basic ament_python package set up already and you want to add some tests to it.

If you are using ament_cmake_python, refer to the the ament_cmake_python docs for how to make tests discoverable. The test contents and invocation with colcon remain the same.

Package Setup

setup.py

Your setup.py must a test dependency on pytest within the call to setup(...):

```python
tests_require=['pytest'],
```

Test Files and Folders

Your test code needs to go in a folder named tests in the root of your package.

Any file that contains tests that you want to run must have the pattern test_FOO.py where FOO can be replaced with anything.

Example package layout:

```plaintext
awesome_ros_package/
   awesome_ros_package/
      __init__.py
      fozzie.py
   package.xml
   setup.cfg
   setup.py
   tests/
      test_init.py
      test_copyright.py
      test_fozzie.py
```
**Test Contents**

You can now write tests to your heart’s content. There are plenty of resources on pytest, but in short, you can write functions with the `test_` prefix and include whatever assert statements you’d like.

```python
def test_math():
    assert 2 + 2 == 5  # This should fail for most mathematical systems
```

**Running Tests**

See the tutorial on how to run tests from the command line for more information on running the tests and inspecting the test results.

**Special Commands**

Beyond the standard colcon testing commands you can also specify arguments to the pytest framework from the command line with the `--pytest-args` flag. For example, you can specify the name of the function to run with

```
colcon test --packages-select <name-of-pkg> --pytest-args -k name_of_the_test_function
```

To see the pytest output while running the tests, use these flags:

```
colcon test --event-handlers console_cohesion+
```

**URDF**

URDF (Unified Robot Description Format) is a file format for specifying the geometry and organization of robots in ROS.

**Building a visual robot model from scratch**

**Goal:** Learn how to build a visual model of a robot that you can view in Rviz

**Tutorial level:** Intermediate

**Time:** 20 minutes

**Contents**

- One Shape
- Multiple Shapes
- Origins
- Material Girl
- Finishing the Model
Note: This tutorial assumes you know how to write well-formatted XML code

In this tutorial, we’re going to build a visual model of a robot that vaguely looks like R2D2. In later tutorials, you’ll learn how to articulate the model, add in some physical properties, and generate neater code with xacro, but for now, we’re going to focus on getting the visual geometry correct.

Before continuing, make sure you have the joint_state_publisher package installed. If you installed urdf_tutorial binaries, this should already be the case. If not, please update your installation to include that package (use rosdep to check).

All of the robot models mentioned in this tutorial (and the source files) can be found in the urdf_tutorial package.

One Shape

First, we’re just going to explore one simple shape. Here’s about as simple as a urdf as you can make. [Source: 01-myfirst.urdf]

```xml
<?xml version="1.0"?>
<robot name="myfirst">
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.6" radius="0.2"/>
      </geometry>
    </visual>
  </link>
</robot>
```

To translate the XML into English, this is a robot with the name myfirst, that contains only one link (a.k.a. part), whose visual component is just a cylinder 0.6 meters long with a 0.2 meter radius. This may seem like a lot of enclosing tags for a simple “hello world” type example, but it will get more complicated, trust me.

To examine the model, launch the display.launch.py file:

```
ros2 launch urdf_tutorial display.launch.py model:=urdf/01-myfirst.urdf
```

This does three things:

- Loads the specified model and saves it as a parameter
- Runs nodes to publish sensor_msgs/msg/JointState and transforms (more on these later)
- Starts Rviz with a configuration file

Note that the launch command above assumes that you are executing it from the urdf_tutorial package directory (ie: the urdf directory is a direct child of the current working directory). If that is not the case, the relative path to 01-myfirst.urdf will not be valid, and you’ll receive an error as soon as the launcher tries to load the urdf as a parameter.

A slightly modified argument allows this to work regardless of the current working directory:

```
ros2 launch urdf_tutorial display.launch.py model:=`ros2 pkg prefix --share urdf_˓tutorial`/urdf/01-myfirst.urdf
```

You’ll have to change all example launch commands given in these tutorials if you are not running them from the urdf_tutorial package location.
After launching `display.launch.py`, you should end up with RViz showing you the following:

![RViz screenshot](image)

**Things to note:**

- The fixed frame is the transform frame where the center of the grid is located. Here, it’s a frame defined by our one link, base_link.
- The visual element (the cylinder) has its origin at the center of its geometry as a default. Hence, half the cylinder is below the grid.

**Multiple Shapes**

Now let’s look at how to add multiple shapes/links. If we just add more link elements to the urdf, the parser won’t know where to put them. So, we have to add joints. Joint elements can refer to both flexible and inflexible joints. We’ll start with inflexible, or fixed joints. [Source: 02-multipleshapes.urdf]

```xml
<?xml version="1.0"?>
<robot name="multipleshapes">
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.6" radius="0.2"/>
      </geometry>
    </visual>
  </link>
</robot>
```

(continues on next page)
• Note how we defined a 0.6m x 0.1m x 0.2m box

• The joint is defined in terms of a parent and a child. URDF is ultimately a tree structure with one root link. This means that the leg’s position is dependent on the base_link’s position.

```xml
<robot>
  <link name="right_leg">
    <visual>
      <geometry>
        <box size="0.6 0.1 0.2"/>
      </geometry>
    </visual>
  </link>

  <joint name="base_to_right_leg" type="fixed">
    <parent link="base_link"/>
    <child link="right_leg"/>
  </joint>
</robot>
```

Both of the shapes overlap with each other, because they share the same origin. If we want them not to overlap we must define more origins.

```bash
ros2 launch urdf_tutorial display.launch.py model:=urdf/02-multipleshapes.urdf
```
Origins

R2D2’s leg attaches to the top half of his torso, on the side. So that’s where we specify the origin of the JOINT to be. Also, it doesn’t attach to the middle of the leg, it attaches to the upper part, so we must offset the origin for the leg as well. We also rotate the leg so it is upright. [Source: 03-origins.urdf]

```xml
<?xml version="1.0"?>
<robot name="origins">
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.6" radius="0.2"/>
      </geometry>
    </visual>
  </link>
  <link name="right_leg">
    <visual>
      <geometry>
        <box size="0.6 0.1 0.2"/>
      </geometry>
      <origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>
    </visual>
  </link>
  <joint name="base_to_right_leg" type="fixed">
    <parent link="base_link"/>
    <child link="right_leg"/>
    <origin xyz="0 -0.22 0.25"/>
  </joint>
</robot>
```

- Let’s start by examining the joint’s origin. It is defined in terms of the parent’s reference frame. So we are -0.22 meters in the y direction (to our left, but to the right relative to the axes) and 0.25 meters in the z direction (up). This means that the origin for the child link will be up and to the right, regardless of the child link’s visual origin tag. Since we didn’t specify a rpy (roll pitch yaw) attribute, the child frame will be default have the same orientation as the parent frame.

- Now, looking at the leg’s visual origin, it has both a xyz and rpy offset. This defines where the center of the visual element should be, relative to its origin. Since we want the leg to attach at the top, we offset the origin down by setting the z offset to be -0.3 meters. And since we want the long part of the leg to be parallel to the z axis, we rotate the visual part PI/2 around the Y axis.

`ros2 launch urdf_tutorial display.launch.py model:=urdf/03-origins.urdf`
• The launch file runs packages that will create TF frames for each link in your model based on your URDF. Rviz uses this information to figure out where to display each shape.

• If a TF frame does not exist for a given URDF link, then it will be placed at the origin in white (ref. related question).

Material Girl

“Alright,” I hear you say. “That’s very cute, but not everyone owns a B21. My robot and R2D2 are not red!” That’s a good point. Let’s take a look at the material tag. [Source: 04-materials.urdf]

```xml
<?xml version="1.0"?>
<robot name="materials">
  <material name="blue">
    <color rgba="0 0 0.8 1"/>
  </material>
  <material name="white">
    <color rgba="1 1 1 1"/>
  </material>
  <link name="base_link">
    <visual>
      <geometry>
```

(continues on next page)
The body is now blue. We’ve defined a new material called “blue”, with the red, green, blue and alpha channels defined as 0,0,0.8 and 1 respectively. All of the values can be in the range [0,1]. This material is then referenced by the base_link’s visual element. The white material is defined similarly.

You could also define the material tag from within the visual element, and even reference it in other links. No one will even complain if you redefine it though.

You can also use a texture to specify an image file to be used for coloring the object

```bash
ros2 launch urdf_tutorial display.launch.py model:=urdf/04-materials.urdf
```
Finishing the Model

Now we finish the model off with a few more shapes: feet, wheels, and head. Most notably, we add a sphere and a some meshes. We’ll also add few other pieces that we’ll use later. [Source: 05-visual.urdf]

```xml
<?xml version="1.0"?>
<robot name="visual">
    <material name="blue">
        <color rgba="0 0 0.8 1"/>
    </material>
    <material name="black">
        <color rgba="0 0 0 1"/>
    </material>
    <material name="white">
        <color rgba="1 1 1 1"/>
    </material>
    <link name="base_link">
        <visual>
            <geometry>
                <cylinder length="0.6" radius="0.2"/>
            </geometry>
        </visual>
    </link>
</robot>
```

(continues on next page)
<visual>
</visual>
</link>

<link name="right_leg">
<visual>
    <geometry>
        <box size="0.6 0.1 0.2"/>
    </geometry>
    <origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>
    <material name="white"/>
</visual>
</link>

<joint name="base_to_right_leg" type="fixed">
    <parent link="base_link"/>
    <child link="right_leg"/>
    <origin xyz="0 -0.22 0.25"/>
</joint>

<link name="right_base">
<visual>
    <geometry>
        <box size="0.4 0.1 0.1"/>
    </geometry>
    <material name="white"/>
</visual>
</link>

<joint name="right_base_joint" type="fixed">
    <parent link="right_leg"/>
    <child link="right_base"/>
    <origin xyz="0 0 -0.6"/>
</joint>

<link name="right_front_wheel">
<visual>
    <origin rpy="1.57075 0 0" xyz="0 0 0"/>
    <geometry>
        <cylinder length="0.1" radius="0.035"/>
    </geometry>
    <material name="black"/>
</visual>
</link>

<joint name="right_front_wheel_joint" type="fixed">
    <parent link="right_base"/>
    <child link="right_front_wheel"/>
    <origin rpy="0 0 0" xyz="0.133333333333 0 -0.085"/>
</joint>

<link name="right_back_wheel">
<visual>
    <origin rpy="1.57075 0 0" xyz="0 0 0"/>
</visual>
</link>

<joint name="right_back_wheel_joint" type="fixed">
    <parent link="right_base"/>
    <child link="right_back_wheel"/>
    <origin rpy="0 0 0" xyz="0.133333333333 0 -0.085"/>
</joint>

(continues on next page)
<geometry>
  <cylinder length="0.1" radius="0.035"/>
</geometry>

<material name="black"/>
</visual>
</link>

<joint name="right_back_wheel_joint" type="fixed">
  <parent link="right_base"/>
  <child link="right_back_wheel"/>
  <origin rpy="0 0 0" xyz="-0.133333333333 0 -0.085"/>
</joint>

<link name="left_leg">
  <visual>
    <geometry>
      <box size="0.6 0.1 0.2"/>
    </geometry>
    <origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>
    <material name="white"/>
  </visual>
</link>

<joint name="base_to_left_leg" type="fixed">
  <parent link="base_link"/>
  <child link="left_leg"/>
  <origin xyz="0 0.22 0.25"/>
</joint>

<link name="left_base">
  <visual>
    <geometry>
      <box size="0.4 0.1 0.1"/>
    </geometry>
    <material name="white"/>
  </visual>
</link>

<joint name="left_base_joint" type="fixed">
  <parent link="left_leg"/>
  <child link="left_base"/>
  <origin xyz="0 0 -0.6"/>
</joint>

<link name="left_front_wheel">
  <visual>
    <origin rpy="1.57075 0 0" xyz="0 0 0"/>
    <geometry>
      <cylinder length="0.1" radius="0.035"/>
    </geometry>
    <material name="black"/>
  </visual>
</link>

(continues on next page)
<joint name="left_front_wheel_joint" type="fixed">
  <parent link="left_base"/>
  <child link="left_front_wheel"/>
  <origin rpy="0 0 0" xyz="0.133333333333 0 -0.085"/>
</joint>

<link name="left_back_wheel">
  <visual>
    <origin rpy="1.57075 0 0" xyz="0 0 0"/>
    <geometry>
      <cylinder length="0.1" radius="0.035"/>
    </geometry>
    <material name="black"/>
  </visual>
</link>

<joint name="left_back_wheel_joint" type="fixed">
  <parent link="left_base"/>
  <child link="left_back_wheel"/>
  <origin rpy="0 0 0" xyz="-0.133333333333 0 -0.085"/>
</joint>

<joint name="gripper_extension" type="fixed">
  <parent link="base_link"/>
  <child link="gripper_pole"/>
  <origin rpy="0 0 0" xyz="0.19 0 0.2"/>
</joint>

<link name="gripper_pole">
  <visual>
    <geometry>
      <cylinder length="0.2" radius="0.01"/>
    </geometry>
    <origin rpy="0 1.57075 0" xyz="0.1 0 0"/>
  </visual>
</link>

<joint name="left_gripper_joint" type="fixed">
  <origin rpy="0 0 0" xyz="0.2 0.01 0"/>
  <parent link="gripper_pole"/>
  <child link="left_gripper"/>
</joint>

<link name="left_gripper">
  <visual>
    <geometry>
      <cylinder length="0.2" radius="0.01"/>
    </geometry>
    <origin rpy="0 0 0" xyz="0.1 0 0"/>
  </visual>
</link>

<joint name="left_tip_joint" type="fixed">
  <origin rpy="0 0 0" xyz="0 0 0"/>
</joint>

(continues on next page)
(continued from previous page)

```xml
  <parent link="left_gripper"/>
  <child link="left_tip"/>
</joint>

<link name="left_tip">
  <visual>
    <origin rpy="0 0 0" xyz="0.09137 0.00495 0"/>
    <geometry>
      <mesh filename="package://urdf_tutorial/meshes/l_finger_tip.dae"/>
    </geometry>
  </visual>
</link>

<joint name="right_gripper_joint" type="fixed">
  <origin rpy="0 0 0" xyz="0.2 -0.01 0"/>
  <parent link="gripper_pole"/>
  <child link="right_gripper"/>
</joint>

<link name="right_gripper">
  <visual>
    <origin rpy="-3.1415 0 0" xyz="0 0 0"/>
    <geometry>
      <mesh filename="package://urdf_tutorial/meshes/l_finger.dae"/>
    </geometry>
  </visual>
</link>

<joint name="right_tip_joint" type="fixed">
  <child link="right_gripper"/>
</joint>

<link name="right_tip">
  <visual>
    <origin rpy="-3.1415 0 0" xyz="0.09137 0.00495 0"/>
    <geometry>
      <mesh filename="package://urdf_tutorial/meshes/l_finger_tip.dae"/>
    </geometry>
  </visual>
</link>

<link name="head">
  <visual>
    <geometry>
      <sphere radius="0.2"/>
    </geometry>
    <material name="white"/>
  </visual>
</link>

<joint name="head_swivel" type="fixed">
  <parent link="base_link"/>
  <child link="head"/>
```

(continues on next page)
<origin xyz="0 0 0.3"/>
</joint>

<link name="box">
  <visual>
    <geometry>
      <box size="0.08 0.08 0.08"/>
    </geometry>
    <material name="blue"/>
  </visual>
</link>

<joint name="tobox" type="fixed">
  <parent link="head"/>
  <child link="box"/>
  <origin xyz="0.1814 0 0.1414"/>
</joint>
</robot>

ros2 launch urdf_tutorial display.launch.py model:=urdf/05-visual.urdf

How to add the sphere should be fairly self explanatory:

<link name="head"/>

(continues on next page)
The meshes here were borrowed from the PR2. They are separate files which you have to specify the path for. You should use the `package://NAME_OF_PACKAGE/path` notation. The meshes for this tutorial are located within the `urdf_tutorial` package, in a folder called meshes.

- The meshes can be imported in a number of different formats. STL is fairly common, but the engine also supports DAE, which can have its own color data, meaning you don’t have to specify the color/material. Often these are in separate files. These meshes reference the `.tif` files also in the meshes folder.
- Meshes can also be sized using relative scaling parameters or a bounding box size.
- We could have also referred to meshes in a completely different package.

There you have it. A R2D2-like URDF model. Now you can continue on to the next step, making it move.

**Building a movable robot model**

**Goal:** Learn how to define movable joints in URDF.

**Tutorial level:** Intermediate

**Time:** 10 minutes

In this tutorial, we’re going to revise the R2D2 model we made in the previous tutorial so that it has movable joints. In the previous model, all of the joints were fixed. Now we’ll explore three other important types of joints: continuous, revolute and prismatic.
Make sure you have installed all prerequisites before continuing. See the previous tutorial for information on what is required.

Again, all of the robot models mentioned in this tutorial can be found in the urdf_tutorial package.

Here is the new urdf with flexible joints. You can compare it to the previous version to see everything that has changed, but we’re just going to focus on three example joints.

To visualize and control this model, run the same command as the last tutorial:

```bash
ros2 launch urdf_tutorial display.launch.py model:=urdf/06-flexible.urdf
```

However now this will also pop up a GUI that allows you to control the values of all the non-fixed joints. Play with the model some and see how it moves. Then, we can take a look at how we accomplished this.

The Head

```xml
<joint name="head_swivel" type="continuous">
  <parent link="base_link"/>
  <child link="head"/>
  <axis xyz="0 0 1"/>
  <origin xyz="0 0 0.3"/>
</joint>
```

The connection between the body and the head is a continuous joint, meaning that it can take on any angle from negative infinity to positive infinity. The wheels are also modeled like this, so that they can roll in both directions forever.

The only additional information we have to add is the axis of rotation, here specified by an xyz triplet, which specifies a vector around which the head will rotate. Since we want it to go around the z axis, we specify the vector “0 0 1”.

4.8. ROS 2 Documentation
The Gripper

```
<joint name="left_gripper_joint" type="revolute">
  <axis xyz="0 0 1"/>
  <limit effort="1000.0" lower="0.0" upper="0.548" velocity="0.5"/>
  <origin rpy="0 0 0" xyz="0.2 0.01 0"/>
  <parent link="gripper_pole"/>
  <child link="left_gripper"/>
</joint>
```

Both the right and the left gripper joints are modeled as revolute joints. This means that they rotate in the same way that the continuous joints do, but they have strict limits. Hence, we must include the limit tag specifying the upper and lower limits of the joint (in radians). We also must specify a maximum velocity and effort for this joint but the actual values don't matter for our purposes here.

The Gripper Arm

```
<joint name="gripper_extension" type="prismatic">
  <parent link="base_link"/>
  <child link="gripper_pole"/>
  <limit effort="1000.0" lower="-0.38" upper="0" velocity="0.5"/>
  <origin rpy="0 0 0" xyz="0.19 0 0.2"/>
</joint>
```

The gripper arm is a different kind of joint, namely a prismatic joint. This means that it moves along an axis, not around it. This translational movement is what allows our robot model to extend and retract its gripper arm.

The limits of the prismatic arm are specified in the same way as a revolute joint, except that the units are meters, not radians.

Other Types of Joints

There are two other kinds of joints that move around in space. Whereas the prismatic joint can only move along one dimension, a planar joint can move around in a plane, or two dimensions. Furthermore, a floating joint is unconstrained, and can move around in any of the three dimensions. These joints cannot be specified by just one number, and therefore aren't included in this tutorial.

Specifying the Pose

As you move the sliders around in the GUI, the model moves in Rviz. How is this done? First the GUI parses the URDF and finds all the non-fixed joints and their limits. Then, it uses the values of the sliders to publish sensor_msgs/msg/JointState messages. Those are then used by robot_state_publisher to calculate all of transforms between the different parts. The resulting transform tree is then used to display all of the shapes in Rviz.
Next steps

Now that you have a visibly functional model, you can add in some physical properties, or start using xacro to simplify your code.

Adding physical and collision properties

Goal: Learn how to add collision and inertial properties to links, and how to add joint dynamics to joints.

Tutorial level: Intermediate

Time: 10 minutes

Contents

- Collision
- Physical Properties
  - Inertia
  - Contact Coefficients
  - Joint Dynamics
- Other Tags
- Next Steps

In this tutorial, we’ll look at how to add some basic physical properties to your URDF model and how to specify its collision properties.

Collision

So far, we’ve only specified our links with a single sub-element, visual, which defines (not surprisingly) what the robot looks like. However, in order to get collision detection to work or to simulate the robot, we need to define a collision element as well. Here is the new urdf with collision and physical properties.

Here is the code for our new base link.

```xml
<link name="base_link">
  <visual>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
    <material name="blue">
      <color rgba="0 0 .8 1"/>
    </material>
  </visual>
  <collision>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
  </collision>
</link>
```
• The collision element is a direct subelement of the link object, at the same level as the visual tag.

• The collision element defines its shape the same way the visual element does, with a geometry tag. The format for the geometry tag is exactly the same here as with the visual.

• You can also specify an origin in the same way as a subelement of the collision tag (as with the visual).

In many cases, you’ll want the collision geometry and origin to be exactly the same as the visual geometry and origin. However, there are two main cases where you wouldn’t:

• **Quicker Processing.** Doing collision detection for two meshes is a lot more computational complex than for two simple geometries. Hence, you may want to replace the meshes with simpler geometries in the collision element.

• **Safe Zones.** You may want to restrict movement close to sensitive equipment. For instance, if we didn’t want anything to collide with R2D2’s head, we might define the collision geometry to be a cylinder encasing his head to prevent anything from getting too close to his head.

### Physical Properties

In order to get your model to simulate properly, you need to define several physical properties of your robot, i.e. the properties that a physics engine like Gazebo would need.

#### Inertia

Every link element being simulated needs an inertial tag. Here is a simple one.

```xml
<link name="base_link">
  <visual>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
    <material name="blue">
      <color rgba="0 0 .8 1"/>
    </material>
  </visual>
  <collision>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
  </collision>
  <inertial>
    <mass value="10"/>
    <inertia ixx="1e-3" ixy="0.0" ixz="0.0" iyy="1e-3" iyz="0.0" izz="1e-3"/>
  </inertial>
</link>
```

• This element is also a subelement of the link object.

• The mass is defined in kilograms.

• The 3x3 rotational inertia matrix is specified with the inertia element. Since this is symmetrical, it can be represented by only 6 elements, as such.

<table>
<thead>
<tr>
<th></th>
<th>ixx</th>
<th>ixy</th>
<th>ixz</th>
</tr>
</thead>
<tbody>
<tr>
<td>ixy</td>
<td></td>
<td>iyy</td>
<td>iyz</td>
</tr>
<tr>
<td>ixz</td>
<td>iyz</td>
<td></td>
<td>izz</td>
</tr>
</tbody>
</table>
• This information can be provided to you by modeling programs such as MeshLab. The inertia of geometric primitives (cylinder, box, sphere) can be computed using Wikipedia’s list of moment of inertia tensors (and is used in the above example).

• The inertia tensor depends on both the mass and the distribution of mass of the object. A good first approximation is to assume equal distribution of mass in the volume of the object and compute the inertia tensor based on the object’s shape, as outlined above.

• If unsure what to put, a matrix with $i_{xx}/i_{yy}/i_{zz}=1e^{-3}$ or smaller is often a reasonable default for a mid-sized link (it corresponds to a box of 0.1 m side length with a mass of 0.6 kg). The identity matrix is a particularly bad choice, since it is often much too high (it corresponds to a box of 0.1 m side length with a mass of 600 kg!).

• You can also specify an origin tag to specify the center of gravity and the inertial reference frame (relative to the link’s reference frame).

• When using realtime controllers, inertia elements of zero (or almost zero) can cause the robot model to collapse without warning, and all links will appear with their origins coinciding with the world origin.

Contact Coefficients

You can also define how the links behave when they are in contact with one another. This is done with a subelement of the collision tag called contact_coefficients. There are three attributes to specify:

• $mu$ - Friction coefficient

• $kp$ - Stiffness coefficient

• $kd$ - Dampening coefficient

Joint Dynamics

How the joint moves is defined by the dynamics tag for the joint. There are two attributes here:

• $friction$ - The physical static friction. For prismatic joints, the units are Newtons. For revolving joints, the units are Newton meters.

• $damping$ - The physical damping value. For prismatic joints, the units are Newton seconds per meter. For revolving joints, Newton meter seconds per radian.

If not specified, these coefficients default to zero.

Other Tags

In the realm of pure URDF (i.e. excluding Gazebo-specific tags), there are two remaining tags to help define the joints: calibration and safety controller. Check out the spec, as they are not included in this tutorial.
Next Steps

Reduce the amount of code and annoying math you have to do by using xacro.

Using Xacro to clean up your code

Goal: Learn some tricks to reduce the amount of code in a URDF file using Xacro

Tutorial level: Intermediate

Time: 20 minutes

Contents

- Using Xacro
- Constants
- Math
- Macros
  - Simple Macro
  - Parameterized Macro
- Practical Usage
  - Leg macro

By now, if you’re following all these steps at home with your own robot design, you might be sick of doing all sorts of math to get very simple robot descriptions to parse correctly. Fortunately, you can use the xacro package to make your life simpler. It does three things that are very helpful.

- Constants
- Simple Math
- Macros

In this tutorial, we take a look at all these shortcuts to help reduce the overall size of the URDF file and make it easier to read and maintain.

Using Xacro

As its name implies, xacro is a macro language for XML. The xacro program runs all of the macros and outputs the result. Typical usage looks something like this:

```bash
xacro model.xacro > model.urdf
```

You can also automatically generate the urdf in a launch file. This is convenient because it stays up to date and doesn’t use up hard drive space. However, it does take time to generate, so be aware that your launch file might take longer to start up.

```python
path_to_urdf = get_package_share_path('pr2_description') / 'robots' / 'pr2.urdf.xacro'
robot_state_publisher_node = launch_ros.actions.Node(
    package='robot_state_publisher',
    (continues on next page)
)
At the top of the URDF file, you must specify a namespace in order for the file to parse properly. For example, these are the first two lines of a valid xacro file:

```xml
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="firefighter">
```

**Constants**

Let’s take a quick look at our base_link in R2D2.

```xml
<link name="base_link">
  <visual>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
    <material name="blue"/>
  </visual>
  <collision>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
  </collision>
</link>
```

The information here is a little redundant. We specify the length and radius of the cylinder twice. Worse, if we want to change that, we need to do so in two different places.

Fortunately, xacro allows you to specify properties which act as constants. Instead, of the above code, we can write this.

```xml
<xacro:property name="width" value="0.2" />
<xacro:property name="bodylen" value="0.6" />
<link name="base_link">
  <visual>
    <geometry>
      <cylinder radius="${width}" length="${bodylen}"/>
    </geometry>
    <material name="blue"/>
  </visual>
  <collision>
    <geometry>
      <cylinder radius="${width}" length="${bodylen}"/>
    </geometry>
  </collision>
</link>
```
• The two values are specified in the first two lines. They can be defined just about anywhere (assuming valid XML), at any level, before or after they are used. Usually they go at the top.

• Instead of specifying the actual radius in the geometry element, we use a dollar sign and curly brackets to signify the value.

• This code will generate the same code shown above.

The value of the contents of the ${} construct are then used to replace the ${}. This means you can combine it with other text in the attribute.

```xml
<xacro:property name="robotname" value="marvin" />
<link name="${robotname}s_leg" />
```

This will generate

```xml
<link name="marvins_leg" />
```

However, the contents in the ${} don’t have to only be a property, which brings us to our next point...

## Math

You can build up arbitrarily complex expressions in the ${} construct using the four basic operations (+,-,\*,/), the unary minus, and parenthesis. Examples:

```xml
<cylinder radius="${wheeldiam/2}" length="0.1"/>
<origin xyz="${reflect*(width+.02)} 0 0.25"/>
```

You can also use more than the basic mathematical operations, like \texttt{sin} and \texttt{cos}.

## Macros

Here's the biggest and most useful component to the xacro package.

### Simple Macro

Let's take a look at a simple useless macro.

```xml
<xacro:macro name="default_origin">
  <origin xyz="0 0 0" rpy="0 0 0"/>
</xacro:macro>
<xacro:default_origin />
```

(This is useless, since if the origin is not specified, it has the same value as this.) This code will generate the following.

```xml
<origin rpy="0 0 0" xyz="0 0 0"/>
```

• The name is not technically a required element, but you need to specify it to be able to use it.

• Every instance of the <xacro:$NAME /> is replaced with the contents of the xacro:macro tag.
Parameterized Macro

You can also parameterize macros so that they don't generate the same exact text every time. When combined with the math functionality, this is even more powerful.

First, let's take an example of a simple macro used in R2D2.

```xml
<xacro:macro name="default_inertial" params="mass">
  <inertial>
    <mass value="${mass}" />
    <inertia
      ixx="1e-3" ixy="0.0" ixz="0.0"
      iyy="1e-3" iyz="0.0"
      izz="1e-3" />
  </inertial>
</xacro:macro>
```

This can be used with the code

```xml
<xacro:default_inertial mass="10"/>
```

The parameters act just like properties, and you can use them in expressions.

You can also use entire blocks as parameters too.

```xml
<xacro:macro name="blue_shape" params="name *shape">
  <link name="${name}"
    >
    <visual>
      <geometry>
        <xacro:insert_block name="shape" />
      </geometry>
      <material name="blue" />
    </visual>
    <collision>
      <geometry>
        <xacro:insert_block name="shape" />
      </geometry>
    </collision>
  </link>
</xacro:macro>

<xacro:blue_shape name="base_link">
  <cylinder radius=".42" length=".01" />
</xacro:blue_shape>
```

- To specify a block parameter, include an asterisk before its parameter name.
- A block can be inserted using the insert_block command.
- Insert the block as many times as you wish.
Practical Usage

The xacro language is rather flexible in what it allows you to do. Here are a few useful ways that xacro is used in the R2D2 model, in addition to the default inertial macro shown above.

To see the model generated by a xacro file, run the same command as with previous tutorials:

ros2 launch urdf_tutorial display.launch.py model:=urdf/08-macroed.urdf.xacro

(The launch file has been running the xacro command this whole time, but since there were no macros to expand, it didn't matter)

Leg macro

Often you want to create multiple similar looking objects in different locations. You can use a macro and some simple math to reduce the amount of code you have to write, like we do with R2's two legs.

```
<xacro:macro name="leg" params="prefix reflect">
  <link name="${prefix}_leg">
    <visual>
      <geometry>
        <box size="${leglen} 0.1 0.2"/>
      </geometry>
      <origin xyz="0 0 -${leglen/2}" rpy="0 ${pi/2} 0"/>
      <material name="white"/>
    </visual>
    <collision>
      <geometry>
        <box size="${leglen} 0.1 0.2"/>
      </geometry>
      <origin xyz="0 0 -${leglen/2}" rpy="0 ${pi/2} 0"/>
    </collision>
    <xacro:default_inertial mass="10"/>
  </link>
  <joint name="base_to_${prefix}_leg" type="fixed">
    <parent link="base_link"/>
    <child link="${prefix}_leg"/>
    <origin xyz="0 ${reflect*(width+.02)} 0.25"/>
  </joint>
</xacro:macro>

<xacro:leg prefix="right" reflect="1"/>
<xacro:leg prefix="left" reflect="-1"/>
```

- Common Trick 1: Use a name prefix to get two similarly named objects.
- Common Trick 2: Use math to calculate joint origins. In the case that you change the size of your robot, changing a property with some math to calculate the joint offset will save a lot of trouble.
- Common Trick 3: Using a reflect parameter, and setting it to 1 or -1. See how we use the reflect parameter to put the legs on either side of the body in the base_to_${prefix}_leg origin.
Using URDF with robot_state_publisher

**Goal:** Simulate a walking robot modeled in URDF and view it in Rviz.

**Tutorial level:** Intermediate

**Time:** 15 minutes

## Contents

- **Background**
- **Prerequisites**
- **Tasks**
  - 1 Create a package
  - 2 Create the URDF File
  - 3 Publish the state
  - 4 Create a launch file
  - 5 Edit the setup.py file
  - 6 Install the package
  - 7 View the results
- **Summary**

## Background

This tutorial will show you how to model a walking robot, publish the state as a tf2 message and view the simulation in Rviz. First, we create the URDF model describing the robot assembly. Next we write a node which simulates the motion and publishes the JointState and transforms. We then use robot_state_publisher to publish the entire robot state to /tf2.

## Prerequisites

- rviz2

As always, don’t forget to source ROS 2 in *every new terminal you open.*

## Tasks

### 1 Create a package

```bash
mkdir -p ~/second_ros2_ws/src
cd ~/second_ros2_ws/src
ros2 pkg create urdf_tutorial_r2d2 --build-type ament_python --dependencies rclpy --license Apache-2.0
cd urdf_tutorial_r2d2
```
You should now see a `urdf_tutorial_r2d2` folder. Next you will make several changes to it.

**2 Create the URDF File**

Create the directory where we will store some assets:

```bash
mkdir -p urdf
```

Download the URDF file and save it as `~/second_ros2_ws/src/urdf_tutorial_r2d2/urdf/r2d2.urdf.xml`. Download the Rviz configuration file and save it as `~/second_ros2_ws/src/urdf_tutorial_r2d2/urdf/r2d2.rviz`.

**3 Publish the state**

Now we need a method for specifying what state the robot is in. To do this, we must specify all three joints and the overall odometry.

Fire up your favorite editor and paste the following code into `~/second_ros2_ws/src/urdf_tutorial_r2d2/urdf_tutorial_r2d2/state_publisher.py`

```python
from math import sin, cos, pi
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile
from geometry_msgs.msg import Quaternion
from sensor_msgs.msg import JointState
from tf2_ros import TransformBroadcaster, TransformStamped
class StatePublisher(Node):

    def __init__(self):
        rclpy.init()
        super().__init__('state_publisher')
        qos_profile = QoSProfile(depth=10)
        self.joint_pub = self.create_publisher(JointState, 'joint_states', qos_profile)
        self.broadcaster = TransformBroadcaster(self, qos=qos_profile)
        self.nodeName = self.get_name()
        self.get_logger().info('{0} started'.format(self.nodeName))

        degree = pi / 180.0
        loop_rate = self.create_rate(30)

        # robot state
        tilt = 0.
        tinc = degree
        swivel = 0.
        angle = 0.
        height = 0.
        hinc = 0.005

        # message declarations
```

(continues on next page)
odom_trans = TransformStamped()
odom_trans.header.frame_id = 'odom'
odom_trans.child_frame_id = 'axis'
joint_state = JointState()
try:
    while rclpy.ok():
        rclpy.spin_once(self)

        # update joint_state
        now = self.get_clock().now()
        joint_state.header.stamp = now.to_msg()
        joint_state.name = ['swivel', 'tilt', 'periscope']
        joint_state.position = [swivel, tilt, height]

        # update transform
        # (moving in a circle with radius=2)
        odom_trans.header.stamp = now.to_msg()
        odom_trans.transform.translation.x = cos(angle)*2
        odom_trans.transform.translation.y = sin(angle)*2
        odom_trans.transform.translation.z = 0.7
        odom_trans.transform.rotation = euler_to_quaternion(0, 0, angle + pi/2)  # roll,pitch,yaw

        # send the joint state and transform
        self.joint_pub.publish(joint_state)
        self.broadcaster.sendTransform(odom_trans)

        # Create new robot state
        tilt += tinc
        if tilt < -0.5 or tilt > 0.0:
            tinc *= -1
        height += hinc
        if height > 0.2 or height < 0.0:
            hinc *= -1
        swivel += degree
        angle += degree/4

        # This will adjust as needed per iteration
        loop_rate.sleep()
except KeyboardInterrupt:
    pass
def euler_to_quaternion(roll, pitch, yaw):
    qx = sin(roll/2) * cos(pitch/2) * cos(yaw/2) - cos(roll/2) * sin(pitch/2) * sin(yaw/2)
    qy = cos(roll/2) * sin(pitch/2) * cos(yaw/2) + sin(roll/2) * cos(pitch/2) * sin(yaw/2)
    qz = cos(roll/2) * sin(pitch/2) * sin(yaw/2) - cos(roll/2) * cos(pitch/2) * cos(yaw/2)
    qw = cos(roll/2) * cos(pitch/2) * cos(yaw/2) + sin(roll/2) * sin(pitch/2) * sin(yaw/2)
return Quaternion(x=qx, y=qy, z=qz, w=qw)

def main():
    node = StatePublisher()
    if __name__ == '__main__':
        main()

4 Create a launch file

Create a new ~/second_ros2_ws/src/urdf_tutorial_r2d2/launch folder. Open your editor and paste the following code, saving it as ~/second_ros2_ws/src/urdf_tutorial_r2d2/launch/demo_launch.py

```python
import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch.substitutions import LaunchConfiguration
from launch_ros.actions import Node

def generate_launch_description():
    use_sim_time = LaunchConfiguration('use_sim_time', default='false')

    urdf_file_name = 'r2d2.urdf.xml'
    urdf = os.path.join(
        get_package_share_directory('urdf_tutorial_r2d2'),
        urdf_file_name)
    with open(urdf, 'r') as infp:
        robot_desc = infp.read()

    return LaunchDescription([
        DeclareLaunchArgument('use_sim_time',
            default_value='false',
            description='Use simulation (Gazebo) clock if true'),
        Node(
            package='robot_state_publisher',
            executable='robot_state_publisher',
            name='robot_state_publisher',
            output='screen',
            parameters=[
                {'use_sim_time': use_sim_time, 'robot_description': robot_desc}],
            arguments=[urdf]),
        Node(
            package='urdf_tutorial_r2d2',
            executable='state_publisher',
            name='state_publisher',
            output='screen'),
    ])
```
5 Edit the setup.py file

You must tell the colcon build tool how to install your Python package. Edit the `~/second_ros2_ws/src/urdf_tutorial_r2d2/setup.py` file as follows:

- include these import statements

```python
import os
from glob import glob
from setuptools import setup
from setuptools import find_packages
```

- append these 2 lines inside data_files

```python
data_files=[
    ...
    (os.path.join('share', package_name, 'launch'),
     glob(os.path.join('launch', '*.launch'))),
    (os.path.join('share', package_name),
     glob('urdf/*'))),
],
```

- modify the entry_points table so you can later run `state_publisher` from a console

```python
'console_scripts': [
    'state_publisher = urdf_tutorial_r2d2.state_publisher:main'
],
```

Save the setup.py file with your changes.

6 Install the package

```bash
cd ~/second_ros2_ws
colcon build --symlink-install --packages-select urdf_tutorial_r2d2
source install/setup.bash
```

7 View the results

Launch the package

```bash
ros2 launch urdf_tutorial_r2d2 demo_launch.py
```

Open a new terminal, the run Rviz using

```bash
rviz2 -d ~/second_ros2_ws/install/urdf_tutorial_r2d2/share/urdf_tutorial_r2d2/r2d2.rviz
```

See the User Guide for details on how to use Rviz.
Summary

You created a JointState publisher node and coupled it with robot_state_publisher to simulate a walking robot. The code used in these examples is originally from here.

Credit is given to the authors of this ROS 1 tutorial from which some content was reused.

Advanced

Enabling topic statistics (C++)

Goal: Enable ROS 2 Topic Statistics and view the output statistics data.

Tutorial level: Advanced

Time: 10 minutes

Table of Contents

• Background
• Prerequisites
• Tasks
  – 1 Write the subscriber node with statistics enabled
    * 1.1 Examine the code
    * 1.2 CMakeLists.txt
  – 2 Build and run
  – 3 Observe published statistic data
• Summary
• Related content

Background

This is a short tutorial on how to enable topic statistics in ROS 2 and view the published statistics output using command line tools (ros2topic).

ROS 2 provides the integrated measurement of statistics for messages received by any subscription, called Topic Statistics. With Topic Statistics enabled for your subscription, you can characterize the performance of your system or use the data to help diagnose any present issues.

For more details please see the Topic Statistics Concepts Page.
Prerequisites

An installation from either binaries or source.

In previous tutorials, you learned how to create a workspace, create a package, and create a C++ publisher and subscriber.

This tutorial assumes that you still have your cpp_pubsub package from the C++ tutorial.

Tasks

1 Write the subscriber node with statistics enabled

Navigate into the ros2_ws/src/cpp_pubsub/src folder, created in the previous tutorial, and download the example talker code by entering the following command:

Linux

```bash
wget -O member_function_with_topic_statistics.cpp https://raw.githubusercontent.com/ros2/examples/iron/rclcpp/topics/minimal_subscriber/member_function_with_topic_statistics.cpp
```

macOS

```bash
wget -O member_function_with_topic_statistics.cpp https://raw.githubusercontent.com/ros2/examples/iron/rclcpp/topics/minimal_subscriber/member_function_with_topic_statistics.cpp
```

Windows

Right click this link and select Save As publisher_member_function.cpp:

https://raw.githubusercontent.com/ros2/examples/iron/rclcpp/topics/minimal_subscriber/member_function_with_topic_statistics.cpp

Now there will be a new file named member_function_with_topic_statistics.cpp. Open the file using your preferred text editor.

```cpp
#include <chrono>
#include <memory>
#include "rclcpp/rclcpp.hpp"
#include "rclcpp/subscription_options.hpp"
#include "std_msgs/msg/string.hpp"

class MinimalSubscriberWithTopicStatistics : public rclcpp::Node {
public:

MinimalSubscriberWithTopicStatistics()
    : Node("minimal_subscriber_with_topic_statistics")
{
    // manually enable topic statistics via options
    auto options = rclcpp::SubscriptionOptions();
    options.topic_stats_options.state = rclcpp::TopicStatisticsState::Enable;
```
// configure the collection window and publish period (default 1s)
options.topic_stats_options.publish_period = std::chrono::seconds(10);

// configure the topic name (default '/statistics')
// options.topic_stats_options.publish_topic = "/topic_statistics"

auto callback = [this](std_msgs::msg::String::SharedPtr msg) {
    this->topic_callback(msg);
};

subscription_ = this->create_subscription<std_msgs::msg::String>(
    "topic", 10, callback, options);

private:
void topic_callback(const std_msgs::msg::String::ConstSharedPtr msg) const {
    RCLCPP_INFO(this->get_logger(), "I heard: '\%s'", msg->data.c_str());
} rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;

int main(int argc, char * argv[]) {
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<MinimalSubscriberWithTopicStatistics>());
    rclcpp::shutdown();
    return 0;
}

1.1 Examine the code

As in the C++ tutorial, we have a subscriber node which receives string messages from the topic topic from the topic_callback function. However, we’ve now added options to configure the subscription to enable topic statistics with the rclcpp::SubscriptionOptions() options struct.

// manually enable topic statistics via options
auto options = rclcpp::SubscriptionOptions();
options.topic_stats_options.state = rclcpp::TopicStatisticsState::Enable;

Optionally, fields such as the statistics collection/publish period and the topic used to publish statistics can be configured as well.

// configure the collection window and publish period (default 1s)
options.topic_stats_options.publish_period = std::chrono::seconds(10);

// configure the topic name (default '/statistics')
// options.topic_stats_options.publish_topic = "/my_topic"

The configurable fields are described in the following table:
<table>
<thead>
<tr>
<th>Subscription Config Field</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>topic_stats_options.state</td>
<td>Enable or disable topic statistics (default <code>rclcpp::TopicStatisticsState::DISABLE</code>)</td>
</tr>
<tr>
<td>topic_stats_options.publish_period</td>
<td>The period in which to collect statistics data and publish a statistics message (default 1s)</td>
</tr>
<tr>
<td>topic_stats_options.publish_topic</td>
<td>The topic to use when publishing statistics data (default /statistics)</td>
</tr>
</tbody>
</table>

### 1.2 CMakeLists.txt

Now open the `CMakeLists.txt` file.

Add the executable and name it `listener_with_topic_statistics` so you can run your node using `ros2 run`:

```cmake
add_executable(listener_with_topic_statistics src/member_function_with_topic_statistics.cpp)
ament_target_dependencies(listener_with_topic_statistics rclcpp std_msgs)
install(TARGETS talker listener listener_with_topic_statistics DESTINATION lib/${PROJECT_NAME})
```

Make sure to save the file, and then your pub/sub system, with topic statistics enabled, should be ready for use.

### 2 Build and run

To build, see the *Build and run* section in the pub/sub tutorial.

Run the subscriber with statistics enabled node:

```
ros2 run cpp_pubsub listener_with_topic_statistics
```

Now run the talker node:

```
ros2 run cpp_pubsub talker
```

The terminal should start publishing info messages every 0.5 seconds, like so:

```
[INFO] [minimal_publisher]: Publishing: "Hello World: 0"
[INFO] [minimal_publisher]: Publishing: "Hello World: 1"
[INFO] [minimal_publisher]: Publishing: "Hello World: 2"
[INFO] [minimal_publisher]: Publishing: "Hello World: 3"
[INFO] [minimal_publisher]: Publishing: "Hello World: 4"
```

The listener will start printing messages to the console, starting at whatever message count the publisher is on at that time, like so:

```
[INFO] [minimal_subscriber_with_topic_statistics]: I heard: "Hello World: 10"
[INFO] [minimal_subscriber_with_topic_statistics]: I heard: "Hello World: 11"
[INFO] [minimal_subscriber_with_topic_statistics]: I heard: "Hello World: 12"
[INFO] [minimal_subscriber_with_topic_statistics]: I heard: "Hello World: 13"
[INFO] [minimal_subscriber_with_topic_statistics]: I heard: "Hello World: 14"
```
Now that the subscriber node is receiving messages, it will periodically publish statistics messages. We will observe these messages in the next section.

3 Observe published statistic data

While the nodes are running, open a new terminal window. Execute the following command:

```
ros2 topic list
```

This will list all currently active topics. You should see the following:

```
/parameter_events
/rosout
/statistics
/topic
```

If you optionally changed the `topic_stats_options.publish_topic` field earlier in the tutorial, then you will see that name instead of `/statistics`.

The subscriber node you created is publishing statistics, for the topic `topic`, to the output topic `/statistics`.

We can visualize this using `RQt`

Now we can view the statistics data published to this topic with the following command:

```
ros2 topic echo /statistics
```

The terminal should start publishing statistics messages every 10 seconds, because the `topic_stats_options.publish_period` subscription configuration was optionally changed earlier in the tutorial.

```---
measurement_source_name: minimal_subscriber_with_topic_statistics
metrics_source: message_age
unit: ms
window_start:
  sec: 1594856666
  nanosec: 931527366
window_stop:
  sec: 1594856676
  nanosec: 930797670
statistics:
  - data_type: 1
data: .nan
  - data_type: 2
    data: .nan
  - data_type: 3
    data: .nan
```
From the message definition the `data_types` are as follows

<table>
<thead>
<tr>
<th>data_type value</th>
<th>statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>average</td>
</tr>
<tr>
<td>2</td>
<td>minimum</td>
</tr>
<tr>
<td>3</td>
<td>maximum</td>
</tr>
<tr>
<td>4</td>
<td>standard deviation</td>
</tr>
<tr>
<td>5</td>
<td>sample count</td>
</tr>
</tbody>
</table>

Here we see the two currently possible calculated statistics for the `std_msgs::msg::String` message published to `/topic` by the `minimal_publisher`. Because the `std_msgs::msg::String` does not have a message header, the `message_age` calculation cannot be performed, so NaNs are returned. However, the `message_period` can be calculated and we see the statistics populated in the message above.
Summary

You created a subscriber node with topic statistics enabled, which published statistics data from the C++’s publisher node. You were able to compile and run this node. While running, you were able to observe the statistics data.

Related content

To observe how the message_age period is calculated please see the ROS 2 Topic Statistics demo.

Using Fast DDS Discovery Server as discovery protocol [community-contributed]

Goal: This tutorial will show how to launch ROS 2 Nodes using the Fast DDS Discovery Server discovery protocol.

Tutorial level: Advanced

Time: 20 minutes

Table of Contents

- Background
- Fast DDS Discovery Server v2
- Prerequisites
- Run this tutorial
  - Setup Discovery Server
  - Launch listener node
  - Launch talker node
  - Demonstrate Discovery Server execution
  - Visualization tool rqt_graph
- Advance use cases
  - Server Redundancy
  - Backup Server
  - Discovery partitions
- ROS 2 Introspection
  - Daemon’s related tools
  - No Daemon tools
- Compare Fast DDS Discovery Server with Simple Discovery Protocol
Background

Starting from ROS 2 Eloquent Elusor, the Fast DDS Discovery Server protocol is a feature that offers a centralized dynamic discovery mechanism, as opposed to the distributed mechanism used in DDS by default. This tutorial explains how to run some ROS 2 examples using the Fast DDS Discovery Server feature as discovery communication.

In order to get more information about the available discovery configuration, please check the following documentation or read the Fast DDS Discovery Server specific documentation.

The Simple Discovery Protocol is the standard protocol defined in the DDS standard. However, it has known disadvantages in some scenarios.

- It does not Scale efficiently, as the number of exchanged packets increases significantly as new nodes are added.
- It requires multicasting capabilities that may not work reliably in some scenarios, e.g. WiFi.

The Fast DDS Discovery Server provides a Client-Server Architecture that allows nodes to connect with each other using an intermediate server. Each node functions as a discovery client, sharing its info with one or more discovery servers and receiving discovery information from it. This reduces discovery-related network traffic and it does not require multicasting capabilities.

These discovery servers can be independent, duplicated or connected with each other in order to create redundancy over the network and avoid having a single point of failure.

Fast DDS Discovery Server v2

The latest ROS 2 Foxy Fitzroy release (December 2020) included a new version, version 2 of the Fast DDS Discovery Server. This version includes a new filter feature that further reduces the number of discovery messages sent. This version uses the topic of the different nodes to decide if two nodes wish to communicate, or if they can be left unmatched (i.e. not discovering each other). The following figure shows the decrease in discovery messages:

This architecture reduces the number of messages sent between the server and clients dramatically. In the following graph, the reduction in network traffic over the discovery phase for the RMF Clinic demonstration is shown:

In order to use this functionality, the discovery server can be configured using the XML configuration for Participants. It is also possible to configure the discovery server using the fastdds tool and an environment variable, which is the approach used in this tutorial. For a more detailed explanation about the configuration of the discovery server, visit the Fast DDS Discovery Server documentation.

Prerequisites

This tutorial assumes you have a ROS 2 Foxy (or newer) installation. If your installation is using a ROS 2 version lower than Foxy, you cannot use the fastdds tool. Thus, in order to use the Discovery Server, you can update your repository to use a different Fast DDS version, or configure the discovery server using the Fast DDS XML QoS configuration.
Run this tutorial

The talker-listener ROS 2 demo creates a talker node that publishes a “hello world” message every second, and a listener node that listens to these messages.

By sourcing ROS 2 you will get access to the CLI tool fastdds. This tool gives access to the discovery tool, which can be used to launch a discovery server. This server will manage the discovery process for the nodes that connect to it.

Important: Do not forget to source ROS 2 in every new terminal opened.

Setup Discovery Server

Start by launching a discovery server with id 0, port 11811 (default port) and listening on all available interfaces.

Open a new terminal and run:

```
fastdds discovery --server-id 0
```

Launch listener node

Execute the listener demo, to listen to the /chatter topic.

In a new terminal, set the environment variable ROS_DISCOVERY_SERVER to the location of the discovery server. (Do not forget to source ROS 2 in every new terminal)

```
export ROS_DISCOVERY_SERVER=127.0.0.1:11811
```

Launch the listener node. Use the argument --remap __node:=listener_discovery_server to change the node’s name for this tutorial.

```
ros2 run demo_nodes_cpp listener --ros-args --remap __node:=listener_discovery_server
```

This will create a ROS 2 node, that will automatically create a client for the discovery server and connect to the server created previously to perform discovery, rather than using multicast.

Launch talker node

Open a new terminal and set the ROS_DISCOVERY_SERVER environment variable as before so that the node starts a discovery client.

```
export ROS_DISCOVERY_SERVER=127.0.0.1:11811
ros2 run demo_nodes_cpp talker --ros-args --remap __node:=talker_discovery_server
```

You should now see the talker publishing “hello world” messages, and the listener receiving these messages.
Demonstrate Discovery Server execution

So far, there is no evidence that this example and the standard talker-listener example are running differently. To clearly demonstrate this, run another node that is not connected to the discovery server. Run a new listener (listening in /chatter topic by default) in a new terminal and check that it is not connected to the talker already running.

```bash
ros2 run demo_nodes_cpp listener --ros-args --remap __node:=simple_listener
```

The new listener node should not be receiving the “hello world” messages.

To finally verify that everything is running correctly, a new talker can be created using the simple discovery protocol (the default DDS distributed discovery mechanism) for discovery.

```bash
ros2 run demo_nodes_cpp talker --ros-args --remap __node:=simple_talker
```

Now you should see the `simple_listener` node receiving the “hello world” messages from `simple_talker` but not the other messages from `talker_discovery_server`.

Visualization tool rqt_graph

The `rqt_graph` tool can be used to verify the nodes and structure of this example. Remember, in order to use `rqt_graph` with the discovery server protocol (i.e., to see the `listener_discovery_server` and `talker_discovery_server` nodes) the `ROS_DISCOVERY_SERVER` environment variable must be set before launching it.

Advance use cases

The following sections show different features of the discovery server that allow you to build a robust discovery server over the network.

Server Redundancy

By using `fastdds` tool, multiple discovery servers can be created. Discovery clients (ROS nodes) can connect to as many servers as desired. This allows to have a redundant network that will work even if some servers or nodes shut down unexpectedly. The figure below shows a simple architecture that provides server redundancy.

In several terminals, run the following code to establish a communication with redundant servers.

```bash
fastdds discovery --server-id 0 --ip-address 127.0.0.1 --port 11811
fastdds discovery --server-id 1 --ip-address 127.0.0.1 --port 11888
--server-id N means server with id N. When referencing the servers with `ROS_DISCOVERY_SERVER`, server 0 must be in first place and server 1 in second place.

export ROS_DISCOVERY_SERVER="127.0.0.1:11811;127.0.0.1:11888"
ros2 run demo_nodes_cpp talker --ros-args --remap __node:=talker

export ROS_DISCOVERY_SERVER="127.0.0.1:11811;127.0.0.1:11888"
ros2 run demo_nodes_cpp listener --ros-args --remap __node:=listener
```
Now, if one of these servers fails, there will still be discovery capability available and nodes will still discover each other.

**Backup Server**

The Fast DDS Discovery Server allows creating a server with backup functionality. This allows the server to restore the last state it saved in case of a shutdown.

In different terminals, run the following code to establish a communication with a backed-up server.

```bash
fastdds discovery --server-id 0 --ip-address 127.0.0.1 --port 11811 --backup
export ROS_DISCOVERY_SERVER="127.0.0.1:11811"
ros2 run demo_nodes_cpp talker --ros-args --remap __node:=talker

export ROS_DISCOVERY_SERVER="127.0.0.1:11811"
ros2 run demo_nodes_cpp listener --ros-args --remap __node:=listener
```

Several backup files are created in the discovery server's working directory (the directory it was launched in). The two SQLite files and two json files contain the information required to start a new server and restore the failed server's state in case of failure, avoiding the need for the discovery process to happen again, and without losing information.

**Discovery partitions**

Communication with discovery servers can be split to create virtual partitions in the discovery information. This means that two endpoints will only know about each other if there is a shared discovery server or a network of discovery servers between them. We are going to execute an example with two independent servers. The following figure shows the architecture.

With this schema Listener 1 will be connected to Talker 1 and Talker 2, as they share Server 1. Listener 2 will connect with Talker 1 as they share Server 2. But Listener 2 will not hear the messages from Talker 2 because they do not share any discovery server or discovery servers, including indirectly via connections between redundant discovery servers.

Run the first server listening on localhost with the default port of 11811.

```bash
fastdds discovery --server-id 0 --ip-address 127.0.0.1 --port 11811
```

In another terminal run the second server listening on localhost using another port, in this case port 11888.

```bash
fastdds discovery --server-id 1 --ip-address 127.0.0.1 --port 11888
```

Now, run each node in a different terminal. Use `ROS_DISCOVERY_SERVER` environment variable to decide which server they are connected to. Be aware that the `ids must match`.

```bash
export ROS_DISCOVERY_SERVER="127.0.0.1:11811;127.0.0.1:11888"
ros2 run demo_nodes_cpp talker --ros-args --remap __node:=talker

export ROS_DISCOVERY_SERVER="127.0.0.1:11811;127.0.0.1:11888"
ros2 run demo_nodes_cpp listener --ros-args --remap __node:=listener
```
We should see how Listener 1 is receiving messages from both talker nodes, while Listener 2 is in a different partition from Talker 2 and so does not receive messages from it.

**Note:** Once two endpoints (ROS nodes) have discovered each other, they do not need the discovery server network between them to listen to each other’s messages.

### ROS 2 Introspection

The [ROS 2 Command Line Interface](https://github.com/ros2/rosserial) supports several introspection tools to analyze the behavior of a ROS 2 network. These tools (i.e. `ros2 bag record`, `ros2 topic list`, etc.) are very helpful to understand a ROS 2 working network.

Most of these tools use DDS simple discovery to exchange topic information with every existing participant (using simple discovery, every participant in the network is connected with each other). However, the new Discovery Server v2 implements a network traffic reduction scheme that limits the discovery data between participants that do not share a topic. This means that nodes will only receive topic’s discovery data if it has a writer or a reader for that topic. As most ROS 2 CLIs need a node in the network (some of them rely on a running ROS 2 daemon, and some create their own nodes), using the Discovery Server v2 these nodes will not have all the network information, and thus their functionality will be limited.

The Discovery Server v2 functionality allows every Participant to run as a **Super Client**, a kind of **Client** that connects to a **Server**, from which it receives all the available discovery information (instead of just what it needs). In this sense, ROS 2 introspection tools can be configured as **Super Client**, thus being able to discover every entity that is using the Discovery Server protocol within the network.

**Note:** In this section we use the term **Participant** as a DDS entity. Each DDS **Participant** corresponds with a ROS 2 **Context**, a ROS 2 abstraction over DDS. **Nodes** `<ROS2Nodes>` are ROS 2 entities that rely on DDS communication interfaces: **DataWriter** and **DataReader**. Each **Participant** can hold multiple ROS 2 **Nodes**. For further details about these concepts, please visit the [Node to Participant mapping design document](https://github.com/ros2/rosserial)

### Daemon’s related tools

The ROS 2 Daemon is used in several ROS 2 CLI introspection tools. It creates its own Participant to add a ROS 2 **Node** to the network graph, in order to receive all the data sent. In order for the ROS 2 CLI to work when using Discovery Server mechanism, the ROS 2 Daemon needs to be configured as **Super Client**. Therefore, this section is devoted to explain how to use ROS 2 CLI with ROS 2 Daemon running as a **Super Client**. This will allow the Daemon to discover the entire Node graph, and to receive all topic and endpoint information. To do so, a Fast DDS XML configuration file is used to configure the ROS 2 Daemon and CLI tools.

Below you can find a XML configuration profile, which for this tutorial should be saved in the working directory as `super_client_configuration_file.xml` file. This file will configure every new participant using it, as a **Super Client**.
Under the `RemoteServer` tag, the `prefix` attribute value should be updated according to the server ID passed on the CLI (see Fast DDS CLI). The value specified in the shown XML snippet corresponds to an ID of value 0.

First of all, instantiate a Discovery Server using Fast DDS CLI specifying an ID of value 0.

```
fastdds discovery -i 0 -l 127.0.0.1 -p 11811
```

Run a talker and a listener that will discover each other through the Server (notice that `ROS_DISCOVERY_SERVER` configuration is the same as the one in `super_client_configuration_file.xml`).

```
export ROS_DISCOVERY_SERVER="127.0.0.1:11811"
ros2 run demo_nodes_cpp listener --ros-args --remap __node:=listener
```

```
export ROS_DISCOVERY_SERVER="127.0.0.1:11811"
ros2 run demo_nodes_cpp talker --ros-args --remap __node:=talker
```

Then, instantiate a ROS 2 Daemon using the Super Client configuration (remember to source ROS 2 installation in every new terminal).

```
export FASTRTPS_DEFAULT_PROFILES_FILE=super_client_configuration_file.xml
ros2 daemon stop
ros2 daemon start
ros2 topic list
ros2 node info /talker
```

(continues on next page)
We can also see the Node’s Graph using the ROS 2 tool `rqt_graph` as follows (you may need to press the refresh button):

```
export FASTRTPS_DEFAULT_PROFILES_FILE=super_client_configuration_file.xml
rqt_graph
```

### No Daemon tools

Some ROS 2 CLI tools do not use the ROS 2 Daemon. In order for these tools to connect with a Discovery Server and receive all the topics information they need to be instantiated as a **Super Client** that connects to the **Server**.

Following the previous configuration, build a simple system with a talker and a listener. First, run a **Server**:

```
fastdds discovery -i 0 -l 127.0.0.1 -p 11811
```

Then, run the talker and listener in separate terminals:

```
export ROS_DISCOVERY_SERVER="127.0.0.1:11811"
ros2 run demo_nodes_cpp listener --ros-args --remap __node:=listener
```

```
export ROS_DISCOVERY_SERVER="127.0.0.1:11811"
ros2 run demo_nodes_cpp talker --ros-args --remap __node:=talker
```

Continue using the ROS 2 CLI with `--no-daemon` option with the new configuration. New nodes will connect with the existing Server and will know every topic. Exporting `ROS_DISCOVERY_SERVER` is not needed as the ROS 2 tools will be configured through the `FASTRTPS_DEFAULT_PROFILES_FILE`.

```
export FASTRTPS_DEFAULT_PROFILES_FILE=super_client_configuration_file.xml
ros2 topic list --no-daemon
ros2 node info /talker --no-daemon --spin-time 2
```

### Compare Fast DDS Discovery Server with Simple Discovery Protocol

In order to compare executing nodes using the *Simple Discovery* Protocol (the default DDS mechanism for distributed discovery) or the *Discovery Server*, two scripts that execute a talker and many listeners and analyze the network traffic during this time are provided. For this experiment, `tshark` is required to be installed on your system. The configuration file is mandatory in order to avoid using intraprocess mode.

**Note:** These scripts require a discovery server closure feature that is only available from versions newer than the version provided in ROS 2 Foxy. In order to use this functionality, compile ROS 2 with Fast DDS v2.1.0 or higher.

These scripts’ features are references for advanced purposes and their study is left to the user.

- bash network traffic generator
- python3 graph generator
- XML configuration
Run the bash script with the path to `setup.bash` file to source ROS 2 as an argument. This will generate the traffic trace for simple discovery. Execute the same script with second argument `SERVER`. It will generate the trace for using the discovery server.

**Note:** Depending on your configuration of `tcpdump`, this script may require `sudo` privileges to read traffic across your network device.

After both executions are done, run the Python script to generate a graph similar to the one below.

```
$ export FASTRTPS_DEFAULT_PROFILES_FILE="no_intraprocess_configuration.xml"
$ sudo bash generate_discovery_packages.bash ~/ros2_foxy/install/local_setup.bash
$ sudo bash generate_discovery_packages.bash ~/ros2_foxy/install/local_setup.bash SERVER
$ python3 discovery_packets.py
```

This graph is the result of a specific run of the experiment. The reader can execute the scripts and generate their own results for comparison. It can easily be seen that network traffic is reduced when using discovery service.

The reduction in traffic is a result of avoiding every node announcing itself and waiting a response from every other node on the network. This creates a huge amount of traffic in large architectures. The reduction from this method increases with the number of nodes, making this architecture more scalable than the Simple Discovery Protocol approach.

The new Fast DDS Discovery Server v2 is available since Fast DDS v2.0.2, replacing the old discovery server. In this new version, those nodes that do not share topics will automatically not discover each other, saving the whole discovery data required to connect them and their endpoints. The experiment above does not show this case, but even so the massive reduction in traffic can be appreciated due to the hidden infrastructure topics of ROS 2 nodes.

**Implementing a custom memory allocator**

**Goal:** This tutorial will show how to use a custom memory allocator when writing ROS 2 C++ code.

**Tutorial level:** Advanced

**Time:** 20 minutes

---

This tutorial will teach you how to integrate a custom allocator for publishers and subscribers so that the default heap allocator is never called while your ROS nodes are executing. The code for this tutorial is available [here](#).
Background

Suppose you want to write real-time safe code, and you've heard about the many dangers of calling “new” during the real-time critical section, because the default heap allocator on most platforms is nondeterministic.

By default, many C++ standard library structures will implicitly allocate memory as they grow, such as `std::vector`. However, these data structures also accept an “Allocator” template argument. If you specify a custom allocator to one of these data structures, it will use that allocator for you instead of the system allocator to grow or shrink the data structure. Your custom allocator could have a pool of memory preallocated on the stack, which might be better suited to real-time applications.

In the ROS 2 C++ client library (rclcpp), we are following a similar philosophy to the C++ standard library. Publishers, subscribers, and the Executor accept an Allocator template parameter that controls allocations made by that entity during execution.

Writing an allocator

To write an allocator compatible with ROS 2’s allocator interface, your allocator must be compatible with the C++ standard library allocator interface.

The C++11 library provides something called `allocator_traits`. The C++11 standard specifies that a custom allocator only needs to fulfill a minimal set of requirements to be used to allocate and deallocate memory in a standard way. `allocator_traits` is a generic structure that fills out other qualities of an allocator based on an allocator written with the minimal requirements.

For example, the following declaration for a custom allocator would satisfy `allocator_traits` (of course, you would still need to implement the declared functions in this struct):

```cpp
template <class T>
struct custom_allocator {
    using value_type = T;
    custom_allocator() noexcept;
    template <class U> custom_allocator (const custom_allocator<U>&) noexcept;
    T* allocate (std::size_t n);
    void deallocate (T* p, std::size_t n);
};
```

You could then access other functions and members of the allocator filled in by `allocator_traits` like so: `std::allocator_traits<custom_allocator<T>>::construct(...)`

To learn about the full capabilities of `allocator_traits`, see https://en.cppreference.com/w/cpp/memory/allocator_traits.

However, some compilers that only have partial C++11 support, such as GCC 4.8, still require allocators to implement a lot of boilerplate code to work with standard library structures such as vectors and strings, because these structures do not use `allocator_traits` internally. Therefore, if you’re using a compiler with partial C++11 support, your allocator will need to look more like this:
template<typename T>
struct pointer_traits {
    using reference = T &;
    using const_reference = const T &;
};

// Avoid declaring a reference to void with an empty specialization
template<>
struct pointer_traits<void> {
};

template<typename T = void>
struct MyAllocator : public pointer_traits<T> {
public:
    using value_type = T;
    using size_type = std::size_t;
    using pointer = T *;
    using const_pointer = const T *;
    using difference_type = typename std::pointer_traits<pointer>::difference_type;

    MyAllocator() noexcept;

    ~MyAllocator() noexcept;

    template<typename U>
    MyAllocator(const MyAllocator<U> & noexcept;

    T * allocate(size_t size, const void * = 0);

    void deallocate(T * ptr, size_t size);

    template<typename U>
    struct rebind {
        typedef MyAllocator<U> other;
    };
};

template<typename T, typename U>
constexpr bool operator==(const MyAllocator<T> &,
    const MyAllocator<U> & noexcept;

    template<typename T, typename U>
    constexpr bool operator!=(const MyAllocator<T> &,
        const MyAllocator<U> & noexcept;
Writing an example main

Once you have written a valid C++ allocator, you must pass it as a shared pointer to your publisher, subscriber, and executor.

```cpp
auto alloc = std::make_shared<MyAllocator<void>>();
rclcpp::PublisherOptionsWithAllocator<MyAllocator<void>> publisher_options;
publisher_options.allocator = alloc;
auto publisher = node->create_publisher<std_msgs::msg::UInt32>(
  "allocator_tutorial", 10, publisher_options);

rclcpp::SubscriptionOptionsWithAllocator<MyAllocator<void>> subscription_options;
subscription_options.allocator = alloc;
auto msg_mem_strat = std::make_shared<rclcpp::message_memory_strategy::MessageMemoryStrategy<
  std_msgs::msg::UInt32, MyAllocator<void>>>(alloc);
auto subscriber = node->create_subscription<std_msgs::msg::UInt32>(
  "allocator_tutorial", 10, callback, subscription_options, msg_mem_strat);

std::shared_ptr<rclcpp::memory_strategy::MemoryStrategy> memory_strategy =
  std::make_shared<AllocatorMemoryStrategy<MyAllocator<void>>>(alloc);
rclcpp::ExecutorOptions options;
options.memory_strategy = memory_strategy;
rclcpp::executors::SingleThreadedExecutor executor(options);
```

You will also need to use your allocator to allocate any messages that you pass along the execution codepath.

```cpp
auto alloc = std::make_shared<MyAllocator<void>>();
```

Once you’ve instantiated the node and added the executor to the node, it’s time to spin:

```cpp
uint32_t i = 0;
while (rclcpp::ok()) {
  msg->data = i;
  i++;
  publisher->publish(msg);
  rclcpp::sleep_for(std::chrono::milliseconds(1));
  executor.spin_some();
}
```

Passing an allocator to the intra-process pipeline

Even though we instantiated a publisher and subscriber in the same process, we aren’t using the intra-process pipeline yet.

The IntraProcessManager is a class that is usually hidden from the user, but in order to pass a custom allocator to it we need to expose it by getting it from the rclcpp Context. The IntraProcessManager makes use of several standard library structures, so without a custom allocator it will call the default new.

```cpp
auto context = rclcpp::contexts::get_global_default_context();
auto options = rclcpp::NodeOptions()
  .context(context)
```

(continues on next page)
.use_intra_process_comms(true);
auto node = rclcpp::Node::make_shared("allocator_example", options);

Make sure to instantiate publishers and subscribers AFTER constructing the node in this way.

**Testing and verifying the code**

How do you know that your custom allocator is actually getting called?

The obvious thing to do would be to count the calls made to your custom allocator’s `allocate` and `deallocate` functions and compare that to the calls to `new` and `delete`.

Adding counting to the custom allocator is easy:

```cpp
T * allocate(size_t size, const void * = 0) {
    // ...
    num_allocs++;
    // ...
}

void deallocate(T * ptr, size_t size) {
    // ...
    num_deallocs++;
    // ...
}
```

You can also override the global `new` and `delete` operators:

```cpp
void operator delete(void * ptr) noexcept {
    if (ptr != nullptr) {
        if (is_running) {
            global_runtime_deallocs++;
        }
        std::free(ptr);
        ptr = nullptr;
    }
}

void operator delete(void * ptr, size_t) noexcept {
    if (ptr != nullptr) {
        if (is_running) {
            global_runtime_deallocs++;
        }
        std::free(ptr);
        ptr = nullptr;
    }
}
```

where the variables we are incrementing are just global static integers, and `is_running` is a global static boolean that gets toggled right before the call to spin.

The example executable prints the value of the variables. To run the example executable, use:


```bash
ros2 run demo_nodes_cpp allocator_tutorial
```

or, to run the example with the intra-process pipeline on:

```bash
ros2 run demo_nodes_cpp allocator_tutorial intra
```

You should get numbers like:

```
Global new was called 15590 times during spin
Global delete was called 15590 times during spin
Allocator new was called 27284 times during spin
Allocator delete was called 27281 times during spin
```

We’ve caught about 2/3 of the allocations/deallocations that happen on the execution path, but where do the remaining 1/3 come from?

As a matter of fact, these allocations/deallocations originate in the underlying DDS implementation used in this example.

Proving this is out of the scope of this tutorial, but you can check out the test for the allocation path that gets run as part of the ROS 2 continuous integration testing, which backtraces through the code and figures out whether certain function calls originate in the rmw implementation or in a DDS implementation:

https://github.com/ros2/realtime_support/blob/iron/tlsf_cpp/test/test_tlsf.cpp#L41

Note that this test is not using the custom allocator we just created, but the TLSF allocator (see below).

## The TLSF allocator

ROS 2 offers support for the TLSF (Two Level Segregate Fit) allocator, which was designed to meet real-time requirements:

https://github.com/ros2/realtime_support/tree/iron/tlsf_cpp

For more information about TLSF, see [http://www.gii.upv.es/tlsf/](http://www.gii.upv.es/tlsf/)

Note that the TLSF allocator is licensed under a dual-GPL/LGPL license.

A full working example using the TLSF allocator is here: [https://github.com/ros2/realtime_support/blob/iron/tlsf_cpp/example/allocator_example.cpp](https://github.com/ros2/realtime_support/blob/iron/tlsf_cpp/example/allocator_example.cpp)

### Unlocking the potential of Fast DDS middleware [community-contributed]

**Goal:** This tutorial will show how to use the extended configuration capabilities of Fast DDS in ROS 2.

**Tutorial level:** Advanced

**Time:** 20 minutes

### Table of Contents

- Background
- Prerequisites
- Mixing synchronous and asynchronous publications in the same node
- Create the node with the publishers
- Create the XML file with the profile configuration
- Execute the publisher node
- Create a node with the subscribers
- Execute the subscriber node
- Analysis of the example
  - Using other FastDDS capabilities with XML
    - Limiting the number of matching subscribers
    - Using partitions within the topic
  - Configuring a service and a client
    - Create the nodes with the service and client
    - Create the XML profiles for the service and client
    - Execute the nodes

Background

The interface between the ROS 2 stack and Fast DDS is provided by the ROS 2 middleware implementation rmw_fastrtps. This implementation is available in all ROS 2 distributions, both from binaries and from sources.

ROS 2 RMW only allows for the configuration of certain middleware QoS (see ROS 2 QoS policies). However, rmw_fastrtps offers extended configuration capabilities to take full advantage of the features in Fast DDS. This tutorial will guide you through a series of examples explaining how to use XML files to unlock this extended configuration.

In order to get more information about using Fast DDS on ROS 2, please check the following documentation.

Prerequisites

This tutorial assumes that you know how to create a package. It also assumes you know how to write a simple publisher and subscriber and a simple service and client. Although the examples are implemented in C++, the same concepts apply to Python packages.

Mixing synchronous and asynchronous publications in the same node

In this first example, a node with two publishers, one of them with synchronous publication mode and the other one with asynchronous publication mode, will be created.

rmw_fastrtps uses synchronous publication mode by default.

With synchronous publication mode the data is sent directly within the context of the user thread. This entails that any blocking call occurring during the write operation would block the user thread, thus preventing the application from continuing its operation. However, this mode typically yields higher throughput rates at lower latencies, since there is no notification nor context switching between threads.

On the other hand, with asynchronous publication mode, each time the publisher invokes the write operation, the data is copied into a queue, a background thread (asynchronous thread) is notified about the addition to the queue, and control
of the thread is returned to the user before the data is actually sent. The background thread is in charge of consuming the queue and sending the data to every matched reader.

**Create the node with the publishers**

First, create a new package named `sync_async_node_example_cpp` on a new workspace:

**Linux**

```bash
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws/src
ros2 pkg create --build-type ament_cmake --dependencies rclcpp std_msgs -- sync_async_node_example_cpp
```

**macOS**

```bash
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws/src
ros2 pkg create --build-type ament_cmake --dependencies rclcpp std_msgs -- sync_async_node_example_cpp
```

**Windows**

```bash
md \ros2_ws\src
cd \ros2_ws\src
ros2 pkg create --build-type ament_cmake --dependencies rclcpp std_msgs -- sync_async_node_example_cpp
```

Then, add a file named `src/sync_async_writer.cpp` to the package, with the following content. Note that the synchronous publisher will be publishing on topic `sync_topic`, while the asynchronous one will be publishing on topic `async_topic`.

```cpp
#include <chrono>
#include <functional>
#include <memory>
#include <string>
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

using namespace std::chrono_literals;

class SyncAsyncPublisher : public rclcpp::Node
{
public:
    SyncAsyncPublisher()
    : Node("sync_async_publisher"), count_(0)
    {
        // Create the synchronous publisher on topic 'sync_topic'
        sync_publisher_ = this->create_publisher<std_msgs::msg::String>("sync_topic", 10);
        // Create the asynchronous publisher on topic 'async_topic'
    }
};
```

(continues on next page)
async_publisher_ = this->create_publisher<std_msgs::msg::String>("async_topic", 10);

// This timer will trigger the publication of new data every half a second
timer_ = this->create_wall_timer(500ms, std::bind(&SyncAsyncPublisher::timer_callback, this));
}

private:

/**
 * Actions to run every time the timer expires
 */

void timer_callback()
{
    // Create a new message to be sent
    auto sync_message = std_msgs::msg::String();
    sync_message.data = "SYNC: Hello, world! " + std::to_string(count_);

    // Log the message to the console to show progress
    RCLCPP_INFO(this->get_logger(), "Synchronously publishing: '%s'", sync_message.data.c_str());

    // Publish the message using the synchronous publisher
    sync_publisher_->publish(sync_message);

    // Create a new message to be sent
    auto async_message = std_msgs::msg::String();
    async_message.data = "ASYNC: Hello, world! " + std::to_string(count_);

    // Log the message to the console to show progress
    RCLCPP_INFO(this->get_logger(), "Asynchronously publishing: '%s'", async_message.data.c_str());

    // Publish the message using the asynchronous publisher
    async_publisher_->publish(async_message);

    // Prepare the count for the next message
    count_++;
}

// This timer will trigger the publication of new data every half a second
rclcpp::TimerBase::SharedPtr timer_;

// A publisher that publishes asynchronously
rclcpp::Publisher<std_msgs::msg::String>::SharedPtr async_publisher_;

// A publisher that publishes synchronously
rclcpp::Publisher<std_msgs::msg::String>::SharedPtr sync_publisher_;

// Number of messages sent so far
size_t count_;
```c
int main(int argc, char * argv[]) {
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<SyncAsyncPublisher>())
    rclcpp::shutdown();
    return 0;
}
```

Now open the CMakeLists.txt file and add a new executable and name it `SyncAsyncWriter` so you can run your node using `ros2 run`:

```
add_executable(SyncAsyncWriter src/sync_async_writer.cpp)
ament_target_dependencies(SyncAsyncWriter rclcpp std_msgs)
```

Finally, add the `install(TARGETS...)` section so `ros2 run` can find your executable:

```
install(TARGETS
    SyncAsyncWriter
    DESTINATION lib/${PROJECT_NAME})
```

You can clean up your `CMakeLists.txt` by removing some unnecessary sections and comments, so it looks like this:

```
cmake_minimum_required(VERSION 3.8)
project(sync_async_node_example_cpp)

# Default to C++14
if(NOT CMAKE_CXX_STANDARD)
    set(CMAKE_CXX_STANDARD 14)
endif()

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
    add_compile_options(-Wall -Wextra -Wpedantic)
endif()

find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)

add_executable(SyncAsyncWriter src/sync_async_writer.cpp)
ament_target_dependencies(SyncAsyncWriter rclcpp std_msgs)

install(TARGETS
    SyncAsyncWriter
    DESTINATION lib/${PROJECT_NAME})

ament_package()
```

If this node is built and run now, both publishers will behave the same, publishing asynchronously in both topics, because this is the default publication mode. The default publication mode configuration can be changed in runtime during the node launching, using an XML file.
Create the XML file with the profile configuration

Create a file with name `SyncAsync.xml` and the following content:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">

  <!-- default publisher profile -->
  <publisher profile_name="default_publisher" is_default_profile="true">
    <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
  </publisher>

  <!-- default subscriber profile -->
  <subscriber profile_name="default_subscriber" is_default_profile="true">
    <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
  </subscriber>

  <!-- publisher profile for topic sync_topic -->
  <publisher profile_name="/sync_topic">
    <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
    <qos>
      <publishMode>
        <kind>SYNCHRONOUS</kind>
      </publishMode>
    </qos>
  </publisher>

  <!-- publisher profile for topic async_topic -->
  <publisher profile_name="/async_topic">
    <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
    <qos>
      <publishMode>
        <kind>ASYNCHRONOUS</kind>
      </publishMode>
    </qos>
  </publisher>

</profiles>
```

Note that several profiles for publisher and subscriber are defined. Two default profiles which are defined setting the `is_default_profile` to `true`, and two profiles with names that coincide with those of the previously defined topics: `sync_topic` and another one for `async_topic`. These last two profiles set the publication mode to `SYNCHRONOUS` or `ASYNCHRONOUS` accordingly. Note also that all profiles specify a `historyMemoryPolicy` value, which is needed for the example to work, and the reason will be explained later on this tutorial.
### Execute the publisher node

You will need to export the following environment variables for the XML to be loaded:

**Linux**

```bash
export RMW_IMPLEMENTATION=rmw_fastrtps_cpp
export RMW_FASTRTPS_USE_QOS_FROM_XML=1
export FASTRTPS_DEFAULT_PROFILES_FILE=path/to/SyncAsync.xml
```

**macOS**

```bash
export RMW_IMPLEMENTATION=rmw_fastrtps_cpp
export RMW_FASTRTPS_USE_QOS_FROM_XML=1
export FASTRTPS_DEFAULT_PROFILES_FILE=path/to/SyncAsync.xml
```

**Windows**

```bash
SET RMW_IMPLEMENTATION=rmw_fastrtps_cpp
SET RMW_FASTRTPS_USE_QOS_FROM_XML=1
SET FASTRTPS_DEFAULT_PROFILES_FILE=path/to/SyncAsync.xml
```

Finally, ensure you have sourced your setup files and run the node:

```bash
source install/setup.bash
ros2 run sync_async_node_example_cpp SyncAsyncWriter
```

You should see the publishers sending the data from the publishing node, like so:

```
[INFO] [1612972049.994630332] [sync_async_publisher]: Synchronously publishing: 'SYNC: Hello, world! 0'
[INFO] [1612972049.995097767] [sync_async_publisher]: Asynchronously publishing: 'ASYNC: Hello, world! 0'
[INFO] [1612972050.494478706] [sync_async_publisher]: Synchronously publishing: 'SYNC: Hello, world! 1'
[INFO] [1612972050.494664334] [sync_async_publisher]: Asynchronously publishing: 'ASYNC: Hello, world! 1'
[INFO] [1612972050.994368474] [sync_async_publisher]: Synchronously publishing: 'SYNC: Hello, world! 2'
[INFO] [1612972050.994549851] [sync_async_publisher]: Asynchronously publishing: 'ASYNC: Hello, world! 2'
```

Now you have a synchronous publisher and an asynchronous publisher running inside the same node.

### Create a node with the subscribers

Next, a new node with the subscribers that will listen to the sync_topic and async_topic publications is going to be created. In a new source file named `src/sync_async_reader.cpp` write the following content:

```cpp
#include <functional>
#include <memory>
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"
```

(continues on next page)
using std::placeholders::_1;

class SyncAsyncSubscriber : public rclcpp::Node
{
  public:

    SyncAsyncSubscriber()
      : Node("sync_async_subscriber")
    {
      // Create the synchronous subscriber on topic 'sync_topic'
      // and tie it to the topic_callback
      sync_subscription_ = this->create_subscription<std_msgs::msg::String>(
          "sync_topic", 10, std::bind(&SyncAsyncSubscriber::topic_callback,
                                      this, _1));

      // Create the asynchronous subscriber on topic 'async_topic'
      // and tie it to the topic_callback
      async_subscription_ = this->create_subscription<std_msgs::msg::String>(
          "async_topic", 10, std::bind(&SyncAsyncSubscriber::topic_callback,
                                       this, _1));
    }

  private:

    /**
     * Actions to run every time a new message is received
     */
    void topic_callback(const std_msgs::msg::String & msg) const
    {
      RCLCPP_INFO(this->get_logger(), "I heard: '%s'", msg.data.c_str());
    }

    // A subscriber that listens to topic 'sync_topic'
    rclcpp::Subscription<std_msgs::msg::String>::SharedPtr sync_subscription_;

    // A subscriber that listens to topic 'async_topic'
    rclcpp::Subscription<std_msgs::msg::String>::SharedPtr async_subscription_;

  int main(int argc, char * argv[])
  {
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<SyncAsyncSubscriber>());
    rclcpp::shutdown();
    return 0;
  }

Open the CMakeLists.txt file and add a new executable and name it SyncAsyncReader under the previous SyncAsyncWriter:

add_executable(SyncAsyncReader src/sync_async_reader.cpp)
ament_target_dependencies(SyncAsyncReader rclcpp std_msgs)
install(TARGETS
    SyncAsyncReader
    DESTINATION lib/${PROJECT_NAME})

Execute the subscriber node

With the publisher node running in one terminal, open another one and export the required environment variables for the XML to be loaded:

Linux

    export RMW_IMPLEMENTATION=rmw_fastrtps_cpp
    export RMW_FASTRTPS_USE_QOS_FROM_XML=1
    export FASTRTPS_DEFAULT_PROFILES_FILE=path/to/SyncAsync.xml

macOS

    export RMW_IMPLEMENTATION=rmw_fastrtps_cpp
    export RMW_FASTRTPS_USE_QOS_FROM_XML=1
    export FASTRTPS_DEFAULT_PROFILES_FILE=path/to/SyncAsync.xml

Windows

    SET RMW_IMPLEMENTATION=rmw_fastrtps_cpp
    SET RMW_FASTRTPS_USE_QOS_FROM_XML=1
    SET FASTRTPS_DEFAULT_PROFILES_FILE=path/to/SyncAsync.xml

Finally, ensure you have sourced your setup files and run the node:

source install/setup.bash
ros2 run sync_async_node_example_cpp SyncAsyncReader

You should see the subscribers receiving the data from the publishing node, like so:

[INFO] [1612972054.495429090] [sync_async_subscriber]: I heard: 'SYNC: Hello, world! 10'
[INFO] [1612972054.995410057] [sync_async_subscriber]: I heard: 'ASYNC: Hello, world! 10'
[INFO] [1612972055.495453494] [sync_async_subscriber]: I heard: 'SYNC: Hello, world! 11'
[INFO] [1612972055.995396561] [sync_async_subscriber]: I heard: 'ASYNC: Hello, world! 11'
[INFO] [1612972056.495534818] [sync_async_subscriber]: I heard: 'SYNC: Hello, world! 12'
[INFO] [1612972056.995473953] [sync_async_subscriber]: I heard: 'ASYNC: Hello, world! 12'

Analysis of the example

Configuration profiles XML

The XML file defines several configurations for publishers and subscribers. You can have a default publisher configuration profile and several topic-specific publisher profiles. The only requirement is that all publisher profiles have a different name and that there is only a single default profile. The same goes for subscribers.

In order to define a configuration for a specific topic, just name the profile after the the ROS 2 topic name (like /sync_topic and /async_topic in the example), and rmw_fastrtps will apply this profile to all publishers and
subscribers for that topic. The default configuration profile is identified by the attribute `is_default_profile` set to `true`, and acts as a fallback profile when there is no other one with a name matching the topic name.

The environment variable `FASTRTPS_DEFAULT_PROFILES_FILE` is used to inform `Fast DDS` the path to the XML file with the configuration profiles to load.

**RMW_FASTRTPS_USE_QOS_FROM_XML**

Among all the configurable attributes, `rmw_fastrtps` treats `publishMode` and `historyMemoryPolicy` differently. By default, these values are set to `ASYNCHRONOUS` and `PREALLOCATED_WITH_REALLOC` within the `rmw_fastrtps` implementation, and the values set on the XML file are ignored. In order to use the values in the XML file, the environment variable `RMW_FASTRTPS_USE_QOS_FROM_XML` must be set to `1`.

However, this entails another caveat: If `RMW_FASTRTPS_USE_QOS_FROM_XML` is set, but the XML file does not define `publishMode` or `historyMemoryPolicy`, these attributes take the `Fast DDS` default value instead of the `rmw_fastrtps` default value. This is important, especially for `historyMemoryPolicy`, because the `Fast DDS` default value is `PREALLOCATED` which does not work with ROS2 topic data types. Therefore, in the example, a valid value for this policy has been explicitly set (`DYNAMIC`).

**Prioritization of rmw_qos_profile_t**

ROS 2 QoS contained in `rmw_qos_profile_t` are always honored, unless set to `*_SYSTEM_DEFAULT`. In that case, XML values (or `Fast DDS` default values in the absence of XML ones) are applied. This means that if any QoS in `rmw_qos_profile_t` is set to something other than `*_SYSTEM_DEFAULT`, the corresponding value in the XML is ignored.

**Using other FastDDS capabilities with XML**

Although we have created a node with two publishers with different configuration, it is not easy to check that they are behaving differently. Now that the basics of XML profiles have been covered, let us use them to configure something which has some visual effect on the nodes. Specifically, a maximum number of matching subscribers on one of the publishers and a partition definition on the other will be set. Note that these are only very simple examples among all the configuration attributes that can be tuned on `rmw_fastrtps` through XML files. Please refer to "Fast DDS" documentation to see the whole list of attributes that can be configured through XML files.

**Limiting the number of matching subscribers**

Add a maximum number of matched subscribers to the `/async_topic` publisher profile. It should look like this:

```xml
<!-- publisher profile for topic async_topic -->
<publisher profile_name="/async_topic">  
    <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>  
    <qos>  
        <publishMode>  
            <kind>ASYNCHRONOUS</kind>  
        </publishMode>  
    </qos>  
    <matchedSubscribersAllocation>  
        <initial>0</initial>  
        <maximum>1</maximum>  
    </matchedSubscribersAllocation>  
</publisher>  
</async_topic>  
(continues on next page)
```
The number of matching subscribers is being limited to one.

Now open three terminals and do not forget to source the setup files and to set the required environment variables. On the first terminal run the publisher node, and the subscriber node on the other two. You should see that only the first subscriber node receives the messages from both topics. The second one could not complete the matching process in the /async_topic because the publisher prevented it, as it had already reached its maximum of matched publishers. Consequently, only the messages from the /sync_topic are going to be received in this third terminal:

```
[INFO] [1613127657.088860890] [sync_async_subscriber]: I heard: 'SYNC: Hello, world! 18'
[INFO] [1613127657.588896594] [sync_async_subscriber]: I heard: 'SYNC: Hello, world! 19'
[INFO] [1613127658.088849401] [sync_async_subscriber]: I heard: 'SYNC: Hello, world! 20'
```

### Using partitions within the topic

The partitions feature can be used to control which publishers and subscribers exchange information within the same topic.

Partitions introduce a logical entity isolation level concept inside the physical isolation induced by a Domain ID. For a publisher to communicate with a subscriber, they have to belong at least to one common partition. Partitions represent another level to separate publishers and subscribers beyond domain and topic. Unlike domain and topic, an endpoint can belong to several partitions at the same time. For certain data to be shared over different domains or topics, there must be a different publisher for each, sharing its own history of changes. However, a single publisher can share the same data sample over different partitions using a single topic data change, thus reducing network overload.

Let us change the /sync_topic publisher to partition part1 and create a new /sync_topic subscriber which uses partition part2. Their profiles should now look like this:

```
<!-- publisher profile for topic sync_topic -->
<publisher profile_name="/sync_topic">
  <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
  <qos>
    <publishMode>
      <kind>SYNCHRONOUS</kind>
    </publishMode>
    <partition>
      <names>
        <name>part1</name>
      </names>
    </partition>
  </qos>
</publisher>

<!-- subscriber profile for topic sync_topic -->
<subscriber profile_name="/sync_topic">
  <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
  <qos>
    <partition>
      <names>
      </names>
    </partition>
  </qos>
</subscriber>
```
Open two terminals. Do not forget to source the setup files and to set the required environment variables. On the first terminal run the publisher node, and the subscriber node on the other one. You should see that only the /async_topic messages are reaching the subscriber. The /sync_topic subscriber is not receiving the data as it is in a different partition from the corresponding publisher.

```
[INFO] [1612972054.995410057] [sync_async_subscriber]: I heard: 'ASYNC: Hello, world! 10'
[INFO] [1612972055.995396561] [sync_async_subscriber]: I heard: 'ASYNC: Hello, world! 11'
[INFO] [1612972056.995473953] [sync_async_subscriber]: I heard: 'ASYNC: Hello, world! 12'
```

### Configuring a service and a client

Services and clients have a publisher and a subscriber each, that communicate through two different topics. For example, for a service named ping there is:

- A service subscriber listening to requests on /rq/ping.
- A service publisher sending responses on /rr/ping.
- A client publisher sending requests on /rq/ping.
- A client subscriber listening to responses on /rr/ping.

Although you can use these topic names to set the configuration profiles on the XML, sometimes you may wish to apply the same profile to all services or clients on a node. Instead of copying the same profile with all topic names generated for all services, you can just create a publisher and subscriber profile pair named service. The same can be done for clients creating a pair named client.

### Create the nodes with the service and client

Start creating the node with the service. Add a new source file named src/ping_service.cpp on your package with the following content:

```cpp
#include <memory>
#include "rclcpp/rclcpp.hpp"
#include "example_interfaces/srv/trigger.hpp"

/**
 * Service action: responds with success=true and prints the request on the console
 */
void ping(const std::shared_ptr<example_interfaces::srv::Trigger::Request> request,
          std::shared_ptr<example_interfaces::srv::Trigger::Response> response)
{
    // The request data is unused
    (void) request;
}
```

(continues on next page)
// Build the response
response->success = true;

// Log to the console
RCLCPP_INFO(rclcpp::get_logger("ping_server"), "Incoming request");
RCLCPP_INFO(rclcpp::get_logger("ping_server"), "Sending back response");
}

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);

    // Create the node and the service
    std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("ping_server");
    rclcpp::Service<example_interfaces::srv::Trigger>::SharedPtr service =
        node->create_service<example_interfaces::srv::Trigger>("ping", &ping);

    // Log that the service is ready
    RCLCPP_INFO(rclcpp::get_logger("ping_server"), "Ready to serve.");

    // run the node
    rclcpp::spin(node);
    rclcpp::shutdown();
}

Create the client in a file named src/ping_client.cpp with the following content:

```cpp
#include <chrono>
#include <memory>
#include "rclcpp/rclcpp.hpp"
#include "example_interfaces/srv/trigger.hpp"

using namespace std::chrono_literals;

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);

    // Create the node and the client
    std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("ping_client");
    rclcpp::Client<example_interfaces::srv::Trigger>::SharedPtr client =
        node->create_client<example_interfaces::srv::Trigger>("ping");

    // Create a request
    auto request = std::make_shared<example_interfaces::srv::Trigger::Request>();

    // Wait for the service to be available
    while (!client->wait_for_service(1s)) {
        if (!rclcpp::ok()) {
            RCLCPP_ERROR(rclcpp::get_logger("ping_client"), "Interrupted while waiting for the service. Exiting.");
        }
    }

    // Call the service
    client->asyncinvoke(rclcpp::Bind(&example_interfaces::srv::Trigger::Callback, request));
```
return 0;
}
RCLCPP_INFO(rclcpp::get_logger("ping_client"), "Service not available, waiting...
again...");
}

// Now that the service is available, send the request
RCLCPP_INFO(rclcpp::get_logger("ping_client"), "Sending request");
auto result = client->async_send_request(request);

// Wait for the result and log it to the console
if (rclcpp::spin_until_future_complete(node, result) ==
   rclcpp::FutureReturnCode::SUCCESS) {
   RCLCPP_INFO(rclcpp::get_logger("ping_client"), "Response received");
} else {
   RCLCPP_ERROR(rclcpp::get_logger("ping_client"), "Failed to call service ping");
}

rclcpp::shutdown();
return 0;
}

Open the CMakeLists.txt file and add two new executables ping_service and ping_client:

```cmake
find_package(example_interfaces REQUIRED)
add_executable(ping_service src/ping_service.cpp)
ament_target_dependencies(ping_service example_interfaces rclcpp)
add_executable(ping_client src/ping_client.cpp)
ament_target_dependencies(ping_client example_interfaces rclcpp)
install(TARGETS
   ping_service
   DESTINATION lib/${PROJECT_NAME})
install(TARGETS
   ping_client
   DESTINATION lib/${PROJECT_NAME})
```

Finally, build the package.
Create the XML profiles for the service and client

Create a file with name `ping.xml` with the following content:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <!-- default publisher profile -->
    <publisher profile_name="default_publisher" is_default_profile="true">
        <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
    </publisher>

    <!-- default subscriber profile -->
    <subscriber profile_name="default_subscriber" is_default_profile="true">
        <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
    </subscriber>

    <!-- service publisher is SYNC -->
    <publisher profile_name="service">
        <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
        <qos>
            <publishMode>
                <kind>SYNCHRONOUS</kind>
            </publishMode>
        </qos>
    </publisher>

    <!-- client publisher is ASYNC -->
    <publisher profile_name="client">
        <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
        <qos>
            <publishMode>
                <kind>ASYNCHRONOUS</kind>
            </publishMode>
        </qos>
    </publisher>
</profiles>
```

This configuration file sets the publication mode to SYNCHRONOUS on the service and to ASYNCHRONOUS on the client. Note that we are only defining the publisher profiles for both the service and the client, but subscriber profiles could be provided too.

**Execute the nodes**

Open two terminals and source the setup files on each one. Then set the required environment variables for the XML to be loaded:

**Linux**

```bash
export RMW_IMPLEMENTATION=rmw_fastrtps_cpp
export RMW_FASTRTPS_USE_QOS_FROM_XML=1
export FASTRTPS_DEFAULT_PROFILES_FILE=path/to/ping.xml
```
macOS

```bash
export RMW_IMPLEMENTATION=rmw_fastrtps_cpp
export RMW_FASTRTPS_USE_QOS_FROM_XML=1
export FASTRTPS_DEFAULT_PROFILES_FILE=path/to/ping.xml
```

Windows

```bash
SET RMW_IMPLEMENTATION=rmw_fastrtps_cpp
SET RMW_FASTRTPS_USE_QOS_FROM_XML=1
SET FASTRTPS_DEFAULT_PROFILES_FILE=path/to/ping.xml
```

On the first terminal run the service node.

```bash
ros2 run sync_async_node_example_cpp ping_service
```

You should see the service waiting for requests:

```
[INFO] [1612977403.805799037] [ping_server]: Ready to serve.
```

On the second terminal, run the client node.

```bash
ros2 run sync_async_node_example_cpp ping_client
```

You should see the client sending the request and receiving the response:

```
[INFO] [1612977404.805799037] [ping_client]: Sending request
[INFO] [1612977404.825473835] [ping_client]: Response received
```

At the same time, the output in the server console has been updated:

```
[INFO] [1612977403.805799037] [ping_server]: Ready to serve.
[INFO] [1612977404.807314904] [ping_server]: Incoming request
[INFO] [1612977404.836405125] [ping_server]: Sending back response
```

**Improved Dynamic Discovery**

**Goal:** This tutorial will show how to use the improved dynamic discovery configuration.

**Tutorial level:** Advanced

**Time:** 15 minutes
Overview

By default, ROS 2 will attempt to find all nodes on all hosts on the same subnet automatically. However, the following options are available to control the ROS 2 discovery range.

Configuration Parameters

- **ROS_AUTOMATIC_DISCOVERY_RANGE**: controls how far ROS nodes will try to discover each other.
  
  Valid options are:
  
  - **SUBNET** is the default, and for DDS based middleware it means it will discover any node reachable via multicast.
  
  - **LOCALHOST** means a node will only try to discover other nodes on the same machine.
  
  - **OFF** means the node won’t discover any other nodes, even on the same machine.
  
  - **SYSTEM_DEFAULT** means “don’t change any discovery settings”.

- **ROS_STATIC_PEERS**: is a semicolon (;) separated list of addresses that ROS should try to discover nodes on. This allows connecting to nodes on specific machines (as long as their discovery range is not set to OFF).

The combination of these two environment variables for local and remote nodes will enable and control the ROS 2 communication discovery range. The following tables highlight the discovery range behavior for possible combination.

A X indicates that nodes A and B will not discover each other and communicate. A 0 indicates that nodes A and B will discover each other and communicate.

<table>
<thead>
<tr>
<th>Same host</th>
<th>Node A setting</th>
<th>Node B setting</th>
<th>With static peer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No static peer</td>
<td>Off</td>
<td>Localhost</td>
</tr>
<tr>
<td>Node A setting</td>
<td>No static peer</td>
<td>Off</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Localhost</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Subnet</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>With static peer</td>
<td>Off</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Localhost</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Subnet</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>
Table 4: Node A and B running in the different hosts

<table>
<thead>
<tr>
<th>Different hosts</th>
<th>Node A setting</th>
<th>Node B setting</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No static peer</td>
<td>With static peer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Off</td>
<td>Off</td>
<td></td>
</tr>
<tr>
<td>Node A setting</td>
<td>Off</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Localhost</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Subnet</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>With static peer</td>
<td>Off</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Localhost</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Subnet</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Examples

For example, the following commands will limit the ROS 2 communication only with localhost and specific peers:

Linux

```
export ROS_AUTOMATIC_DISCOVERY_RANGE=LOCALHOST
export ROS_STATIC_PEERS=192.168.0.1;remote.com
```

To maintain this setting between shell sessions, you can add the command to your shell startup script:

```
echo "export ROS_AUTOMATIC_DISCOVERY_RANGE=LOCALHOST" >> ~/.bashrc
echo "export ROS_STATIC_PEERS=192.168.0.1;remote.com" >> ~/.bashrc
```

macOS

```
export ROS_AUTOMATIC_DISCOVERY_RANGE=LOCALHOST
export ROS_STATIC_PEERS=192.168.0.1;remote.com
```

To maintain this setting between shell sessions, you can add the command to your shell startup script:

```
echo "export ROS_AUTOMATIC_DISCOVERY_RANGE=LOCALHOST" >> ~/.bash_profile
echo "export ROS_STATIC_PEERS=192.168.0.1;remote.com" >> ~/.bash_profile
```

Windows

```
set ROS_AUTOMATIC_DISCOVERY_RANGE=LOCALHOST
set ROS_STATIC_PEERS=192.168.0.1;remote.com
```

If you want to make this permanent between shell sessions, also run:

```
setx ROS_AUTOMATIC_DISCOVERY_RANGE LOCALHOST
setx ROS_STATIC_PEERS 192.168.0.1;remote.com
```
Recording a bag from a node (C++)

Goal: Record data from your own C++ node to a bag.

Tutorial level: Advanced

Time: 20 minutes

Contents

- Background
- Prerequisites
- Tasks
  - 1 Create a package
  - 2 Write the C++ node
  - 3 Build and run
  - 4 Record synthetic data from a node
  - 5 Record synthetic data from an executable
- Summary

Background

rosbag2 doesn’t just provide the ros2 bag command line tool. It also provides a C++ API for reading from and writing to a bag from your own source code. This allows you to subscribe to a topic and save the received data to a bag at the same time as performing any other processing of your choice on that data.

Prerequisites

You should have the rosbag2 packages installed as part of your regular ROS 2 setup.

If you’ve installed from Debian packages on Linux, it may be installed by default. If it is not, you can install it using this command.

```
sudo apt install ros-iron-rosbag2
```

This tutorial discusses using ROS 2 bags, including from the terminal. You should have already completed the basic ROS 2 bag tutorial.
Tasks

1 Create a package

Open a new terminal and source your ROS 2 installation so that ros2 commands will work. Navigate into the ros2_ws directory created in a previous tutorial. Navigate into the ros2_ws/src directory and create a new package:

```
ros2 pkg create --build-type ament_cmake bag_recorder_nodes --dependencies example_interfaces rclcpp rosbag2_cpp std_msgs
```

Your terminal will return a message verifying the creation of your package bag_recorder_nodes and all its necessary files and folders. The --dependencies argument will automatically add the necessary dependency lines to package.xml and CMakeLists.txt. In this case, the package will use the rosbag2_cpp package as well as the rclcpp package. A dependency on the example_interfaces package is also required for later parts of this tutorial.

1.1 Update package.xml

Because you used the --dependencies option during package creation, you don’t have to manually add dependencies to package.xml or CMakeLists.txt. As always, though, make sure to add the description, maintainer email and name, and license information to package.xml.

```
<description>C++ bag writing tutorial</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache License 2.0</license>
```

2 Write the C++ node

Inside the ros2_ws/src/bag_recorder_nodes/src directory, create a new file called simple_bag_recorder.cpp and paste the following code into it.

```
#include <rclcpp/rclcpp.hpp>
#include <std_msgs/msg/string.hpp>
#include <rosbag2_cpp/writer.hpp>

using std::placeholders::_1;

class SimpleBagRecorder : public rclcpp::Node
{
public:
    SimpleBagRecorder()
    : Node("simple_bag_recorder")
    {
        writer_ = std::make_unique<rosbag2_cpp::Writer>();
        writer_->open("my_bag");
        subscription_ = create_subscription<std_msgs::msg::String>(
            "chatter", 10, std::bind(&SimpleBagRecorder::topic_callback, this, _1));
    }
};
```

(continues on next page)
2.1 Examine the code

The \texttt{#include} statements at the top are the package dependencies. Note the inclusion of headers from the \texttt{rosbag2\_cpp} package for the functions and structures necessary to work with bag files.

In the class constructor we begin by creating the writer object we will use to write to the bag.

```cpp
writer_ = std::make_unique<rosbag2\_cpp::Writer>();
```

Now that we have a writer object, we can open the bag using it. We specify just the URI of the bag to create, leaving other options at their defaults. The default storage options are used, which means that an \texttt{mcap}-format bag will be created. The default conversion options are used, too, which will perform no conversion, instead storing messages in the serialisation format they are received in.

```cpp
writer_->open("my\_bag");
```

With the writer now set up to record data we pass to it, we create a subscription and specify a callback for it. We will write data to the bag in the callback.

```cpp
subscription_ = create\_subscription<\texttt{std\-msgs}::\texttt{msg::String}>(
  "chatter", 10, std\::bind(&SimpleBagRecorder::topic\_callback, this, _1));
```

The callback itself is different from a typical callback. Rather than receiving an instance of the data type of the topic, we instead receive a \texttt{rclcpp}::\texttt{SerializedMessage}. We do this for two reasons.

1. The message data will need to be serialised by \texttt{robag2} before being written to the bag, so rather than unserialising it when receiving the data and then re-serialising it, we ask ROS to just give us the serialised message as-is.
2. The writer API can accept a serialised message.
void topic_callback(std::shared_ptr<rclcpp::SerializedMessage> msg) const
{

Within the subscription callback, the first thing to do is determine the time stamp to use for the stored message. This can be anything appropriate to your data, but two common values are the time at which the data was produced, if known, and the time it is received. The second option, the time of reception, is used here.

rclcpp::Time time_stamp = this->now();

We can then write the message into the bag. Because we have not yet registered any topics with the bag, we must specify the full topic information with the message. This is why we pass in the topic name and the topic type.

writer_->write(msg, "chatter", "std_msgs/msg/String", time_stamp);

The class contains two member variables.

1. The subscription object.

2. A managed pointer to the writer object used to write to the bag. Note the type of writer used here is the rosbag2_cpp::Writer, the generic writer interface. Other writers may be available with different behaviours.

rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
std::unique_ptr<rosbag2_cpp::Writer> writer_;

The file finishes with the main function used to create an instance of the node and start ROS processing it.

int main(int argc, char * argv[])
{
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<SimpleBagRecorder>());
    rclcpp::shutdown();
    return 0;
}

2.2 Add executable

Now open the CMakeLists.txt file.

Near the top of the file, change CMAKE_CXX_STANDARD from 14 to 17.

# Default to C++17
if(NOT CMAKE_CXX_STANDARD)
    set(CMAKE_CXX_STANDARD 17)
endif()

Below the dependencies block, which contains find_package(rosbag2_cpp REQUIRED), add the following lines of code.

add_executable(simple_bag_recorder src/simple_bag_recorder.cpp)
ament_target_dependencies(simple_bag_recorder rclcpp rosbag2_cpp std_msgs)

install(TARGETS
    simple_bag_recorder
)

(continues on next page)
3 Build and run

Navigate back to the root of your workspace, ros2_ws, and build your new package.

Linux

```bash
colcon build --packages-select bag_recorder_nodes
```

macOS

```bash
colcon build --packages-select bag_recorder_nodes
```

Windows

```bash
colcon build --merge-install --packages-select bag_recorder_nodes
```

Open a new terminal, navigate to ros2_ws, and source the setup files.

Linux

```bash
source install/setup.bash
```

macOS

```bash
source install/setup.bash
```

Windows

```bash
call install/setup.bat
```

Now run the node:

```bash
ros2 run bag_recorder_nodes simple_bag_recorder
```

Open a second terminal and run the talker example node.

```bash
ros2 run demo_nodes_cpp talker
```

This will start publishing data on the chatter topic. As the bag-writing node receives this data, it will write it to the my_bag bag.

Terminate both nodes. Then, in one terminal start the listener example node.

```bash
ros2 run demo_nodes_cpp listener
```

In the other terminal, use ros2 bag to play the bag recorded by your node.

```bash
ros2 bag play my_bag
```

You will see the messages from the bag being received by the listener node.

If you wish to run the bag-writing node again, you will first need to delete the my_bag directory.
4 Record synthetic data from a node

Any data can be recorded into a bag, not just data received over a topic. A common use case for writing to a bag from your own node is to generate and store synthetic data. In this section you will learn how to write a node that generates some data and stores it in a bag. We will demonstrate two approaches for doing this. The first uses a node with a timer; this is the approach that you would use if your data generation is external to the node, such as reading data directly from hardware (e.g. a camera). The second approach does not use a node; this is the approach you can use when you do not need to use any functionality from the ROS infrastructure.

4.1 Write a C++ node

Inside the ros2_ws/src/bag_recorder_nodes/src directory, create a new file called data_generator_node.cpp and paste the following code into it.

```cpp
#include <chrono>
#include <example_interfaces/msg/int32.hpp>
#include <rclcpp/rclcpp.hpp>
#include <rosbag2_cpp/writer.hpp>

using namespace std::chrono_literals;

class DataGenerator : public rclcpp::Node
{
public:
    DataGenerator()
    : Node("data_generator")
    {
        data_.data = 0;
        writer_ = std::make_unique<rosbag2_cpp::Writer>();

        writer_->open("timed_synthetic_bag");

        writer_->create_topic(
            {"synthetic",
             "example_interfaces/msg/Int32",
             rmw_get_serialization_format(),
             ""});

        timer_ = create_wall_timer(1s, std::bind(&DataGenerator::timer_callback, this));
    }

private:
    void timer_callback()
    {
        writer_->write(data_, "synthetic", now());

        ++data_.data;
    }

    rclcpp::TimerBase::SharedPtr timer_;
}
```
4.2 Examine the code

Much of this code is the same as the first example. The important differences are described here.

First, the name of the bag is changed.

```cpp
writer_->open("timed_synthetic_bag");
```

In this example we are registering the topic with the bag in advance. This is optional in most cases, but it must be done when passing in a serialised message without topic information.

```cpp
writer_->create_topic(
    {"synthetic",
     "example_interfaces/msg/Int32",
     rmw_get_serialization_format(),
     ""});
```

Rather than a subscription to a topic, this node has a timer. The timer fires with a one-second period, and calls the given member function when it does.

```cpp
timer_ = create_wall_timer(1s, std::bind(&DataGenerator::timer_callback, this));
```

Within the timer callback, we generate (or otherwise obtain, e.g. read from a serial port connected to some hardware) the data we wish to store in the bag. The important difference between this and the previous sample is that the data is not yet serialised. Instead we are passing a ROS message data type to the writer object, in this case an instance of `example_interfaces/msg/Int32`. The writer will serialise the data for us before writing it into the bag.

```cpp
writer_->write(data_, "synthetic", now());
```

4.3 Add executable

Open the `CMakeLists.txt` file and add the following lines after the previously-added lines (specifically, after the `install(TARGETS ...)` macro call).

```cpp
add_executable(data_generator_node src/data_generator_node.cpp)
ament_target_dependencies(data_generator_node rclcpp rosbag2_cpp example_interfaces)
install(TARGETS
```

(continues on next page)
4.4 Build and run

Navigate back to the root of your workspace, ros2_ws, and build your package.

Linux

```bash
colcon build --packages-select bag_recorder_nodes
```

macOS

```bash
colcon build --packages-select bag_recorder_nodes
```

Windows

```bash
colcon build --merge-install --packages-select bag_recorder_nodes
```

Open a new terminal, navigate to ros2_ws, and source the setup files.

Linux

```bash
source install/setup.bash
```

macOS

```bash
source install/setup.bash
```

Windows

```bash
call install/setup.bat
```

(If the timed_synthetic_bag directory already exists, you must first delete it before running the node.)

Now run the node:

```bash
ros2 run bag_recorder_nodes data_generator_node
```

Wait for 30 seconds or so, then terminate the node with `ctrl-c`. Next, play back the created bag.

```bash
ros2 bag play timed_synthetic_bag
```

Open a second terminal and echo the `/synthetic` topic.

```bash
ros2 topic echo /synthetic
```

You will see the data that was generated and stored in the bag printed to the console at a rate of one message per second.
5 Record synthetic data from an executable

Now that you can create a bag that stores data from a source other than a topic, you will learn how to generate and record synthetic data from a non-node executable. The advantage of this approach is simpler code and rapid creation of a large quantity of data.

5.1 Write a C++ executable

Inside the `ros2_ws/src/bag_recorder_nodes/src` directory, create a new file called `data_generator_executable.cpp` and paste the following code into it.

```cpp
#include <chrono>
#include <rclcpp/rclcpp.hpp> // For rclcpp::Clock, rclcpp::Duration and rclcpp::Time
#include <example_interfaces/msg/int32.hpp>
#include <rosbag2_cpp/writer.hpp>
#include <rosbag2_cpp/writers/sequential_writer.hpp>
#include <rosbag2_storage/serialized_bag_message.hpp>

using namespace std::chrono_literals;

int main(int argc, char** argv) {
  example_interfaces::msg::Int32 data;
  data.data = 0;
  std::unique_ptr<rosbag2_cpp::Writer> writer_ = std::make_unique<rosbag2_cpp::Writer>();

  writer_->open("big_synthetic_bag");
  writer_->create_topic(
    "synthetic",
    "example_interfaces/msg/Int32",
    rmw_get_serialization_format(),
    ""
  );

  rclcpp::Clock clock;
  rclcpp::Time time_stamp = clock.now();
  for (int32_t ii = 0; ii < 100; ++ii) {
    writer_->write(data, "synthetic", time_stamp);
    ++data.data;
    time_stamp += rclcpp::Duration(1s);
  }

  return 0;
}
```
5.2 Examine the code

A comparison of this sample and the previous sample will reveal that they are not that different. The only significant difference is the use of a for loop to drive the data generation rather than a timer.

Notice that we are also now generating time stamps for the data rather than relying on the current system time for each sample. The time stamp can be any value you need it to be. The data will be played back at the rate given by these time stamps, so this is a useful way to control the default playback speed of the samples. Notice also that while the gap between each sample is a full second in time, this executable does not need to wait a second between each sample. This allows us to generate a lot of data covering a wide span of time in much less time than playback will take.

```cpp
rclcpp::Clock clock;
rclcpp::Time time_stamp = clock.now();
for (int32_t ii = 0; ii < 100; ++ii) {
    writer_->write(data, "synthetic", time_stamp);
    ++data.data;
    time_stamp += rclcpp::Duration(1s);
}
```

5.3 Add executable

Open the CMakeLists.txt file and add the following lines after the previously-added lines.

```makefile
add_executable(data_generator_executable src/data_generator_executable.cpp)
ament_target_dependencies(data_generator_executable rclcpp rosbag2_cpp example_interfaces)
install(TARGETS
data_generator_executable
    DESTINATION lib/${PROJECT_NAME}
)
```

5.4 Build and run

Navigate back to the root of your workspace, `ros2_ws`, and build your package.

Linux

```
colcon build --packages-select bag_recorder_nodes
```

macOS

```
colcon build --packages-select bag_recorder_nodes
```

Windows

```
colcon build --merge-install --packages-select bag_recorder_nodes
```

Open a terminal, navigate to `ros2_ws`, and source the setup files.

Linux
source install/setup.bash

macOS
source install/setup.bash

Windows
call install/setup.bat

(If the big_synthetic_bag directory already exists, you must first delete it before running the executable.)

Now run the executable:

```
ros2 run bag_recorder_nodes data_generator_executable
```

Now play back the created bag.

```
ros2 bag play big_synthetic_bag
```

Open a second terminal and echo the /synthetic topic.

```
ros2 topic echo /synthetic
```

You will see the data that was generated and stored in the bag printed to the console at a rate of one message per second. Even though the bag was generated rapidly it is still played back at the rate the time stamps indicate.

**Summary**

You created a node that records data it receives on a topic into a bag. You tested recording a bag using the node, and verified the data was recorded by playing back the bag. You then went on to create a node and an executable to generate synthetic data and store it in a bag.

**Recording a bag from a node (Python)**

**Goal:** Record data from your own Python node to a bag.

**Tutorial level:** Advanced

**Time:** 20 minutes

**Contents**

- Background
- Prerequisites
- Tasks
  - 1 Create a package
  - 2 Write the Python node
  - 3 Build and run
Background

rosbag2 doesn’t just provide the ros2 bag command line tool. It also provides a Python API for reading from and writing to a bag from your own source code. This allows you to subscribe to a topic and save the received data to a bag at the same time as performing any other processing of your choice on that data. You may do this, for example, to save data from a topic and the result of processing that data without needing to send the processed data over a topic just to record it. Because any data can be recorded in a bag, it is also possible to save data generated by another source than a topic, such as synthetic data for training sets. This is useful, for example, for quickly generating a bag that contains a large number of samples spread over a long playback time.

Prerequisites

You should have the rosbag2 packages installed as part of your regular ROS 2 setup.

If you’ve installed from Debian packages on Linux, it may be installed by default. If it is not, you can install it using this command.

```
sudo apt install ros-iron-rosbag2
```

This tutorial discusses using ROS 2 bags, including from the terminal. You should have already completed the basic ROS 2 bag tutorial.

Tasks

1 Create a package

Open a new terminal and source your ROS 2 installation so that ros2 commands will work.

Follow these instructions to create a new workspace named ros2_ws.

Navigate into the ros2_ws/src directory and create a new package:

```
ros2 pkg create --build-type ament_python bag_recorder_nodes_py --dependencies rclpy__
˓→rosbag2_py example_interfaces std_msgs
```

Your terminal will return a message verifying the creation of your package bag_recorder_nodes_py and all its necessary files and folders. The --dependencies argument will automatically add the necessary dependency lines to the package.xml. In this case, the package will use the rosbag2_py package as well as the rclpy package. A dependency on the example_interfaces package is also required for message definitions.
1.1 Update package.xml and setup.py

Because you used the --dependencies option during package creation, you don’t have to manually add dependencies to package.xml. As always, though, make sure to add the description, maintainer email and name, and license information to package.xml.

```
<description>Python bag writing tutorial</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache License 2.0</license>
```

Also be sure to add this information to the setup.py file as well.

```
maintainer='Your Name',
maintainer_email='you@email.com',
description='Python bag writing tutorial',
license='Apache License 2.0',
```

2 Write the Python node

Inside the ros2_ws/src/bag_recorder_nodes_py/bag_recorder_nodes_py directory, create a new file called simple_bag_recorder.py and paste the following code into it.

```
import rclpy
from rclpy.node import Node
from rclpy.serialization import serialize_message
from std_msgs.msg import String
import rosbag2_py

class SimpleBagRecorder(Node):
    def __init__(self):
        super().__init__('simple_bag_recorder')
        self.writer = rosbag2_py.SequentialWriter()

        storage_options = rosbag2_py._storage.StorageOptions(
            uri='my_bag',
            storage_id='mcap')
        converter_options = rosbag2_py._storage.ConverterOptions('','')
        self.writer.open(storage_options, converter_options)

        topic_info = rosbag2_py._storage.TopicMetadata(
            name='chatter',
            type='std_msgs/msg/String',
            serialization_format='cdr')
        self.writer.create_topic(topic_info)

        self.subscription = self.create_subscription(String,
            'chatter',
            self.topic_callback,
            10)
        self.subscription

(continues on next page)
def topic_callback(self, msg):
    self.writer.write('chatter', serialize_message(msg),
                      self.get_clock().now().nanoseconds)

def main(args=None):
    rclpy.init(args=args)
    sbr = SimpleBagRecorder()
    rclpy.spin(sbr)
    rclpy.shutdown()

if __name__ == '__main__':
    main()

2.1 Examine the code

The import statements at the top are the package dependencies. Note the importation of the rosbag2_py package for
the functions and structures necessary to work with bag files.

In the class constructor, we begin by creating the writer object that we will use to write to the bag. We are creating a
SequentialWriter, which writes messages into the bag in the order they are received. Other writers with different
behaviours may be available in rosbag2.

    self.writer = rosbag2_py.SequentialWriter()

Now that we have a writer object, we can open the bag using it. We specify the URI of the bag to create and the
format (mcap), leaving other options at their defaults. The default conversion options are used, which will perform no
conversion and store the messages in the serialization format they are received in.

    storage_options = rosbag2_py._storage.StorageOptions(
        uri='my_bag',
        storage_id='mcap')
    converter_options = rosbag2_py._storage.ConverterOptions('', '')
    self.writer.open(storage_options, converter_options)

Next, we need to tell the writer about the topics we wish to store. This is done by creating a TopicMetadata object
and registering it with the writer. This object specifies the topic name, topic data type, and serialization format used.

    topic_info = rosbag2_py._storage.TopicMetadata(
        name='chatter',
        type='std_msgs/msg/String',
        serialization_format='cdr')
    self.writer.create_topic(topic_info)

With the writer now set up to record data we pass to it, we create a subscription and specify a callback for it. We will
write data to the bag in the callback.
self.subscription = self.create_subscription(  
    String,  
    'chatter',  
    self.topic_callback,  
    10)  
self.subscription

The callback receives the message in unserialized form (as is standard for the rclpy API) and passes the message to the writer, specifying the topic that the data is for and the timestamp to record with the message. However, the writer requires serialised messages to store in the bag. This means that we need to serialise the data before passing it to the writer. For this reason, we call `serialize_message()` and pass the result of that to the writer, rather than passing in the message directly.

def topic_callback(self, msg):
    self.writer.write(  
        'chatter',  
        serialize_message(msg),  
        self.get_clock().now().nanoseconds)

The file finishes with the main function used to create an instance of the node and start ROS processing it.

def main(args=None):
    rclpy.init(args=args)
    sbr = SimpleBagRecorder()
    rclpy.spin(sbr)
    rclpy.shutdown()
colcon build --merge-install --packages-select bag_recorder_nodes_py

Open a new terminal, navigate to `ros2_ws`, and source the setup files.

Linux

source install/setup.bash

macOS

source install/setup.bash

Windows

call install/setup.bat

Now run the node:

`ros2 run bag_recorder_nodes_py simple_bag_recorder`

Open a second terminal and run the `talker` example node.

`ros2 run demo_nodes_cpp talker`

This will start publishing data on the `chatter` topic. As the bag-writing node receives this data, it will write it to the `my_bag` bag. If the `my_bag` directory already exists, you must first delete it before running the `simple_bag_recorder` node. This is because `rosbag2` will not overwrite existing bags by default, and so the destination directory cannot exist.

Terminate both nodes. Then, in one terminal start the `listener` example node.

`ros2 run demo_nodes_cpp listener`

In the other terminal, use `ros2 bag` to play the bag recorded by your node.

`ros2 bag play my_bag`

You will see the messages from the bag being received by the `listener` node.

If you wish to run the bag-writing node again, you will first need to delete the `my_bag` directory.

4 Record synthetic data from a node

Any data can be recorded into a bag, not just data received over a topic. A common use case for writing to a bag from your own node is to generate and store synthetic data. In this section you will learn how to write a node that generates some data and stores it in a bag. We will demonstrate two approaches for doing this. The first uses a node with a timer; this is the approach that you would use if your data generation is external to the node, such as reading data directly from hardware (e.g. a camera). The second approach does not use a node; this is the approach you can use when you do not need to use any functionality from the ROS infrastructure.
4.1 Write a Python node

Inside the `ros2_ws/src/bag_recorder_nodes_py/bag_recorder_nodes_py` directory, create a new file called `data_generator_node.py` and paste the following code into it.

```python
import rclpy
from rclpy.node import Node
from rclpy.serialization import serialize_message
from example_interfaces.msg import Int32

import rosbag2_py

class DataGeneratorNode(Node):
    def __init__(self):
        super().__init__('data_generator_node')
        self.data = Int32()
        self.data.data = 0
        self.writer = rosbag2_py.SequentialWriter()

        storage_options = rosbag2_py._storage.StorageOptions(
            uri='timed_synthetic_bag',
            storage_id='mcap')
        converter_options = rosbag2_py._storage.ConverterOptions('', '')
        self.writer.open(storage_options, converter_options)

        topic_info = rosbag2_py._storage.TopicMetadata(
            name='synthetic',
            type='example_interfaces/msg/Int32',
            serialization_format='cdr')
        self.writer.create_topic(topic_info)

        self.timer = self.create_timer(1, self.timer_callback)

    def timer_callback(self):
        self.writer.write('synthetic', serialize_message(self.data),
                          self.get_clock().now().nanoseconds)
        self.data.data += 1

def main(args=None):
    rclpy.init(args=args)
    dgn = DataGeneratorNode()
    rclpy.spin(dgn)
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```
4.2 Examine the code

Much of this code is the same as the first example. The important differences are described here.

First, the name of the bag is changed.

```python
storage_options = rosbag2_py._storage.StorageOptions(
    uri='timed_synthetic_bag',
    storage_id='mcap')
```

The name of the topic is also changed, as is the data type stored.

```python
topic_info = rosbag2_py._storage.TopicMetadata(
    name='synthetic',
    type='example_interfaces/msg/Int32',
    serialization_format='cdr')
self.writer.create_topic(topic_info)
```

Rather than a subscription to a topic, this node has a timer. The timer fires with a one-second period, and calls the given member function when it does.

```python
self.timer = self.create_timer(1, self.timer_callback)
```

Within the timer callback, we generate (or otherwise obtain, e.g. read from a serial port connected to some hardware) the data we wish to store in the bag. As with the previous example, the data is not yet serialised, so we must serialise it before passing it to the writer.

```python
self.writer.write(
    'synthetic',
    serialize_message(self.data),
    self.get_clock().now().nanoseconds)
```

4.3 Add executable

Open the setup.py file in the bag_recorder_nodes_py package and add an entry point for your node.

```python
entry_points={
    'console_scripts': [
        'simple_bag_recorder = bag_recorder_nodes_py.simple_bag_recorder:main',
        'data_generator_node = bag_recorder_nodes_py.data_generator_node:main',
    ],
},
```

4.4 Build and run

Navigate back to the root of your workspace, ros2_ws, and build your package.

Linux

```bash
colcon build --packages-select bag_recorder_nodes_py
```

macOS
colcon build --packages-select bag_recorder_nodes_py

Windows

colcon build --merge-install --packages-select bag_recorder_nodes_py

Open a new terminal, navigate to `ros2_ws`, and source the setup files.

Linux

source install/setup.bash

macOS

source install/setup.bash

Windows

call install/setup.bat

If the `timed_synthetic_bag` directory already exists, you must first delete it before running the node.

Now run the node:

```
ros2 run bag_recorder_nodes_py data_generator_node
```

Wait for 30 seconds or so, then terminate the node with `ctrl-c`. Next, play back the created bag.

```
ros2 bag play timed_synthetic_bag
```

Open a second terminal and echo the `/synthetic` topic.

```
ros2 topic echo /synthetic
```

You will see the data that was generated and stored in the bag printed to the console at a rate of one message per second.

### 5 Record synthetic data from an executable

Now that you can create a bag that stores data from a source other than a topic, you will learn how to generate and record synthetic data from a non-node executable. The advantage of this approach is simpler code and rapid creation of a large quantity of data.

#### 5.1 Write a Python executable

Inside the `ros2_ws/src/bag_recorder_nodes_py/bag_recorder_nodes_py` directory, create a new file called `data_generator_executable.py` and paste the following code into it.

```python
from rclpy.clock import Clock
from rclpy.duration import Duration
from rclpy.serialization import serialize_message
from example_interfaces.msg import Int32

import rosbag2_py
```

(continues on next page)
def main(args=None):
    writer = rosbag2_py.SequentialWriter()
    storage_options = rosbag2_py._storage.StorageOptions(
        uri='big_synthetic_bag',
        storage_id='mcap')
    converter_options = rosbag2_py._storage.ConverterOptions('', '')
    writer.open(storage_options, converter_options)
    topic_info = rosbag2_py._storage.TopicMetadata(
        name='synthetic',
        type='example_interfaces/msg/Int32',
        serialization_format='cdr')
    writer.create_topic(topic_info)
    time_stamp = Clock().now()
    for ii in range(0, 100):
        data = Int32()
        data.data = ii
        writer.write('synthetic', serialize_message(data), time_stamp.nanoseconds)
        time_stamp += Duration(seconds=1)

if __name__ == '__main__':
    main()

5.2 Examine the code

A comparison of this sample and the previous sample will reveal that they are not that different. The only significant
difference is the use of a for loop to drive the data generation rather than a timer.

Notice that we are also now generating time stamps for the data rather than relying on the current system time for each
sample. The time stamp can be any value you need it to be. The data will be played back at the rate given by these
time stamps, so this is a useful way to control the default playback speed of the samples. Notice also that while the
gap between each sample is a full second in time, this executable does not need to wait a second between each sample.
This allows us to generate a lot of data covering a wide span of time in much less time than playback will take.
5.3 Add executable

Open the setup.py file in the bag_recorder_nodes_py package and add an entry point for your node.

```python
entry_points={
    'console_scripts': [
        'simple_bag_recorder = bag_recorder_nodes_py.simple_bag_recorder:main',
        'data_generator_node = bag_recorder_nodes_py.data_generator_node:main',
        'data_generator_executable = bag_recorder_nodes_py.data_generator_executable:main',
    ],
},
```

5.4 Build and run

Navigate back to the root of your workspace, ros2_ws, and build your package.

Linux

```
colcon build --packages-select bag_recorder_nodes_py
```

macOS

```
colcon build --packages-select bag_recorder_nodes_py
```

Windows

```
colcon build --merge-install --packages-select bag_recorder_nodes_py
```

Open a terminal, navigate to ros2_ws, and source the setup files.

Linux

```
source install/setup.bash
```

macOS

```
source install/setup.bash
```

Windows

```
call install/setup.bat
```

If the big_synthetic_bag directory already exists, you must first delete it before running the executable.

Now run the executable:

```
ros2 run bag_recorder_nodes_py data_generator_executable
```

Note that the executable runs and finishes very quickly.

Now play back the created bag.

```
ros2 bag play big_synthetic_bag
```

Open a second terminal and echo the /synthetic topic.
You will see the data that was generated and stored in the bag printed to the console at a rate of one message per second. Even though the bag was generated rapidly it is still played back at the rate the time stamps indicate.

**Summary**

You created a node that records data it receives on a topic into a bag. You tested recording a bag using the node, and verified the data was recorded by playing back the bag. This approach can be used to record a bag with additional data than it received over a topic, for example with results obtained from processing the received data. You then went on to create a node and an executable to generate synthetic data and store it in a bag. The latter approaches are useful especially for generating synthetic data that can be used, for example, as training sets.

**Simulators**

Several advanced robot simulators can be used with ROS 2, such as Gazebo, Webots, etc. Unlike turtlesim, they provide fairly realistic results relying on physics-based models for robots, sensors, actuators and objects. Hence, what you observe in simulation is very close to what you will get when transferring your ROS 2 controllers to a real robot. This set of tutorials will teach you how to configure different simulators with ROS 2.

**Webots**

This set of tutorials will teach you how to configure the Webots simulator with ROS 2.

**Installation (Ubuntu)**

**Goal:** Install the `webots_ros2` package and run simulation examples on Ubuntu.

**Tutorial level:** Advanced

**Time:** 10 minutes

**Contents**

- Background
- Prerequisites
  - Multiple Installations of Webots
- Tasks
  - 1 Install `webots_ros2`
  - 2 Launch the `webots_ros2_universal_robot` example
Background

The `webots_ros2` package provides an interface between ROS 2 and Webots. It includes several sub-packages, including `webots_ros2_driver`, which allows you to start Webots and communicate with it. This interface is used in most of the following tutorials, so it is required to install it beforehand. Other sub-packages are mainly examples that show multiple possible implementations using the interface. In this tutorial, you are going to install the package and learn how to run one of these examples.

Prerequisites

It is recommended to understand basic ROS principles covered in the beginner Tutorials. In particular, Creating a workspace and Creating a package are useful prerequisites.

The Webots software should be installed in order to use the `webots_ros2` interface. You can follow the installation procedure or build it from sources.

Alternatively, you can also let `webots_ros2` download and install Webots automatically. This option appears when you launch an example of the package and no Webots installation is found.

Multiple Installations of Webots

If you have installed different versions of Webots on your computer, `webots_ros2` will look for Webots at the following locations (in this order):

1. If the `ROS2_WEBOTS_HOME` environment variable is set, ROS 2 will use the Webots in this folder, regardless of its version.
2. If the `WEBOTS_HOME` environment variable is set, ROS 2 will use the Webots in this folder, regardless of its version.
3. If none of these variables is set, `webots_ros2` will look for Webots in the default installation paths for a compatible version: `/usr/local/webots` and `/snap/webots/current/usr/share/webots`.
4. If Webots couldn't be found, `webots_ros2` will show a window offering the automatic installation of the latest compatible version of Webots.

Tasks

1 Install `webots_ros2`

You can either install the official released package, or install it from the latest up-to-date sources from Github.

Install `webots_ros2` distributed package

Run the following command in a terminal.

```
sudo apt-get install ros-iron-webots-ros2
```

Install `webots_ros2` from sources

Create a ROS 2 workspace with its `src` directory.

```
mkdir -p ~/ros2_ws/src
```

Source the ROS 2 environment.
source /opt/ros/iron/setup.bash

Retrieve the sources from Github.

```
cd ~/ros2_ws
git clone --recurse-submodules https://github.com/cyberbotics/webots_ros2.git src/webots_ros2
```

Install the package dependencies.

```
sudo apt install python3-pip python3-rosdep python3-colcon-common-extensions
sudo rosdep init && rosdep update
rosdep install --from-paths src --ignore-src --rosdistro iron
```

Build the package using colcon.

```
colcon build
```

Source this workspace.

```
source install/local_setup.bash
```

## 2 Launch the `webots_ros2_universal_robot` example

The following instructions explain how to start a provided example.

First source the ROS 2 environment, if not done already.

```
source /opt/ros/iron/setup.bash
```

Setting the `WEBOTS_HOME` environment variable allows you to start a specific Webots installation.

```
export WEBOTS_HOME=/usr/local/webots
```

If installed from sources, source your ROS 2 workspace, if not done already.

```
cd ~/ros2_ws
source install/local_setup.bash
```

Use the ROS 2 launch command to start demo packages (e.g. `webots_ros2_universal_robot`).

```
ros2 launch webots_ros2_universal_robot multirobot_launch.py
```

### Installation (Windows)

**Goal:** Install the `webots_ros2` package and run simulation examples on Windows.

**Tutorial level:** Advanced

**Time:** 10 minutes
Background

The `webots_ros2` package provides an interface between ROS 2 and Webots. It includes several sub-packages, including `webots_ros2_driver`, which allows ROS nodes to communicate with Webots. Other sub-packages are mainly examples that show multiple possible implementations using the interface. In this tutorial, you are going to install the package and learn how to run one of these examples.

Prerequisites

It is recommended to understand basic ROS principles covered in the beginner Tutorials. In particular, Creating a workspace and Creating a package are useful prerequisites.

Webots is a prerequisite to use the `webots_ros2` package. You can follow the installation procedure or build it from sources.

Alternatively, you can also let `webots_ros2` download Webots automatically. This option appears when you launch an example of the package and no Webots installation is found.

Multiple Installations of Webots

If you have more than one installation of Webots, ROS 2 will look for Webots at the following locations (in this order):

1. If the `ROS2_WEBOTS_HOME` environment variable is set, ROS 2 will use the Webots in this folder, regardless of its version.
2. If the `WEBOTS_HOME` environment variable is set, ROS 2 will use the Webots in this folder, regardless of its version.
3. If none of the previous points is set/installed ROS 2 will look for Webots in the default installation paths for a compatible version: C:\Program Files\Webots.
4. If Webots couldn’t be found, `webots_ros2` will show a window and offer automatic Webots installation of the last compatible version.
Tasks

1 Install WSL2

On Windows, WSL (Windows Subsystem for Linux) improves the user experience with ROS 2 compared to native Windows installation, as it runs on a Linux platform. Install WSL with an Ubuntu version which is compatible with your ROS distribution and upgrade to WSL2 following the official Microsoft tutorial.

2 Install ROS 2 in WSL

Install ROS 2 inside Ubuntu WSL, following *Ubuntu (Debian packages)*.

3 Install *webots_ros2*

You can then either install *webots_ros2* from the official released package, or install it from the latest up-to-date sources from Github.

The following commands must be run inside the WSL environment.

Install *webots_ros2* distributed package

Run the following command in a terminal.

```
sudo apt-get install ros-iron-webots-ros2
```

Install *webots_ros2* from sources

Create a ROS 2 workspace with its `src` directory.

```
mkdir -p ~/ros2_ws/src
```

Source the ROS 2 environment.

```
source /opt/ros/iron/setup.bash
```

Retrieve the sources from Github.

```
cd ~/ros2_ws
git clone --recurse-submodules https://github.com/cyberbotics/webots_ros2.git src/webots_\--ros2
```

Install the package dependencies.

```
sudo apt install python3-pip python3-rosdep python3-colcon-common-extensions
sudo rosdep init && rosdep update
rosdep install --from-paths src --ignore-src --rosdistro iron
```

Build the package using `colcon`.

```
colcon build
```

Source this workspace.

```
source install/local_setup.bash
```
4 Launch the webots_ros2_universal_robot example

WSL doesn’t support hardware acceleration (yet). Therefore, Webots should be started on Windows, while the ROS part is running inside WSL. To do so, the following commands must be run inside the WSL environment.

First source the ROS 2 environment, if not done already.

```bash
source /opt/ros/iron/setup.bash
```

Setting the `WEBOTS_HOME` environment variable allows you to start a specific Webots installation (e.g. `C:\Program Files\Webots`). Use the mount point `/mnt` to refer to a path on native Windows.

```bash
export WEBOTS_HOME=/mnt/c/Program Files/Webots
```

If installed from sources, source your ROS 2 workspace, if not done already.

```bash
cd ~/ros2_ws
source install/local_setup.bash
```

Use the ROS 2 launch command to start demo packages (e.g. `webots_ros2_universal_robot`).

```bash
ros2 launch webots_ros2_universal_robot multirobot_launch.py
```

5 RViz troubleshooting

With recent versions of WSL2, RViz should work out of the box.

You can check if it works correctly by running any example that uses RViz, for example:

```bash
sudo apt install ros-iron-slam-toolbox
ros2 launch webots_ros2_tiago robot_launch.py rviz:=true slam:=true
```

The Tiago robot can be controlled using:

```bash
ros2 run teleop_twist_keyboard teleop_twist_keyboard
```

With older WSL versions, RViz2 may not work directly, as no display is available. To use RViz, you can either upgrade WSL or enable X11 forwarding.

Upgrade WSL

In a Windows shell:

```bash
wsl --update
```

Enable X11 forwarding

For older versions of WSL, the following steps can be followed:

1. Install `VcXsrv`.
2. Launch `VcXsrv`. You can leave most of the parameters default, except the Extra settings page, where you must set Clipboard, Primary Selection and Disable access control and unset Native opengl.
3. You can save the configuration for future launches.
4. Click on Finish, you will see that the X11 server is running in the icon tray.
5. In your WSL environment, export the `DISPLAY` variable.
export DISPLAY=$(ip route list default | awk '{print }'):0
You can add this to your .bashrc, so that it is set for every future WSL environment.
echo "export DISPLAY=$(ip route list default | awk '{print }'):0" >> ~/.bashrc

Installation (macOS)

Goal: Install the webots_ros2 package and run simulation examples on macOS.

Tutorial level: Advanced

Time: 10 minutes

Contents

- Background
- Prerequisites
- Tasks
  - 1 Create the VM image
  - 2 Configure the VM
  - 3 Install webots_ros2
  - 4 Launch the webots_ros2_universal_robot example
- Pre-configured Images

Background

The webots_ros2 package provides an interface between ROS 2 and Webots. It includes several sub-packages, including webots_ros2_driver, which allows you to start Webots and communicate with it. Other sub-packages are mainly examples that show multiple possible implementations using the interface. In this tutorial, you are going to install the package and learn how to run one of these examples.

Prerequisites

It is recommended to understand basic ROS principles covered in the beginner Tutorials. In particular, Creating a workspace and Creating a package are useful prerequisites.

It is necessary to install Webots natively on the mac in order to use the webots_ros2 package in the virtual machine as explained below. You can follow the installation procedure or build it from sources.
Tasks

On macOS, a solution based on UTM virtual machines provides an improved user experience with ROS 2 compared to native macOS installation, as it runs ROS in a Linux environment. However, Webots should be installed natively on macOS and it will be able to communicate with the ROS nodes running in the Virtual Machine (VM). This solution allows for native 3D hardware acceleration for Webots. The VM runs all the ROS part (including RViz) and connects to the host machine through TCP to start Webots. A shared folder allows the script to transfer the world and other resource files from the VM to macOS where Webots is running.

The following steps explain how to create the VM image with the installation of the `webots_ros2` released package. It is also possible to install it from sources. In the Pre-configured Images section, you can find already configured images for every release of Webots (starting from R2023a) to download.

1 Create the VM image

Install UTM on your macOS machine. The link can be found on the official UTM website.

Download the .iso image of Ubuntu 22.04 for Humble and Rolling or Ubuntu 20.04 for Foxy. Be sure to download the image corresponding to your CPU architecture.

In the UTM software:

- Create a new image and choose Virtualize option.
- Select the ISO image you have downloaded in the Boot ISO Image field.
- Leave all hardware settings at default (including hardware acceleration disabled).
- In the Shared Directory window, select a folder that will be used by `webots_ros2` to transfer all the Webots assets to the host. In this example, the selected folder is `/Users/username/shared`.
- Leave all the remaining parameters as default.
- Start the VM. Note that you can select another shared folder each time you start the VM.
- During the first launch of the VM, install Ubuntu and choose a username for your account. In this example, the username is `ubuntu`.
- Once Ubuntu is installed, close the VM, remove the iso image from the CD/DVD field and restart the VM.

2 Configure the VM

In this section, ROS 2 is installed in the VM and the shared folder is configured. The following instructions and commands are all run inside the VM.

- Open a terminal in the started VM and install the ROS 2 distribution you need by following the instructions in Ubuntu (Debian packages):
- Create a folder in the VM to use as a shared folder. In this example, the shared folder in the VM is `/home/ubuntu/shared`.

```bash
dir /home/ubuntu/shared
```
- To mount this folder to the host, execute the following command. Don’t forget to modify the path to the shared folder, if it is different in your case.

```bash
sudo mount -t 9p -o trans=virtio share /home/ubuntu/shared -oversion=9p2000.L
```
• To automatically mount this folder to the host when starting the VM, add the following line to /etc/fstab. Don’t forget to modify the path to the shared folder, if it is different in your case.

```bash
share /home/ubuntu/shared 9p trans=virtio,version=9p\000.L,rw,_netdev, →nofail 0 0
```

• The environment variable `WEBOTS_SHARED_FOLDER` must always be set in order for the package to work properly in the VM. This variable specifies the location of the shared folder that is used to exchange data between the host machine and the virtual machine (VM) to the `webots_ros2` package. The value to use for this variable should be in the format of `<host shared folder>:<VM shared folder>`, where `<host shared folder>` is the path to the shared folder on the host machine and `<VM shared folder>` is the path to the same shared folder on the VM.

In this example:

```bash
export WEBOTS_SHARED_FOLDER=/Users/username/shared:/home/ubuntu/shared
```

You can add this command line to the `~/.bashrc` file to automatically set this environment variable when starting a new terminal.

### 3 Install `webots_ros2`

You can either install `webots_ros2` from the official released package, or install it from the latest up-to-date sources from Github.

**Install `webots_ros2` distributed package**

Run the following command in the VM terminal.

```bash
sudo apt-get install ros-iron-webots-ros2
```

**Install `webots_ros2` from sources**

**Install git.**

```bash
sudo apt-get install git
```

Create a ROS 2 workspace with its `src` directory.

```bash
mkdir -p ~/ros2_ws/src
```

Source the ROS 2 environment.

```bash
source /opt/ros/iron/setup.bash
```

Retrieve the sources from Github.

```bash
cd ~/ros2_ws
git clone --recursive-submodules https://github.com/cyberbotics/webots_ros2.git src/webots_ →ros2
```

Install the package dependencies.

```bash
sudo apt install python3-pip python3-rosdep python3-colcon-common-extensions
sudo rosdep init && rosdep update
rosdep install --from-paths src --ignore-src --rosdistro iron
```
Build the package using colcon.

```
colcon build
```

Source this workspace.

```
source install/local_setup.bash
```

### 4 Launch the webots_ros2_universal_robot example

As mentioned in previous sections, the package uses the shared folder to communicate with Webots from the VM to the host. In order for Webots to be started on the host from the VM’s ROS package, a local TCP simulation server must be run.

The server can be downloaded here: `local_simulation_server.py`. Specify the Webots installation folder in `WEBOTS_HOME` environment variable (e.g. `/Applications/Webots.app`) and run the server using the following commands in a new terminal on the host (not in the VM):

```
export WEBOTS_HOME=/Applications/Webots.app
python3 local_simulation_server.py
```

In the VM, open a terminal and execute the following commands to start a package:

First source the ROS 2 environment, if not done already.

```
source /opt/ros/iron/setup.bash
```

If installed from sources, source your ROS 2 workspace, if not done already.

```
cd ~/ros2_ws
source install/local_setup.bash
```

If not already set in `~/.bashrc`, set `WEBOTS_SHARED_FOLDER` (see previous sections for details). Be sure to change the paths according to the location of your respective directories.

```
export WEBOTS_SHARED_FOLDER=/Users/username/shared:/home/ubuntu/shared
```

Use the ROS 2 launch command to start demo packages (e.g. `webots_ros2_universal_robot`).

```
ros2 launch webots_ros2_universal_robot multirobot_launch.py
```

If Webots is closed or the ROS 2 process is interrupted, the local server will automatically wait for a new package launch and the shared folder will be cleaned for the next run.

### Pre-configured Images

If you don’t want to setup the VM from scratch, the following links provide you with pre-configured UTM images for each version of Webots. The `webots_ros2` version is installed from the official repository (not from sources) and is typically the first one that is compatible with the corresponding Webots version. You are welcome to download an image and upgrade the package, or install it from sources if necessary.

- Version 2023.0.2 for Webots R2023a [6.6 GB]
When adding the downloaded image to the UTM software, you should also choose the path to the host shared folder before starting the VM in the drop-down menu (e.g. /Users/username/shared). Once the VM is started, the WEBOTS_SHARED_FOLDER environment variable must always be set for the package to work properly in the virtual machine (VM). This variable specifies to the webots_ros2 package the location of the shared folder that is used to exchange data between the host machine and the VM. The value for this variable should be in the format of <host shared folder>:<VM shared folder>, where <host shared folder> is the path to the shared folder on the host machine and <VM shared folder> is the path to the same shared folder on the VM.

In the pre-configured images, WEBOTS_SHARED_FOLDER is already set in ~/.bashrc. You will need to update it to use the correct path for the host folder:

```
export WEBOTS_SHARED_FOLDER=/Users/username/shared:/home/ubuntu/shared
```

Setting up a robot simulation (Basic)

Goal: Setup a robot simulation and control it from ROS 2.

Tutorial level: Advanced

Time: 30 minutes

Contents

- Background
- Prerequisites
- Tasks
  - 1 Create the package structure
  - 2 Setup the simulation world
  - 3 Edit the my_robot_driver plugin
  - 4 Create the my_robot.urdf file
  - 5 Create the launch file
  - 6 Edit additional files
  - 7 Test the code
- Summary
- Next steps

Background

In this tutorial, you are going to use the Webots robot simulator to set-up and run a very simple ROS 2 simulation scenario.

The webots_ros2 package provides an interface between ROS 2 and Webots. It includes several sub-packages, but in this tutorial, you are going to use only the webots_ros2_driver sub-package to implement a Python or C++ plugin controlling a simulated robot. Some other sub-packages contain demos with different robots such as the TurtleBot3. They are documented in the Webots ROS 2 examples page.
Prerequisites

It is recommended to understand basic ROS principles covered in the beginner *Tutorials*. In particular, *Using turtlesim, ros2, and rqt, Understanding topics, Creating a workspace, Creating a package* and *Creating a launch file* are useful prerequisites.

Linux

The Linux and ROS commands of this tutorial can be run in a standard Linux terminal. The following page *Installation (Ubuntu)* explains how to install the *webots_ros2* package on Linux.

Windows

The Linux and ROS commands of this tutorial must be run in a WSL (Windows Subsystem for Linux) environment. The following page *Installation (Windows)* explains how to install the *webots_ros2* package on Windows.

macOS

The Linux and ROS commands of this tutorial must be run in a pre-configured Linux Virtual Machine (VM). The following page *Installation (macOS)* explains how to install the *webots_ros2* package on macOS.

This tutorial is compatible with version 2023.1.0 of *webots_ros2* and Webots R2023b, as well as upcoming versions.

Tasks

1. Create the package structure

Let's organize the code in a custom ROS 2 package. Create a new package named *my_package* from the src folder of your ROS 2 workspace. Change the current directory of your terminal to *ros2_ws/src* and run:

```bash
ros2 pkg create --build-type ament_python --license Apache-2.0 --node-name my_robot_driver my_package --dependencies rclpy geometry_msgs webots_ros2_driver
```

The `--node-name my_robot_driver` option will create a `my_robot_driver.py` template Python plugin in the `my_package` subfolder that you will modify later. The `--dependencies rclpy geometry_msgs webots_ros2_driver` option specifies the packages needed by the `my_robot_driver.py` plugin in the package `.xml` file.

Let's add a `launch` and a `worlds` folder inside the `my_package` folder.

```bash
cd my_package
mkdir launch
mkdir worlds
```

You should end up with the following folder structure:

```
src/
  └── my_package/
      ├── launch/
      │    └── __init__.py
      │    └── my_robot_driver.py
      └── resource/
          └── my_package
              └── test/
```

(continues on next page)
The \texttt{--node-name MyRobotDriver} option will create a \texttt{MyRobotDriver.cpp} template C++ plugin in the \texttt{my_package/src} subfolder that you will modify later. The \texttt{--dependencies rclcpp geometry_msgs webots_ros2_driver pluginlib} option specifies the packages needed by the \texttt{MyRobotDriver} plugin in the \texttt{package.xml} file.

Let's add a \texttt{launch}, a \texttt{worlds} and a \texttt{resource} folder inside the \texttt{my_package} folder.

```bash
cd my_package
mkdir launch
mkdir worlds
mkdir resource
```

Two additional files must be created: the header file for \texttt{MyRobotDriver} and the \texttt{my_robot_driver.xml} pluginlib description file.

```bash
touch my_robot_driver.xml
touch include/my_package/MyRobotDriver.hpp
```

You should end up with the following folder structure:

```
src/
  my_package/
    include/
      my_package/
        MyRobotDriver.hpp
    launch/
    resource/
    src/
      MyRobotDriver.cpp
    worlds/
    CMakeList.txt
    my_robot_driver.xml
    package.xml
```
2 Setup the simulation world

You will need a world file containing a robot to launch your simulation. Download this world file and move it inside my_package/worlds/.

This is actually a fairly simple text file you can visualize in a text editor. A simple robot is already included in this my_world.wbt world file.

Note: In case you want to learn how to create your own robot model in Webots, you can check this tutorial.

3 Edit the my_robot_driver plugin

The webots_ros2_driver sub-package automatically creates a ROS 2 interface for most sensors. More details on existing device interfaces and how to configure them is given in the second part of the tutorial: Setting up a robot simulation (Advanced). In this task, you will extend this interface by creating your own custom plugin. This custom plugin is a ROS node equivalent to a robot controller. You can use it to access the Webots robot API and create your own topics and services to control your robot.

Note: The purpose of this tutorial is to show a basic example with a minimum number of dependencies. However, you could avoid the use of this plugin by using another webots_ros2 sub-package named webots_ros2_control, introducing a new dependency. This other sub-package creates an interface with the ros2_control package that facilitates the control of a differential wheeled robot.

Python

Open my_package/my_package/my_robot_driver.py in your favorite editor and replace its contents with the following:

```python
import rclpy
from geometry_msgs.msg import Twist

HALF_DISTANCE_BETWEEN_WHEELS = 0.045
WHEEL_RADIUS = 0.025

class MyRobotDriver:
    def __init__(self, webots_node, properties):
        self.__robot = webots_node.robot

        self.__left_motor = self.__robot.getDevice('left wheel motor')
        self.__right_motor = self.__robot.getDevice('right wheel motor')

        self.__left_motor.setPosition(float('inf'))
        self.__left_motor.setVelocity(0)

        self.__right_motor.setPosition(float('inf'))
        self.__right_motor.setVelocity(0)

        self.__target_twist = Twist()

        rclpy.init(args=None)
```

(continues on next page)
```python
self.__node = rclpy.create_node('my_robot_driver')
self.__node.create_subscription(Twist, 'cmd_vel', self.__cmd_vel_callback, 1)

def __cmd_vel_callback(self, twist):
    self.__target_twist = twist

def step(self):
    rclpy.spin_once(self.__node, timeout_sec=0)
    forward_speed = self.__target_twist.linear.x
    angular_speed = self.__target_twist.angular.z

    command_motor_left = (forward_speed - angular_speed * HALF_DISTANCE_BETWEEN_WHEELS) / WHEEL_RADIUS
    command_motor_right = (forward_speed + angular_speed * HALF_DISTANCE_BETWEEN_WHEELS) / WHEEL_RADIUS

    self.__left_motor.setVelocity(command_motor_left)
    self.__right_motor.setVelocity(command_motor_right)
```

As you can see, the `MyRobotDriver` class implements three methods.

The first method, named `init(self, ...)`, is actually the ROS node counterpart of the Python `__init__(self, ...)` constructor. The `init` method always takes two arguments:

- The `webots_node` argument contains a reference on the Webots instance.
- The `properties` argument is a dictionary created from the XML tags given in the URDF files (4 Create the `my_robot.urdf` file) and allows you to pass parameters to the controller.

The robot instance from the simulation `self.__robot` can be used to access the Webots robot API. Then, it gets the two motor instances and initializes them with a target position and a target velocity. Finally a ROS node is created and a callback method is registered for a ROS topic named `/cmd_vel` that will handle Twist messages.

```python
def init(self, webots_node, properties):
    self.__robot = webots_node.robot

    self.__left_motor = self.__robot.getDevice('left wheel motor')
    self.__right_motor = self.__robot.getDevice('right wheel motor')

    self.__left_motor.setPosition(float('inf'))
    self.__left_motor.setVelocity(0)

    self.__right_motor.setPosition(float('inf'))
    self.__right_motor.setVelocity(0)

    self.__target_twist = Twist()

    rclpy.init(args=None)
    self.__node = rclpy.create_node('my_robot_driver')
    self.__node.create_subscription(Twist, 'cmd_vel', self.__cmd_vel_callback, 1)
```

Then comes the implementation of the `__cmd_vel_callback(self, twist)` callback private method that will be called for each Twist message received on the `/cmd_vel` topic and will save it in the `self.__target_twist` member variable.
Finally, the `step(self)` method is called at every time step of the simulation. The call to `rclpy.spin_once()` is needed to keep the ROS node running smoothly. At each time step, the method will retrieve the desired `forward_speed` and `angular_speed` from `self.__target_twist`. As the motors are controlled with angular velocities, the method will then convert the `forward_speed` and `angular_speed` into individual commands for each wheel. This conversion depends on the structure of the robot, more specifically on the radius of the wheel and the distance between them.

```python
def step(self):
    rclpy.spin_once(self.__node, timeout_sec=0)
    forward_speed = self.__target_twist.linear.x
    angular_speed = self.__target_twist.angular.z

    command_motor_left = (forward_speed - angular_speed * HALF_DISTANCE_BETWEEN_WHEELS) / WHEEL_RADIUS
    command_motor_right = (forward_speed + angular_speed * HALF_DISTANCE_BETWEEN_WHEELS) / WHEEL_RADIUS

    self.__left_motor.setVelocity(command_motor_left)
    self.__right_motor.setVelocity(command_motor_right)
```

C++

Open `my_package/src/MyRobotDriver.hpp` in your favorite editor and replace its contents with the following:

```c++
#ifndef WEBOTS_ROS2_PLUGIN_EXAMPLE_HPP
#define WEBOTS_ROS2_PLUGIN_EXAMPLE_HPP

#include "rclcpp/macros.hpp"
#include "webots_ros2_driver/PluginInterface.hpp"
#include "webots_ros2_driver/WebotsNode.hpp"

#include "geometry_msgs/msg/twist.hpp"
#include "rclcpp/rclcpp.hpp"

namespace my_robot_driver {
class MyRobotDriver : public webots_ros2_driver::PluginInterface {
public:
    void step() override;
    void init(webots_ros2_driver::WebotsNode *node,
              std::unordered_map<std::string, std::string> &parameters) override;

private:
    void cmdVelCallback(const geometry_msgs::msg::Twist::SharedPtr msg);

    rclcpp::Subscription<geometry_msgs::msg::Twist::SharedPtr
                       cmd_vel_subscription_;
    geometry_msgs::msg::Twist cmd_vel_msg;
    WbDeviceTag right_motor;
    WbDeviceTag left_motor;
};
```

(continues on next page)
The class MyRobotDriver is defined, which inherits from the webots_ros2_driver::PluginInterface class. The plugin has to override step(...) and init(...) functions. More details are given in the MyRobotDriver.cpp file. Several helper methods, callbacks and member variables that will be used internally by the plugin are declared privately.

Then, open my_package/src/MyRobotDriver.cpp in your favorite editor and replace its contents with the following:

```cpp
#include "my_package/MyRobotDriver.hpp"
#include "rclcpp/rclcpp.hpp"
#include <cstdio>
#include <functional>
#include <webots/motor.h>
#include <webots/robot.h>

#define HALF_DISTANCE_BETWEEN_WHEELS 0.045
#define WHEEL_RADIUS 0.025

namespace my_robot_driver {

void MyRobotDriver::init(
    webots_ros2_driver::WebotsNode *node,
    std::unordered_map<std::string, std::string> &parameters) {

    right_motor = wb_robot_get_device("right wheel motor");
    left_motor = wb_robot_get_device("left wheel motor");

    wb_motor_set_position(left_motor, INFINITY);
    wb_motor_set_velocity(left_motor, 0.0);

    wb_motor_set_position(right_motor, INFINITY);
    wb_motor_set_velocity(right_motor, 0.0);

    cmd_vel_subscription_ = node->create_subscription<geometry_msgs::msg::Twist>(
        "/cmd_vel", rclcpp::SensorDataQoS().reliable(),
        std::bind(&MyRobotDriver::cmdVelCallback,
        this
        std::placeholders::_1));
}

void MyRobotDriver::cmdVelCallback(
    const geometry_msgs::msg::Twist::SharedPtr msg) {
    cmd_vel_msg.linear = msg->linear;
    cmd_vel_msg.angular = msg->angular;
}

void MyRobotDriver::step() {
    auto forward_speed = cmd_vel_msg.linear.x;
    auto angular_speed = cmd_vel_msg.angular.z;

    auto command_motor_left =
        (forward_speed + angular_speed * HALF_DISTANCE_BETWEEN_WHEELS) / WHEEL_RADIUS;
```
auto command_motor_right =
    (forward_speed + angular_speed * HALF_DISTANCE_BETWEEN_WHEELS) / WHEEL_RADIUS;

wb_motor_set_velocity(left_motor, command_motor_left);
wb_motor_set_velocity(right_motor, command_motor_right);
}
} // namespace my_robot_driver

#include "pluginlib/class_list_macros.hpp"
PLUGINLIB_EXPORT_CLASS(my_robot_driver::MyRobotDriver,
    webots_ros2_driver::PluginInterface)

The MyRobotDriver::init method is executed once the plugin is loaded by the webots_ros2_driver package. It takes two arguments:

- A pointer to the WebotsNode defined by webots_ros2_driver, which allows to access the ROS 2 node functions.
- The parameters argument is an unordered map of strings, created from the XML tags given in the URDF files (4 Create the my_robot.urdf file) and allows to pass parameters to the controller. It is not used in this example.

It initializes the plugin by setting up the robot motors, setting their positions and velocities, and subscribing to the /cmd_vel topic.

```cpp
void MyRobotDriver::init(
    webots_ros2_driver::WebotsNode *node,
    std::unordered_map<std::string, std::string> &parameters) {

    right_motor = wb_robot_get_device("right wheel motor");
    left_motor = wb_robot_get_device("left wheel motor");

    wb_motor_set_position(left_motor, INFINITY);
    wb_motor_set_velocity(left_motor, 0.0);

    wb_motor_set_position(right_motor, INFINITY);
    wb_motor_set_velocity(right_motor, 0.0);

    cmd_vel_subscription_ = node->create_subscription<geometry_msgs::msg::Twist>(
        "/cmd_vel", rclcpp::SensorDataQoS().reliable(),
        std::bind(&MyRobotDriver::cmdVelCallback, this, std::placeholders::_1));
}
```

Then comes the implementation of the cmdVelCallback() callback function that will be called for each Twist message received on the /cmd_vel topic and will save it in the cmd_vel_msg member variable.

```cpp
void MyRobotDriver::cmdVelCallback(
    const geometry_msgs::msg::Twist::SharedPtr msg) {
    cmd_vel_msg.linear = msg->linear;
    cmd_vel_msg.angular = msg->angular;
}
```

The step() method is called at every time step of the simulation. At each time step, the method will retrieve the desired forward_speed and angular_speed from cmd_vel_msg. As the motors are controlled with angular velocities, the method will then convert the forward_speed and angular_speed into individual commands for each wheel. This
conversion depends on the structure of the robot, more specifically on the radius of the wheel and the distance between them.

```cpp
void MyRobotDriver::step() {
    auto forward_speed = cmd_vel_msg.linear.x;
    auto angular_speed = cmd_vel_msg.angular.z;

    auto command_motor_left =
        (forward_speed - angular_speed * HALF_DISTANCE_BETWEEN_WHEELS) / WHEEL_RADIUS;
    auto command_motor_right =
        (forward_speed + angular_speed * HALF_DISTANCE_BETWEEN_WHEELS) / WHEEL_RADIUS;

    wb_motor_set_velocity(left_motor, command_motor_left);
    wb_motor_set_velocity(right_motor, command_motor_right);
}
```

The final lines of the file define the end of the my_robot_driver namespace and include a macro to export the MyRobotDriver class as a plugin using the PLUGINLIB_EXPORT_CLASS macro. This allows the plugin to be loaded by the Webots ROS2 driver at runtime.

```cpp
#include "pluginlib/class_list_macros.hpp"
PLUGINLIB_EXPORT_CLASS(my_robot_driver::MyRobotDriver, webots_ros2_driver::PluginInterface)
```

**Note:** While the plugin is implemented in C++, the C API must be used to interact with the Webots controller library.

### 4 Create the my_robot.urdf file

You now have to create a URDF file to declare the MyRobotDriver plugin. This will allow the webots_ros2_driver ROS node to launch the plugin and connect it to the target robot.

In the my_package/resource folder create a text file named my_robot.urdf with this content:

```xml
<?xml version="1.0" ?>
<robot name="My robot">
    <webots>
        <plugin type="my_package.my_robot_driver.MyRobotDriver" />
    </webots>
</robot>
```

The `type` attribute specifies the path to the class given by the hierarchical structure of files. webots_ros2_driver is responsible for loading the class based on the specified package and modules.

```xml
<?xml version="1.0" ?>
<robot name="My robot">
    <webots>

(continues on next page)
The `<plugin>` tag is used to load a plugin from a specified namespace and class name. The `type` attribute specifies the namespace and class name to load. The `<webots>` tag is used to include the plugin in the Webots simulation. The `<robot>` tag is used to define the robot in the URDF file.

### Note
This simple URDF file doesn't contain any link or joint information about the robot as it is not needed in this tutorial. However, URDF files usually contain much more information as explained in the URDF tutorial.

### Note
Here the plugin does not take any input parameter, but this can be achieved with a tag containing the parameter name.

#### Python
```xml
<plugin type="my_package.my_robot_driver.MyRobotDriver">
  <parameterName>someValue</parameterName>
</plugin>
```

#### C++
```xml
<plugin type="my_robot_driver::MyRobotDriver">
  <parameterName>someValue</parameterName>
</plugin>
```

This is namely used to pass parameters to existing Webots device plugins (see Setting up a robot simulation (Advanced)).

### 5 Create the launch file

Let's create the launch file to easily launch the simulation and the ROS controller with a single command. In the `my_package/launch` folder create a new text file named `robot_launch.py` with this code:

```python
import os
import launch
from launch import LaunchDescription
from ament_index_python.packages import get_package_share_directory
from webots_ros2_driver.webots_launcher import WebotsLauncher
from webots_ros2_driver.webots_controller import WebotsController

def generate_launch_description():
    package_dir = get_package_share_directory('my_package')
    robot_description_path = os.path.join(package_dir, 'resource', 'my_robot.urdf')

    webots = WebotsLauncher(
        world=os.path.join(package_dir, 'worlds', 'my_world.wbt')
    )

    my_robot_driver = WebotsController(
```
robot_name='my_robot',
parameters=[
    {'robot_description': robot_description_path},
]
)

return LaunchDescription([
    webots,
    my_robot_driver,
    launch.actions.RegisterEventHandler(
        event_handler=launch.event_handlers.OnProcessExit(
            target_action=webots,
            on_exit=[launch.actions.EmitEvent(event=launch.events.Shutdown())],
        )
    )
])

The WebotsLauncher object is a custom action that allows you to start a Webots simulation instance. You have to specify in the constructor which world file the simulator will open.

```python
webots = WebotsLauncher(
    world=os.path.join(package_dir, 'worlds', 'my_world.wbt')
)
```

Then, the ROS node interacting with the simulated robot is created. This node, named WebotsController, is located in the webots_ros2_driver package.

Linux

The node will be able to communicate with the simulated robot by using a custom protocol based on IPC and shared memory.

Windows

The node (in WSL) will be able to communicate with the simulated robot (in Webots on native Windows) through a TCP connection.

macOS

The node (in the docker container) will be able to communicate with the simulated robot (in Webots on native macOS) through a TCP connection.

In your case, you need to run a single instance of this node, because you have a single robot in the simulation. But if you had more robots in the simulation, you would have to run one instance of this node per robot. The robot_name parameter is used to define the name of the robot the driver should connect to. The robot_description parameter holds the path to the URDF file which refers to the MyRobotDriver plugin. You can see the WebotsController node as the interface that connects your controller plugin to the target robot.

```python
my_robot_driver = WebotsController(
    robot_name='my_robot',
    parameters=[
        {'robot_description': robot_description_path},
    ]
)
```

After that, the two nodes are set to be launched in the LaunchDescription constructor:
Finally, an optional part is added in order to shutdown all the nodes once Webots terminates (e.g., when it gets closed from the graphical user interface).

```
launch.actions.RegisterEventHandler(
    event_handler=launch.event_handlers.OnProcessExit(
        target_action=webots,
        on_exit=[launch.actions.EmitEvent(event=launch.events.Shutdown())],
    )
)
```

**Note:** More details on `WebotsController` and `WebotsLauncher` arguments can be found on the nodes reference page.

## 6 Edit additional files

**Python**

Before you can start the launch file, you have to modify the `setup.py` file to include the extra files you added. Open `my_package/setup.py` and replace its contents with:

```
from setuptools import find_packages, setup

package_name = 'my_package'
data_files = []
data_files.append(('share/ament_index/resource_index/packages', ['resource/' + package_name]))
data_files.append(('share/' + package_name + '/launch', ['launch/robot_launch.py']))
data_files.append(('share/' + package_name + '/worlds', ['worlds/my_world.wbt']))
data_files.append(('share/' + package_name + '/resource', ['resource/my_robot.urdf']))
data_files.append(('share/' + package_name, ['package.xml']))

setup(
    name=package_name,
    version='0.0.0',
    packages=find_packages(exclude=['test']),
    data_files=data_files,
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='user',
    maintainer_email='user.name@mail.com',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={'console_scripts': [
        'my_robot_driver = my_package.my_robot_driver:main',
    ]}
)
```
This sets up the package and adds in the `data_files` variable the newly added files: `my_world.wbt`, `my_robot.urdf` and `robot_launch.py`.

**C++**

Before you can start the launch file, you have to modify `CMakeLists.txt` and `my_robot_driver.xml` files:

- **CMakeLists.txt** defines the compilation rules of your plugin.
- **my_robot_driver.xml** is necessary for the pluginlib to find your Webots ROS 2 plugin.

Open `my_package/my_robot_driver.xml` and replace its contents with:

```xml
<library path="my_package">
  <!-- The 'type' attribute is a reference to the plugin class. -->
  <!-- The 'base_class_type' attribute is always 'webots_ros2_driver::PluginInterface'. -->
  <class type="my_robot_driver::MyRobotDriver" base_class_type="webots_ros2_driver::PluginInterface">
    <description>
      This is a Webots ROS 2 plugin example
    </description>
  </class>
</library>
```

Open `my_package/CMakeLists.txt` and replace its contents with:

```cmake
# Besides the package specific dependencies we also need the 'pluginlib' and 'webots_ros2_driver'
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)
find_package(geometry_msgs REQUIRED)
find_package(pluginlib REQUIRED)
find_package(webots_ros2_driver REQUIRED)

# Export the plugin configuration file
pluginlib_export_plugin_description_file(webots_ros2_driver my_robot_driver.xml)

# MyRobotDriver library
add_library(
  ${PROJECT_NAME}
  SHARED
  src/MyRobotDriver.cpp
)
```

(continues on next page)
target_include_directories(
    ${PROJECT_NAME}
    PRIVATE
    include
)
ament_target_dependencies(
    ${PROJECT_NAME}
    pluginlib
    rclcpp
    webots_ros2_driver
)
install(TARGETS
    ${PROJECT_NAME}
    ARCHIVE DESTINATION lib
    LIBRARY DESTINATION lib
    RUNTIME DESTINATION bin
)
# Install additional directories.
install(DIRECTORY
    launch
    resource
    worlds
    DESTINATION share/${PROJECT_NAME}/
)
ament_export_include_directories(
    include
)
ament_export_libraries(
    ${PROJECT_NAME}
)
ament_package()

The CMakeLists.txt exports the plugin configuration file with the pluginlib_export_plugin_description_file(), defines a shared library of the C++ plugin src/MyRobotDriver.cpp, and sets the include and library dependencies using ament_target_dependencies().

The file then installs the library, the directories launch, resource, and worlds to the share/my_package directory. Finally, it exports the include directories and libraries using ament_export_include_directories() and ament_export_libraries(), respectively, and declares the package using ament_package().

7 Test the code

Linux

From a terminal in your ROS 2 workspace run:

```
colcon build
source install/local_setup.bash
ros2 launch my_package robot_launch.py
```

This will launch the simulation. Webots will be automatically installed on the first run in case it was not already installed.
Windows

From a terminal in your WSL ROS 2 workspace run:

```
colcon build
export WEBOTS_HOME=/mnt/c/Program\ Files/Webots
source install/local_setup.bash
ros2 launch my_package robot_launch.py
```

Be sure to use the `/mnt` prefix in front of your path to the Webots installation folder to access the Windows file system from WSL.

This will launch the simulation. Webots will be automatically installed on the first run in case it was not already installed.

macOS

On macOS, a local server must be started on the host to start Webots from the VM. The local server can be downloaded on the webots-server repository.

In a terminal of the host machine (not in the VM), specify the Webots installation folder (e.g. `/Applications/Webots.app`) and start the server using the following commands:

```
export WEBOTS_HOME=/Applications/Webots.app
python3 local_simulation_server.py
```

From a terminal in the Linux VM in your ROS 2 workspace, build and launch your custom package with:

```
colcon build
source install/local_setup.bash
ros2 launch my_package robot_launch.py
```

**Note:** If you want to install Webots manually, you can download it [here](#).

Then, open a second terminal and send a command with:

```
ros2 topic pub /cmd_vel geometry_msgs/Twist "linear: { x: 0.1 }"
```

The robot is now moving forward.
At this point, the robot is able to blindly follow your motor commands. But it will eventually bump into the wall as you order it to move forwards.

Close the Webots window, this should also shutdown your ROS nodes started from the launcher. Close also the topic command with Ctrl+C in the second terminal.
Summary

In this tutorial, you set-up a realistic robot simulation with Webots and implemented a custom plugin to control the motors of the robot.

Next steps

To improve the simulation, the robot’s sensors can be used to detect obstacles and avoid them. The second part of the tutorial shows how to implement such behaviour:

- Setting up a robot simulation (Advanced).

Setting up a robot simulation (Advanced)

Goal: Extend a robot simulation with an obstacle avoider node.

Tutorial level: Advanced

Time: 20 minutes

Contents

- Background
- Prerequisites
- Tasks
  - 1 Updating my_robot.urdf
  - 2 Creating a ROS node to avoid obstacles
  - 3 Updating additional files
  - 4 Test the obstacle avoidance code
- Summary
- Next steps

Background

In this tutorial you will extend the package created in the first part of the tutorial: Setting up a robot simulation (Basic). The aim is to implement a ROS 2 node that avoids obstacles using the robot’s distance sensors. This tutorial focuses on using robot devices with the webots_ros2_driver interface.
Prerequisites

This is a continuation of the first part of the tutorial: *Setting up a robot simulation (Basic)*. It is mandatory to start with the first part to set up the custom packages and necessary files.

This tutorial is compatible with version 2023.1.0 of `webots_ros2` and Webots R2023b, as well as upcoming versions.

Tasks

1 Updating `my_robot.urdf`

As mentioned in *Setting up a robot simulation (Basic)*, `webots_ros2_driver` contains plugins to interface most of Webots devices with ROS 2 directly. These plugins can be loaded using the `<device>` tag in the URDF file of the robot. The `reference` attribute should match the Webots device name parameter. The list of all existing interfaces and the corresponding parameters can be found on the devices reference page. For available devices that are not configured in the URDF file, the interface will be automatically created and default values will be used for ROS parameters (e.g. update rate, topic name, and frame name).

In `my_robot.urdf` replace the whole contents with:

**Python**

```xml
<?xml version="1.0" ?>
<robot name="My robot">
  <webots>
    <device reference="ds0" type="DistanceSensor">
      <ros>
        <topicName>/left_sensor</topicName>
        <alwaysOn>true</alwaysOn>
      </ros>
    </device>
    <device reference="ds1" type="DistanceSensor">
      <ros>
        <topicName>/right_sensor</topicName>
        <alwaysOn>true</alwaysOn>
      </ros>
    </device>
    <plugin type="my_package.my_robot_driver.MyRobotDriver" />
  </webots>
</robot>
```

**C++**

```xml
<?xml version="1.0" ?>
<robot name="My robot">
  <webots>
    <device reference="ds0" type="DistanceSensor">
      <ros>
        <topicName>/left_sensor</topicName>
        <alwaysOn>true</alwaysOn>
      </ros>
    </device>
    <device reference="ds1" type="DistanceSensor">
      <ros>
        <topicName>/right_sensor</topicName>
        <alwaysOn>true</alwaysOn>
      </ros>
    </device>
    <plugin type="my_package.my_robot_driver.MyRobotDriver" />
  </webots>
</robot>
```
In addition to your custom plugin, the webots_ros2_driver will parse the <device> tags referring to the Distance-Sensor nodes and use the standard parameters in the <ros> tags to enable the sensors and name their topics.

2 Creating a ROS node to avoid obstacles

Python

The robot will use a standard ROS node to detect the wall and send motor commands to avoid it. In the my_package/my_package/ folder, create a file named obstacle_avoider.py with this code:

```python
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Range
from geometry_msgs.msg import Twist

MAX_RANGE = 0.15

class ObstacleAvoider(Node):
    def __init__(self):
        super().__init__('obstacle_avoider')

        self.__publisher = self.create_publisher(Twist, 'cmd_vel', 1)

        self.create_subscription(Range, 'left_sensor', self.__left_sensor_callback, 1)
        self.create_subscription(Range, 'right_sensor', self.__right_sensor_callback, 1)

    def __left_sensor_callback(self, message):
        self.__left_sensor_value = message.range

    def __right_sensor_callback(self, message):
        self.__right_sensor_value = message.range

        command_message = Twist()

        command_message.linear.x = 0.1

        if self.__left_sensor_value < 0.9 * MAX_RANGE or self.__right_sensor_value < 0.9 * MAX_RANGE:
            command_message.angular.z = -2.0

        self.__publisher.publish(command_message)
```

(continues on next page)
def main(args=None):
    rclpy.init(args=args)
    avoider = ObstacleAvoider()
    rclpy.spin(avoider)
    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    avoider.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()

This node will create a publisher for the command and subscribe to the sensors topics here:

```python
self.__publisher = self.create_publisher(Twist, 'cmd_vel', 1)

self.create_subscription(Range, 'left_sensor', self.__left_sensor_callback, 1)
self.create_subscription(Range, 'right_sensor', self.__right_sensor_callback, 1)
```

When a measurement is received from the left sensor it will be copied to a member field:

```python
def __left_sensor_callback(self, message):
    self.__left_sensor_value = message.range
```

Finally, a message will be sent to the /cmd_vel topic when a measurement from the right sensor is received. The `command_message` will register at least a forward speed in `linear.x` in order to make the robot move when no obstacle is detected. If any of the two sensors detect an obstacle, `command_message` will also register a rotational speed in `angular.z` in order to make the robot turn right.

```python
def __right_sensor_callback(self, message):
    self.__right_sensor_value = message.range

    command_message = Twist()
    command_message.linear.x = 0.1

    if self.__left_sensor_value < 0.9 * MAX_RANGE or self.__right_sensor_value < 0.9 * MAX_RANGE:
        command_message.angular.z = -2.0

    self.__publisher.publish(command_message)
```

C++

The robot will use a standard ROS node to detect the wall and send motor commands to avoid it. In the `my_package/include/my_package` folder, create a header file named `ObstacleAvoider.hpp` with this code:

```cpp
#include <memory>
```
#include "geometry_msgs/msg/twist.hpp"
#include "rclcpp/rclcpp.hpp"
#include "sensor_msgs/msg/range.hpp"

class ObstacleAvoider : public rclcpp::Node {
public:
    explicit ObstacleAvoider();
private:
    void leftSensorCallback(const sensor_msgs::msg::Range::SharedPtr msg);
    void rightSensorCallback(const sensor_msgs::msg::Range::SharedPtr msg);

    rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr publisher_{};
    rclcpp::Subscription<sensor_msgs::msg::Range>::SharedPtr left_sensor_sub_{};
    rclcpp::Subscription<sensor_msgs::msg::Range>::SharedPtr right_sensor_sub_{};

    double left_sensor_value{0.0};
    double right_sensor_value{0.0};
};

In the my_package/src folder, create a source file named ObstacleAvoider.cpp with this code:

#include "my_package/ObstacleAvoider.hpp"

#define MAX_RANGE 0.15

ObstacleAvoider::ObstacleAvoider() : Node("obstacle_avoider") {
    publisher_ = create_publisher<geometry_msgs::msg::Twist>("/cmd_vel", 1);

    left_sensor_sub_ = create_subscription<sensor_msgs::msg::Range>(
        "/left_sensor", 1,
        std::bind(&ObstacleAvoider::leftSensorCallback, this,
            std::placeholders::_1));

    right_sensor_sub_ = create_subscription<sensor_msgs::msg::Range>(
        "/right_sensor", 1,
        std::bind(&ObstacleAvoider::rightSensorCallback, this,
            std::placeholders::_1));
}

void ObstacleAvoider::leftSensorCallback(
    const sensor_msgs::msg::Range::SharedPtr msg) {
    left_sensor_value = msg->range;
}

void ObstacleAvoider::rightSensorCallback(
    const sensor_msgs::msg::Range::SharedPtr msg) {
    right_sensor_value = msg->range;

    auto command_message = std::make_unique<geometry_msgs::msg::Twist>();
    command_message->linear.x = 0.1;
if (left_sensor_value < 0.9 * MAX_RANGE ||
     right_sensor_value < 0.9 * MAX_RANGE) {
    command_message->angular.z = -2.0;
}

publisher_->publish(std::move(command_message));
}

int main(int argc, char *argv[]) {
    rclcpp::init(argc, argv);
    auto avoider = std::make_shared<ObstacleAvoider>();
    rclcpp::spin(avoider);
    rclcpp::shutdown();
    return 0;
}

This node will create a publisher for the command and subscribe to the sensors topics here:

publisher_ = create_publisher<geometry_msgs::msg::Twist>("/cmd_vel", 1);

left_sensor_sub_ = create_subscription<sensor_msgs::msg::Range>(
    "/left_sensor", 1,
    std::bind(&ObstacleAvoider::leftSensorCallback, this,
              std::placeholders::_1));

right_sensor_sub_ = create_subscription<sensor_msgs::msg::Range>(
    "/right_sensor", 1,
    std::bind(&ObstacleAvoider::rightSensorCallback, this,
              std::placeholders::_1));

When a measurement is received from the left sensor it will be copied to a member field:

void ObstacleAvoider::leftSensorCallback(
    const sensor_msgs::msg::Range::SharedPtr msg) {
    left_sensor_value = msg->range;
}

Finally, a message will be sent to the /cmd_vel topic when a measurement from the right sensor is received. The command_message will register at least a forward speed in linear.x in order to make the robot move when no obstacle is detected. If any of the two sensors detect an obstacle, command_message will also register a rotational speed in angular.z in order to make the robot turn right.

void ObstacleAvoider::rightSensorCallback(
    const sensor_msgs::msg::Range::SharedPtr msg) {
    right_sensor_value = msg->range;

    auto command_message = std::make_unique<geometry_msgs::msg::Twist>();

    command_message->linear.x = 0.1;

    if (left_sensor_value < 0.9 * MAX_RANGE ||
        right_sensor_value < 0.9 * MAX_RANGE) {
(continues on next page)
command_message->angular.z = -2.0;
}
publisher_->publish(std::move(command_message));
}

3 Updating additional files

You have to modify these two other files to launch your new node.

Python

Edit setup.py and replace 'console_scripts' with:

```python
'console_scripts': [
    'my_robot_driver = my_package.my_robot_driver:main',
    'obstacle_avoider = my_package.obstacle_avoider:main'
],
```

This will add an entry point for the obstacle_avoider node.

C++

Edit CMakeLists.txt and add the compilation and installation of the obstacle_avoider:

```bash
cmake_minimum_required(VERSION 3.5)
project(my_package)

if(NOT CMAKE_CXX_STANDARD)
    set(CMAKE_CXX_STANDARD 14)
endif()

# Besides the package specific dependencies we also need the `pluginlib` and `webots_ros2_driver`
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)
find_package(geometry_msgs REQUIRED)
find_package(pluginlib REQUIRED)
find_package(webots_ros2_driver REQUIRED)

# Export the plugin configuration file
pluginlib_export_plugin_description_file(webots_ros2_driver my_robot_driver.xml)

# Obstacle avoider
include_directories(
    include
)
add_executable(obstacle_avoider
    src/ObstacleAvoider.cpp
)
ament_target_dependencies(obstacle_avoider
    rclcpp

dependencies(rclcpp
    pluginlib
)
Go to the file `robot_launch.py` and replace `def generate_launch_description():` with:
```python
def generate_launch_description():
    package_dir = get_package_share_directory('my_package')
    robot_description_path = os.path.join(package_dir, 'resource', 'my_robot.urdf')

    webots = WebotsLauncher(
        world=os.path.join(package_dir, 'worlds', 'my_world.wbt')
    )

    my_robot_driver = WebotsController(
        robot_name='my_robot',
        parameters=[
            {'robot_description': robot_description_path},
        ]
    )

    obstacle_avoider = Node(
        package='my_package',
        executable='obstacle_avoider',
    )

    return LaunchDescription([
        webots,
        my_robot_driver,
        obstacle_avoider,
        launch.actions.RegisterEventHandler(
            event_handler=launch.event_handlers.OnProcessExit(
                target_action=webots,
                on_exit=[launch.actions.EmitEvent(event=launch.events.Shutdown())],
            )
        )
    ])
```

This will create an obstacle_avoider node that will be included in the LaunchDescription.

4 Test the obstacle avoidance code

Launch the simulation from a terminal in your ROS 2 workspace:

Linux

From a terminal in your ROS 2 workspace run:

```
colcon build
source install/local_setup.bash
ros2 launch my_package robot_launch.py
```

Windows

From a terminal in your WSL ROS 2 workspace run:

```
colcon build
export WEBOTS_HOME=/mnt/c/Program\ Files/Webots
source install/local_setup.bash
ros2 launch my_package robot_launch.py
```
Be sure to use the `/mnt` prefix in front of your path to the Webots installation folder to access the Windows file system from WSL.

**macOS**

In a terminal of the host machine (not in the VM), if not done already, specify the Webots installation folder (e.g. `/Applications/Webots.app`) and start the server using the following commands:

```
export WEBOTS_HOME=/Applications/Webots.app
python3 local_simulation_server.py
```

Note that the server keeps running once the ROS 2 nodes are ended. You don’t need to restart it every time you want to launch a new simulation. From a terminal in the Linux VM in your ROS 2 workspace, build and launch your custom package with:

```
 cd ~/ros2_ws
 colcon build
 source install/local_setup.bash
 ros2 launch my_package robot_launch.py
```

Your robot should go forward and before hitting the wall it should turn clockwise. You can press **Ctrl+F10** in Webots or go to the **View** menu, **Optional Rendering** and **Show DistanceSensor Rays** to display the range of the distance sensors of the robot.
Summary

In this tutorial, you extended the basic simulation with an obstacle avoider ROS 2 node that publishes velocity commands based on the distance sensor values of the robot.

Next steps

You might want to improve the plugin or create new nodes to change the behavior of the robot. You can also implement a reset handler to automatically restart your ROS nodes when the simulation is reset from the Webots interface:

- Setting up a Reset Handler.

Setting up a Reset Handler

Goal: Extend a robot simulation with a reset handler to restart nodes when the reset button of Webots is pressed.

Tutorial level: Advanced

Time: 10 minutes

Contents

- Background
- Prerequisites
- Reset Handler for Simple Cases (Controllers Only)
- Reset Handler for Multiple Nodes (No Shutdown Required)
- Reset Handler Requiring Node Shutdown
- Summary

Background

In this tutorial, you will learn how to implement a reset handler in a robot simulation using Webots. The Webots reset button reverts the world to the initial state and restarts controllers. It is convenient as it quickly resets the simulation, but in the context of ROS 2, robot controllers are not started again making the simulation stop. The reset handler allows you to restart specific nodes or perform additional actions when the reset button in Webots is pressed. This can be useful for scenarios where you need to reset the state of your simulation or restart specific components without completely restarting the complete ROS system.
Prerequisites

Before proceeding with this tutorial, make sure you have completed the following:

- Understanding of ROS 2 nodes and topics covered in the beginner Tutorials.
- Knowledge of Webots and ROS 2 and its interface package.
- Familiarity with Setting up a robot simulation (Basic).

Reset Handler for Simple Cases (Controllers Only)

In the launch file of your package, add the respawn parameter.

```python
def generate_launch_description():
    robot_driver = WebotsController(
        robot_name='my_robot',
        parameters=[
            {'robot_description': robot_description_path}
        ],
        # Every time one resets the simulation the controller is automatically respawned
        respawn=True
    )

    # Starts Webots
    webots = WebotsLauncher(world=PathJoinSubstitution([package_dir, 'worlds', world]))

    return LaunchDescription([
        webots,
        robot_driver
    ])
```

On reset, Webots kills all driver nodes. Therefore, to start them again after reset, you should set the respawn property of the driver node to True. It will ensure driver nodes are up and running after the reset.

Reset Handler for Multiple Nodes (No Shutdown Required)

If you have some other nodes that have to be started along with the driver node (e.g. ros2_control nodes), then you can use the OnProcessExit event handler:

```python
def get_ros2_control_spawners(*args):
    # Declare here all nodes that must be restarted at simulation reset
    ros_control_node = Node(
        package='controller_manager',
        executable='spawner',
        arguments=['diffdrive_controller']
    )
    return [ros_control_node]

def generate_launch_description():
    # (continues on next page)
```
robot_driver = WebotsController(
    robot_name='my_robot',
    parameters=[
        {'robot_description': robot_description_path}
    ],

    # Every time one resets the simulation the controller is automatically respawned
    respawn=True
)

# Starts Webots
webots = WebotsLauncher(world=PathJoinSubstitution([package_dir, 'worlds', world]))

# Declare the reset handler that respawns nodes when robot_driver exits
reset_handler = launch.actions/RegisterEventHandler(
    event_handler=launch.event_handlers.OnProcessExit(
        target_action=robot_driver,
        on_exit=get_ros2_control_spawners,
    )
)

return LaunchDescription([webots, robot_driver, reset_handler] + get_ros2_control_spawners())

It is not possible to use the respawn property on the ros2_control node, as the spawner exits during launch time and not when the simulation is reset. Instead we should declare a list of nodes in a function (e.g. get_ros2_control_spawners). The nodes of this list are started along other nodes when executing the launch file. With the reset_handler, the function is also declared as action to start when the robot_driver node exits, which corresponds to the moment when the simulation is reset in the Webots interface. The robot_driver node still has the respawn property set to True, so that it gets restarted along with ros2_control nodes.

**Reset Handler Requiring Node Shutdown**

With the current ROS 2 launch API, there is no way to make the reset work in launch files where nodes need to be shutdown before the restart (e.g. Nav2 or RViz). The reason is that currently, ROS 2 doesn't allow to shutdown specific nodes from a launch file. There is a solution, but it requires to manually restart nodes after pushing the reset button.

Webots needs to be started in a specific launch file without other nodes.

def generate_launch_description():
    # Starts Webots
    webots = WebotsLauncher(world=PathJoinSubstitution([package_dir, 'worlds', world]))

    return LaunchDescription([webots])

A second launch file must be started from another process. This launch file contains all other nodes, including robot controllers/plugins, Navigation2 nodes, RViz, state publishers, etc.
The second launch file contains a handler that triggers a shutdown event when the driver node exits (which is the case when the simulation is reset). This second launch file must be manually restarted from the command line after pressing the reset button.
Summary

In this tutorial, you learned how to implement a reset handler in a robot simulation using Webots. The reset handler allows you to restart specific nodes or perform additional actions when the reset button in Webots is pressed. You explored different approaches based on the complexity of your simulation and the requirements of your nodes.

Gazebo

This set of tutorials will teach you how to configure the Gazebo simulator with ROS 2.

Setting up a robot simulation (Gazebo)

Goal: Launch a Simulation with Gazebo and ROS 2

Tutorial level: Advanced

Time: 20 minutes

Contents

- Prerequisites
- Tasks
  - 1 Launch the simulation
  - 2 Configuring ROS 2
  - 3 Visualizing lidar data in ROS 2
- Summary

Prerequisites

First of all you should install ROS 2 and Gazebo. You have two options:

- Install from deb packages. To check which versions are available from deb packages please check this table.
- Compile from sources:
  - ROS 2 install instructions
  - Gazebo install instructions

Tasks

1 Launch the simulation

In this demo you are going to simulate a simple diff drive robot in Gazebo. You are going to use one of the worlds defined in the Gazebo examples called visualize_lidar.sdf. To run this example you should execute the following command in a terminal:

Linux
When the simulation is running you can check the topics provided by Gazebo with the `ign` command line tool:

**Linux**

```
ign topic -l
```

Which should show:

```
/clock
gazebo/resource_paths
gui/camera/pose
gui/record_video/stats
/model/vehicle_blue/odometry
/model/vehicle_blue/tf
/stats
/world/visualize_lidar_world/clock
/world/visualize_lidar_world/dynamic_pose/info
```
Since you have not launched a ROS 2 nodes yet, the output from `ros2 topic list` should be free of any robot topics:

Linux

```
ros2 topic list
```

Which should show:

```
/parameter_events
/rosout
```

## 2 Configuring ROS 2

To be able to communicate our simulation with ROS 2 you need to use a package called `ros_gz_bridge`. This package provides a network bridge which enables the exchange of messages between ROS 2 and Gazebo Transport. You can install this package by typing:

Linux

```
sudo apt-get install ros-iron-ros-ign-bridge
```

At this point you are ready to launch a bridge from ROS to Gazebo. In particular you are going to create a bridge for the topic `/model/vehicle_blue/cmd_vel`:

Linux

```
source /opt/ros/iron/setup.bash
ros2 run ros_gz_bridge parameter_bridge /model/vehicle_blue/cmd_vel@geometry_msgs/msg/[Twist]ignition.msgs.Twist
```

For more details about the `ros_gz_bridge` please check this [README](#).

Once the bridge is running the robot is able to follow your motor commands. There are two options:

- Send a command to the topic using `ros2 topic pub`

  Linux

  ```
  ros2 topic pub /model/vehicle_blue/cmd_vel geometry_msgs/Twist "linear: { x: 0. -1 }
  ```

- `teleop_twist_keyboard` package. This node takes keypresses from the keyboard and publishes them as Twist messages. You can install it typing:

  Linux

  ```
  sudo apt-get install ros-iron-teleop-twist-keyboard
  ```
The default topic where `teleop_twist_keyboard` is publishing Twist messages is `/cmd_vel` but you can remap this topic to make use of the topic used in the bridge:

```
source /opt/ros/iron/setup.bash
ros2 run teleop_twist_keyboard teleop_twist_keyboard --ros-args -r /cmd_vel:=/model/vehicle_blue/cmd_vel
```

Which will show:

```
This node takes keypresses from the keyboard and publishes them as Twist messages. It works best with a US keyboard layout.

---------------------------
Moving around:
   u   i   o
   j   k   l
   m   ,   .

For Holonomic mode (strafing), hold down the shift key:

---------------------------
   U   I   O
   J   K   L
   M   <   >

t : up (+z)
b : down (-z)

anything else : stop

q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%

CTRL-C to quit

currently: speed 0.5 turn 1.0
```

### 3 Visualizing lidar data in ROS 2

The diff drive robot has a lidar. To send the data generated by Gazebo to ROS 2, you need to launch another bridge. In the case the data from the lidar is provided in the Gazebo Transport topic `/lidar2`, which you are going to remap in the bridge. This topic will be available under the topic `/lidar_scan`:

```
source /opt/ros/iron/setup.bash
ros2 run ros_gz_bridge parameter_bridge /lidar2@sensor_msgs/msg/LaserScan[ignition.msgs.LaserScan --ros-args -r /lidar2:=/laser_scan
```

To visualize the data from the lidar in ROS 2 you can use Rviz2:

```
Linux
```
source /opt/ros/iron/setup.bash
rviz2

Then you need to configure the fixed frame:

<table>
<thead>
<tr>
<th>Global Options</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed Frame</td>
<td>vehicle_blue/lidar_link/gpu_lidar</td>
</tr>
<tr>
<td>Background Color</td>
<td>48; 48; 48</td>
</tr>
<tr>
<td>Frame Rate</td>
<td>30</td>
</tr>
</tbody>
</table>

And then click in the button “Add” to include a display to visualize the lidar:
Now you should see the data from the lidar in Rviz2:
Summary

In this tutorial, you launched a robot simulation with Gazebo, launched bridges with actuators and sensors, visualized data from a sensor, and moved a diff drive robot.

Security

Setting up security

Goal: Set up security with sros2.

Tutorial level: Advanced

Time: 15 minutes

Contents

- Background
- Installation
  - Installing from source
  - Selecting an alternate middleware
- Run the demo
  - 1. Create a folder for the security files
  - 2. Generate a keystore
  - 3. Generate keys and certificates
Background

The sros2 package provides the tools and instructions to use ROS2 on top of DDS-Security. The security features have been tested across platforms (Linux, macOS, and Windows) as well as across different languages (C++ and Python). The SROS2 has been designed to work with any secure middleware, although not all middleware is open source and support varies depending on the ROS distribution in use. Please reach out to the ROS 2 Security Working Group if you encounter any support issues.

Installation

Typically security is available following installation using the ROS 2 Installation Guide and the configuration guide. However, if you intend to install from source or switch middleware implementations, consider the following caveats:

Installing from source

Before installing from source, you will need to have a recent version openssl (1.0.2g or later) installed:

Linux

```
sudo apt update
sudo apt install libssl-dev
```

MacOS

```
brew install openssl
```

You will need to have OpenSSL on your library path to run DDS-Security demos. Run the following command, and consider adding to your ~/.bash_profile:

```
export DYLD_LIBRARY_PATH=`brew --prefix openssl`/lib:$DYLD_LIBRARY_PATH
export OPENSSL_ROOT_DIR=`brew --prefix openssl`
```

Windows

If you don't have OpenSSL installed, please follow these instructions

Fast DDS requires an additional CMake flag to build the security plugins, so the colcon invocation needs to be modified to pass:

```
colcon build --symlink-install --cmake-args -DSECURITY=ON
```
Selecting an alternate middleware

If you choose not to use the default middleware implementation, be sure to change your DDS implementation before proceeding.

ROS2 allows you to change the DDS implementation at runtime. See how to work with multiple RMW implementations <../../../How-To-Guides/Working-with-multiple-RMW-implementations> to explore different middleware implementations.

Note that secure communication between vendors is not supported.

Run the demo

1. Create a folder for the security files

   Begin by creating folder to store all the files necessary for this demo:

   Linux
   ```
   mkdir ~/sros2_demo
   ```

   MacOS
   ```
   mkdir ~/sros2_demo
   ```

   Windows
   ```
   md C:\dev\ros2\sros2_demo
   ```

2. Generate a keystore

   Use the sros2 utilities to create the keystore. Files in the keystore will be used to enable security for all the participants in the ROS 2 graph.

   Linux
   ```
   cd ~/sros2_demo
   rosf2 security create_keystore demo_keystore
   ```

   MacOS
   ```
   cd ~/sros2_demo
   rosf2 security create_keystore demo_keystore
   ```

   Windows
   ```
   cd sros2_demo
   rosf2 security create_keystore demo_keystore
   ```
3. Generate keys and certificates

Once the keystore is created, create keys and certificates for each node with security enabled. For our demo, that includes the talker and listener nodes. This command uses the `create_enclave` feature which is covered in more detail in the next tutorial.

**Linux**

```
ros2 security create_enclave demo_keystore /talker_listener/talker
ros2 security create_enclave demo_keystore /talker_listener/listener
```

**MacOS**

```
ros2 security create_enclave demo_keystore /talker_listener/talker
ros2 security create_enclave demo_keystore /talker_listener/listener
```

**Windows**

```
ros2 security create_enclave demo_keystore /talker_listener/talker
ros2 security create_enclave demo_keystore /talker_listener/listener
```

If unable to write 'random state' appears then set the environment variable RANDFILE.

```
set RANDFILE=C:\dev\ros2\sros2_demo\.rnd
```

Then re-run the commands above.

4. Configure environment variables

Three environment variables allow the middleware to locate encryption materials and enable (and possibly enforce) security. These and other security-related environment variables are described in the ROS 2 DDS-Security Integration design document.

**Linux**

```
export ROS_SECURITY_KEYSTORE=~/.sros2_demo/demo_keystore
export ROS_SECURITY_ENABLE=true
export ROS_SECURITY_STRATEGY=Enforce
```

**MacOS**

```
export ROS_SECURITY_KEYSTORE=~/.sros2_demo/demo_keystore
export ROS_SECURITY_ENABLE=true
export ROS_SECURITY_STRATEGY=Enforce
```

**Windows**

```
set ROS_SECURITY_KEYSTORE=%cd%/demo_keystore
set ROS_SECURITY_ENABLE=true
set ROS_SECURITY_STRATEGY=Enforce
```

These variables need to be defined in each terminal used for the demo. For convenience you can add them to your boot environment.
5. Run the talker/listener demo

Begin the demo by launching the talker node.

```bash
ros2 run demo_nodes_cpp talker --ros-args --enclave /talker_listener/talker
```

In another terminal, do the same to launch the listener node. The environment variables in this terminal must be properly set as described in step 4 above.

```bash
ros2 run demo_nodes_py listener --ros-args --enclave /talker_listener/listener
```

These nodes will be communicating using authentication and encryption! If you look at the packet contents (for example, using `tcpdump` or `Wireshark` as covered in another tutorial), you can see that the messages are encrypted.

Note: You can switch between the C++ (`demo_nodes_cpp`) and Python (`demo_nodes_py`) packages arbitrarily.

These nodes are able to communicate because we have created the appropriate keys and certificates for them.

Leave both nodes running as you answer the questions below.

**Take the Quiz!**

**Question 1**
Open another terminal session, but **do not** set the environment variables so that security is not enabled. Start the listener. What do you expect to happen?

**Answer 1**
The listener launches but does not receive any messages. All traffic is encrypted, and without security enabled the listener does not receive anything.

**Question 2**
Stop the listener, set the environment variable `ROS_SECURITY_ENABLE` to `true` and start the listener again. What results do you expect this time?

**Answer 2**
The listener still launches but does not receive messages. Although security has now been enabled, it is not been configured properly since ROS is unable to locate the key files. The listener launches, but in non-secure mode since security is not enforced, which means that although the properly configured talker is sending encrypted messages, this listener is unable to decrypt them.

**Question 3**
Stop the listener and set `ROS_SECURITY_STRATEGY` to `Enforce`. What happens now?

**Answer 3**
The listener fails to launch. Security has been enabled and is being enforced. Since it still is not properly configured, an error is thrown rather than launching in non-secure mode.
Learn More!

Are you ready to go further with ROS Security? Take a look at the Secure Turtlebot2 Demo. You’ll find a functioning and complex implementation of ROS 2 security, ready to try out your own custom scenarios. Be sure to create pull requests and issues here so we can continue improving security support in ROS!

Understanding the security keystore

**Goal:** Explore files located in the ROS 2 security keystore.

**Tutorial level:** Advanced

**Time:** 15 minutes

<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Background</td>
</tr>
<tr>
<td>• Security Artifact Locations</td>
</tr>
<tr>
<td>• Public Key Materials</td>
</tr>
<tr>
<td>• Private Key Materials</td>
</tr>
<tr>
<td>• Domain Governance Policy</td>
</tr>
<tr>
<td>• Security Enclaves</td>
</tr>
<tr>
<td>• Take the quiz!</td>
</tr>
</tbody>
</table>

**Background**

Before proceeding ensure you have completed the Setting up security tutorial.

The sros2 package can be used to create keys, certificates and policies necessary to enable ROS 2 security. However, the security configuration is extremely flexible. A basic understanding of the ROS 2 Security Keystore will allow integration with an existing PKI (Public Key Infrastructure) and managament of sensitive key materials consistent with organizational policies.

**Security Artifact Locations**

With communications security enabled in the prior tutorial, let’s take a look at the files which were created when security was enabled. These are the files which make encryption possible.

The sros2 utilities (ros2 security ...) separate files into public, private and enclave key materials.

ROS uses the directory defined by the environmental variable ROS_SECURITY_KEYSTORE as the keystore. For this tutorial, we use the directory ~/sros2_demo/demo_keystore.
**Public Key Materials**

You will find three encryption certificates in the public directory at `~/sros2_demo/demo_keystore/public`; however, the identity and permissions certificates are actually just a link to the Certificate Authority (CA) certificate.

In a public key infrastructure, the Certificate Authority acts as a trust anchor: it validates the identities and permissions of participants. For ROS, that means all the nodes that participate in the ROS graph (which may extend to an entire fleet of individual robots). By placing the Certificate Authority’s certificate (`ca.cert.pem`) in the proper location on the robot, all ROS nodes can establish mutual trust with other nodes using the same Certificate Authority.

Although in our tutorials we create a Certificate Authority on-the-fly, in a production system this should be done according to a pre-defined security plan. Typically the Certificate Authority for a production system will be created off-line, and placed on the robot during initial setup. It may be unique for each robot, or shared across a fleet of robots all intended to trust each other.

DDS (and ROS, by extension) supports separation of identity and permission trust chains, so each function has its own certificate authority. In most cases a ROS system security plan does not require a separation between these duties, so the security utilities generate a single Certificate Authority which is used for both identity and permissions.

Use `openssl` to view this x509 certificate and display it as text:

```bash
cd ~/sros2_demo/demo_keystore/public
openssl x509 -in ca.cert.pem -text -noout
```

The output should look similar to the following:

```
Certificate:
  Data:
    Version: 3 (0x2)
    Signature Algorithm:  ecdsa-with-SHA256
    Issuer: CN = sros2testCA
    Validity
      Not Before: Jun 1 16:57:37 2021 GMT
      Not After : May 31 16:57:37 2031 GMT
    Subject: CN = sros2testCA
    Subject Public Key Info:
      Public Key Algorithm: id-ecPublicKey
      Public-Key: (256 bit)
        pub:
          dc:ed:39:c5:32
        ASN1 OID: prime256v1
        NIST CURVE: P-256
    X509v3 extensions:
      X509v3 Basic Constraints: critical
        CA:TRUE, pathlen:1
    Signature Algorithm: ecdsa-with-SHA256
      30:45:02:21:00:00:d4:fc:d8:45:ff:a4:51:49:98:4c:f0:c4:3f:
      e0:e7:33:19:8e:31:3c:d0:43:e7:e9:8f:36:f0:90:18:ed:d7:
```
Some things to note about this CA certificate:

• The certificate subject name `sros2testCA` is the default provided by the `sros2` utilities.
• This certificate is valid for ten years from time of creation
• Like all certificates, this contains a public key used for public-private key encryption
• As a Root Certificate Authority, this is a self-signed certificate; i.e., it is signed using its own private key.

Since this is a public certificate, it can be freely copied as needed to establish trust throughout your ROS system.

Private Key Materials

Private key materials can be found in the keystore directory `~/sros2_demo/demo_keystore/private`. Similar to the public directory, this contains one certificate authority key `ca.key.pem` and symbolic links to it to be used as both an Identity and a Permissions CA private key.

**Warning:** Protect this private key and create a secure backup of it!

This is the private key associated with the public Certificate Authority which serves as the anchor for all security in your ROS system. You will use it to modify encryption policies for the ROS graph and to add new ROS participants. Depending upon your robot’s security needs, the key can be protected with access permissions and locked down to another account, or it can be moved off the robot entirely and onto another system or device. If the file is lost, you will be unable to change access permissions and add new participants to the system. Similarly, any user or process with access to the file has the ability to modify system policies and participants.

This file is only required for configuring the robot, but is not needed for the robot to run. It can safely be stored offline in another system or removable media.

The `sros2` utilities use elliptic curve cryptography rather than RSA for improved security and reduced key size. Use the following command to show details about this elliptic curve private key:

```
cd ~/sros2_demo/demo_keystore/private
openssl ec -in ca.key.pem -text -noout
```

Your output should look similar to the following:

```
read EC key
Private-Key: (256 bit)
priv:
   ae:77
pub:
   dc:ed:39:c5:32
ASN1 OID: prime256v1
NIST CURVE: P-256
```

In addition to the private key itself, note that the public key is listed, and it matches the public key listed in the Certificate Authority `ca.cert.pem`. 

4.8. ROS 2 Documentation
Domain Governance Policy

Find the domain governance policy in the enclave directory within the keystore, `~/sros2_demo/demo_keystore/enclaves`. The enclave directory contains XML governance policy document `governance.xml`, as well as a copy of the document which has been signed by the Permissions CA as `governance.p7s`.

The `governance.p7s` file contains domain-wide settings such as how to handle unauthenticated participants, whether to encrypt discovery, and default rules for access to topics.

Use the following command to validate the S/MIME signature of the governance file:

```
openssl smime -verify -in governance.p7s -CAfile ../public/permissions_ca.cert.pem
```

This command will print out the XML document, and the last line will be **Verification successful** to show that the document was properly signed by the Permissions CA.

Security Enclaves

Secure processes (typically ROS nodes) run within a security enclave. In the simplest case, all the processes can be consolidated into the same enclave, and all processes will then use the same security policy. However, to apply different policies to different processes, the processes can use different security enclaves when starting. For more details about security enclaves, see the design document. The security enclave is specified by using the ROS argument `--enclave` when running a node.

Each security enclave requires six files in order to enable security. Each file must be named as defined below, and as outlined in the DDS Security standard. In order to avoid having multiple copies of the same files, the `sros2` utilities create links for each enclave to the single governance policy, the Identity CA and Permissions CA described above.

See the following six files within the `listener` enclave. Three are specific to this enclave, while three are generic to this ROS system:

- `key.pem`, the private key used to encrypt and decrypt within this enclave
- `cert.pem`, the public certificate for this enclave; this certificate has been signed by the Identity CA
- `permissions.p7s`, the permissions for this enclave; this file has been signed with the Permissions CA
- `governance.p7s`, a link to the signed security policy file for this domain
- `identity_ca.cert.pem`, a link to the Identity CA for this domain
- `permissions_ca.cert.pem`, a link to the Permissions CA for this domain

The private encryption key `key.pem` should be protected according to your security plan. This key encrypts, decrypts and validates communications within this specific enclave. Should the key be lost or stolen, revoke the key and create a new identity for this enclave.

The file `permissions.xml` has also been created in this directory and can be used to recreate the signed permissions file. However, this file is not required to enable security since DDS uses the signed version of the file instead.
Take the quiz!

See if you can answer these questions about the ROS security keystore. Begin with a new terminal session and enable security with the keystore created in the prior tutorial:

```bash
export ROS_SECURITY_KEYSTORE=~/.sros2_demo/demo_keys
export ROS_SECURITY_ENABLE=true
export ROS_SECURITY_STRATEGY=Enforce

cd ~/.sros2_demo/demo_keystore/enclaves/talker_listener/listener
```

Make a backup copy of `permissions.p7s` before beginning.

**Question 1**
Open `permissions.p7s` in a text editor. Make a negligible change to the XML content (e.g., add a space or a blank line) and save the file. Launch the listener node:

```bash
ros2 run demo_nodes_cpp listener --ros-args --enclave /talker_listener/listener
```

What do you expect to happen?

Can you launch the talker node?

```bash
ros2 run demo_nodes_cpp talker --ros-args --enclave /talker_listener/talker
```

What is the difference between launching the listener and launching the talker?

**Answer 1**
The listener fails to launch and throws an error. When the `permissions.p7s` file was modified—however minor—the file's signature became invalid. A node will not launch with security enabled and enforced when the permissions file is invalid.

The talker will start as expected. It uses the `permissions.p7s` file in a different enclave, and the file is still valid.

**Question 2**
What command lets you check to see if the signature on the modified `permissions.p7s` file is valid?

**Answer 2**
Check that `permissions.p7s` has been properly signed by the Permissions CA using the `openssl smime` command:

```bash
openssl smime -verify -in permissions.p7s -CAfile permissions_ca.cert.pem
```

Restore your original, properly signed `permissions.p7s` file before proceeding to the next tutorial.

### Ensuring security across machines

**Goal:** Make two different machines communicate securely.

**Tutorial level:** Advanced

**Time:** 5 minutes

---

4.8. ROS 2 Documentation
Background

Before proceeding ensure you have completed the Setting up security tutorial.

The previous tutorials have used two ROS nodes on the same machine sending all network communications over the localhost interface. Let's extend that scenario to involve multiple machines, since the benefits of authentication and encryption then become more obvious.

Suppose that the machine with the keystore created in the previous demo has a hostname Alice, and that we want to also use another machine with hostname Bob for our multi-machine talker/listener demo. We need to move some keys from Alice to Bob to allow SROS 2 to authenticate and encrypt the transmissions.

Create the second keystore

Begin by creating an empty keystore on Bob; the keystore is actually just an empty directory:

Linux

```bash
ssh Bob
mkdir ~/sros2_demo
exit
```

MacOS

```bash
ssh Bob
mkdir ~/sros2_demo
exit
```

Windows

```bash
ssh Bob
md C:\dev\ros2\sros2_demo
exit
```

Copy files

Next copy the keys and certificates for the talker program from Alice to Bob. Since the keys are just text files, we can use scp to copy them.

Linux

```bash
cd ~/sros2_demo/demo_keystore
scp -r talker USERNAME@Bob:~/sros2_demo/demo_keystore
```

MacOS
Windows

```
cd ~/sros2_demo/demo_keystore
scp -r talker USERNAME@Bob:~/sros2_demo/demo_keystore
```

**Warning:** Note that in this case the entire keystore is shared across the different machines which may not be the desired behavior, as it may result in a security risk. Please refer to *Deployment Guidelines* for more information in this regard.

That will be very quick, since it’s just copying some very small text files. Now, we’re ready to run a multi-machine talker/listener demo!

**Launch the nodes**

Once the environment is set up, run the talker on Bob:

```
ros2 run demo_nodes_cpp talker --ros-args --enclave /talker_listener/talker
```

and launch the listener on Alice:

```
ros2 run demo_nodes_py listener --ros-args --enclave /talker_listener/listener
```

Alice will now be receiving encrypted messages from Bob.

With two machines successfully communicating using both encryption and authentication, you can use the same procedure to add more machines to your ROS graph.

**Examining network traffic**

**Goal:** Capture and examine raw ROS 2 network traffic.

**Tutorial level:** Advanced

**Time:** 20 minutes

**Contents**

- *Background*
- *Run the demo*
  - Install tcpdump
  - Start the talker and listener
  - Display unencrypted discovery packets
  - Display unencrypted data packets
  - Enable encryption
Background

ROS 2 communications security is all about protecting communications between nodes. Prior tutorials enabled security, but how can you really tell if traffic is being encrypted? In this tutorial we’ll take a look at capturing live network traffic to show the difference between encrypted and unencrypted traffic.

Note: rmw_fastrtps_cpp uses Shared Memory Transport by default to improve the performance in the transport layer when the endpoints are in the same host system. Security enclaves are still applied, and data will be encrypted. However, you cannot capture live network traffic since the data will not be on the network interface. If you are using rmw_fastrtps_cpp, you need to either go through this tutorial and use a different host system between the publisher and subscriber, or disable shared memory transport with Enabling UDP Transport and How to set Fast-DDS XML configuration.

Run the demo

Install tcpdump

Begin in a new terminal window by installing tcpdump, a command-line tool for capturing and displaying network traffic. Although this tutorial describes tcpdump commands, you can also use Wireshark, a similar graphical tool for capturing and analyzing traffic.

```bash
sudo apt update
sudo apt install tcpdump
```

Run following commands on a single machine through multiple ssh sessions.

Start the talker and listener

Start both the talker and the listener again, each in its own terminal. The security environment variables are not set so security is not enabled for these sessions.

```bash
# Disable ROS Security for both terminals
unset ROS_SECURITY_ENABLE

# In terminal 1:
ros2 run demo_nodes_cpp talker --ros-args --enclave /talker_listener/talker

# In terminal 2:
ros2 run demo_nodes_cpp listener --ros-args --enclave /talker_listener/listener
```
Display unencrypted discovery packets

With the talker and listener running, open another terminal and start `tcpdump` to look at the network traffic. You need to use `sudo` since reading raw network traffic is a privileged operation.

The command below uses the `-X` option to print packet contents, the `-i` option to listen for packets on any interface, and captures only UDP port 7400 traffic.

```bash
sudo tcpdump -X -i any udp port 7400
```

You should see packets like the following:

```
20:18:04.400770 IP 8_xterm.46392 > 239.255.0.1.7400: UDP, length 252
 0x0000: 4500 0118 d4bb b000 0111 7399 c0a8 8007 E.....@.......
 0x0010: eeff 0001 b538 1ce8 0104 31c6 5254 5053 8.........RTPS
 ...0x00c0: 5800 0400 3f0c 3f0c 6200 1c00 1800 0000 X...?.?.b.......
 0x00d0: 2f74 616c 6b65 725f 6c69 7374 656e 6572 /talker_listener
 0x00e0: 2f74 616c 6b65 7200 2c00 2800 2100 0000 /talker.,.(.!...
 0x00f0: 656e 636c 6176 653d 2f74 616c 6b65 725f enclave=/talker_
 0x0100: 6c69 7374 656e 6572 2f74 616c 6b65 723b listener/talker;
 0x0110: 0000 0000 0000 0000 ........
```

This is a discovery datagram—the talker looking for subscribers. As you can see, the node name (`/talker_listener/talker`) and the enclave (also `/talker_listener/talker`) are passed in plain text. You should also see similar discovery datagrams from the listener node. Some other features of a typical discovery packet:

- The destination address is 239.255.0.1, which is a multicast IP address; ROS 2 uses multicast traffic for discovery by default.
- UDP 7400 is the destination port, as per the DDS-RTPS specification.
- The packet contains the “RTPS” tag, also as defined to the DDS-RTPS specification.

Display unencrypted data packets

Use `tcpdump` to capture non-discovery RTPS packets by filtering on UDP ports above 7400:

```bash
sudo tcpdump -i any -X udp portrange 7401-7500
```

You will see few different types of packets, but watch for something like the following which is obviously data being sent from a talker to a listener:

```
20:49:17.927303 IP localhost.46392 > localhost.7415: UDP, length 84
 0x0000: 4500 0070 5b53 4000 4011 e127 7f00 0001 E..p[S@.....
 0x0010: 7f00 0001 b538 1cf7 005c fe6f 5254 5053 8.......
 0x0020: 0203 010f 010f 4874 e752 0000 0100 0000 .....Ht.R.......
 0x0030: 0901 0800 cdee b760 5bf3 5aed 1505 3000 .......
 0x0040: 0000 1000 0000 1204 0000 1203 0000 0000 ...........
 0x0050: 5708 0000 0001 0000 1200 0000 4865 6c6c 做得你好....Hell
 0x0060: 6f20 576f 726c 643a 2032 3133 3500 0000 o.World:.2135...
```

Some features to note about this packet:

- The message contents, “Hello World: 2135”, are sent in clear text.
• The source and destination IP address is localhost: since both nodes are running on the same machine, the nodes discovered each other on the localhost interface

**Enable encryption**

Stop both the talker and the listener nodes. Enable encryption for both by setting the security environment variables and run them again.

```
# In terminal 1:
export ROS_SECURITY_KEYSTORE=~/.sros2_demo/demo_keys
export ROS_SECURITY_ENABLE=true
export ROS_SECURITY_STRATEGY=Enforce
ros2 run demo_nodes_cpp talker --ros-args --enclave /talker_listener/talker

# In terminal 2:
export ROS_SECURITY_KEYSTORE=~/.sros2_demo/demo_keys
export ROS_SECURITY_ENABLE=true
export ROS_SECURITY_STRATEGY=Enforce
ros2 run demo_nodes_cpp listener --ros-args --enclave /talker_listener/listener
```

**Display encrypted discovery packets**

Run the same `tcpdump` command used earlier to examine the output of discovery traffic with encryption enabled:

```
sudo tcpdump -X -i any udp port 7400
```

The typical discovery packet looks somewhat like the following:

```
21:09:07.336617 IP 8_xterm.60409 > 239.255.0.1.7400: UDP, length 596
0x0000: 4500 0270 c2f6 4000 0111 83d6 c0a8 8007 E..p...@........
0x0010: eeff 0001 ebef 1ce8 025c 331e 5254 5053 .........\3.RTPS
0x0020: 0203 010f bbdd 199c 7522 b6cb 699f 74ae ........u"..i.t.
0x00c0: 5800 0400 3f0c ff0f 6200 2000 1a00 0000 X...?...b......
0x00d0: 2f74 616c 6b65 725f 6c69 7374 656e 6572 /talker_listener
0x00e0: 2f6c 6973 7465 6e65 7200 0000 2c00 2800 /listener,...,(.
0x00f0: 2300 0000 655e 636c 6176 653d 2f74 616c 2f6e 7465 6e65 7200 0000 2c00 2800 /listener,...,(. 0x0200: 2f74 616c 6b65 725f 6c69 7374 656e 6572 /talker_listener
```

This packet is much larger and includes information which can be used to set up encryption among ROS nodes. As we will see shortly, this actually includes some of the security configuration files that were created when we enabled security. Interested in learning more? Take a look at the excellent paper *[Network Reconnaissance and Vulnerability Excavation of Secure DDS Systems]* to understand why this matters.
Display encrypted data packets

Now use `tcpdump` to capture data packets:

```
sudo tcpdump -i any -X udp portrange 7401-7500
```

A typical data packet looks like the following:

```
21:18:14.531102 IP localhost.54869 > localhost.7415: UDP, length 328
0x0000: 4500 0164 bb42 4000 4011 8044 7f00 0001 E..d.B@..D....
0x0010: 7f00 0001 d655 1c7f 0150 ff63 5254 5053 .....U...P.cRTPS
0x0020: 0203 010f daf7 10ce d977 449b bb33 f04a ........wD..3.J
0x0030: 3301 1400 0000 0003 492a 6066 8603 cdb5 3.......I"f....
0x0040: 9df6 5da6 8402 2136 0c01 1400 0000 0000 ..]...6........
0x0050: 0203 010f daf7 10ce d977 449b bb33 f04a ........wD..3.J
...
0x0130: 7905 d390 3201 1400 3ae5 0b60 3906 967e y...2.....`9..~
0x0140: 5b17 fd42 de95 54b9 0000 0000 3401 1400 [.B..T......4...
0x0150: 42ae f04d 0559 84c5 7116 1c51 91ba 3799 B..M..Y..q..Q..7.
0x0160: 0000 0000 ....
```

The data in this RTPS packet is all encrypted.

In addition to this data packet, you should see additional packets with node and enclave names; these support other ROS features such as parameters and services. Encryption options for these packets can also be controlled by security policy.

Setting access controls

Goal: Limit the topics a node can use.

Tutorial level: Advanced

Time: 20 minutes

Contents

- Background
  - Modify permissions.xml
  - Sign the policy file
  - Launch the node
  - Use the templates
Background

Before proceeding ensure you have completed the Setting up security tutorial.

Permissions are quite flexible and can be used to control many behaviors within the ROS graph.

For this tutorial, we demonstrate a policy which only allows publishing messages on the default chatter topic. This would prevent, for instance, remapping the topic when launching the listener or using the same security enclaves for another purpose.

In order to enforce this policy, we need to update the permissions.xml file and re-sign it before launching the node. This can be done by modifying the permissions file by hand, or by using XML templates.

Modify permissions.xml

Begin by making a backup of your permissions files, and open permissions.xml for editing:

```bash
cd ~/sros2_demo/demo_keystore/enclaves/talker_listener/talker
mv permissions.p7s permissions.p7s~
mv permissions.xml permissions.xml~
vi permissions.xml
```

We will be modifying the <allow_rule> for <publish> and <subscribe>. The topics in this XML file use the DDS naming format, not the ROS name. Find details on mapping topic names between ROS and DDS in the Topic and Service Names design document.

Paste the following XML content into permission.xml, save the file and exit the text editor. This shows the chatter and rosout ROS topics renamed to the DDS rt/chatter and rt/rosout topics, respectively:

```xml
  <permissions>
    <grant name="/talker_listener/talker">
      <subject_name>CN=/talker_listener/talker</subject_name>
      <validity>
        <not_before>2021-06-01T16:57:53</not_before>
        <not_after>2031-05-31T16:57:53</not_after>
      </validity>
      <allow_rule>
        <domains>
          <id>0</id>
        </domains>
        <publish>
          <topics>
            <topic>rt/chatter</topic>
            <topic>rt/rosout</topic>
            <topic>rt/parameter_events</topic>
            <topic>/*/talker/*</topic>
          </topics>
        </publish>
        <subscribe>
          <topics>
            <topic>rt/parameter_events</topic>
            <topic>/*/talker/*</topic>
          </topics>
        </subscribe>
      </allow_rule>
    </grant>
  </permissions>
</dds>
```

(continues on next page)
This policy allows the talker to publish on the chatter and the rosoout topics. It also allows includes publish and subscribe permissions needed for the talker node to manage parameters (a requirement for all nodes). Discovery permissions remain unchanged from the original template.

**Sign the policy file**

This next command creates the new S/MIME signed policy file permissions.p7s from the updated XML file permissions.xml. The file must be signed with the Permissions CA certificate, which requires access to the Permissions CA private key. If the private key has been protected, additional steps may be required to unlock and use it according to your security plan.

```
openssl smime -sign -text -in permissions.xml -out permissions.p7s \
    --signer permissions_ca.cert.pem \
    -inkey ~/sros2_demo/demo_keystore/private/permissions_ca.key.pem
```

**Launch the node**

With the updated permissions in place, we can launch the node successfully using the same command used in prior tutorials:

```
ros2 run demo_nodes_cpp talker --ros-args --enclave /talker_listener/talker
```

However, attempting to remap the chatter topic prevents the node from launching (note that this requires the ROS_SECURITY_STRATEGY set to Enforce).

```
ros2 run demo_nodes_cpp talker --ros-args --enclave /talker_listener/talker \
    --remap chatter:=not_chatter
```
Use the templates

Security policies can quickly become confusing, so the sros2 utilities add the ability to create policies from templates. Do this by using the sample policy file provided in the sros2 repository. Let’s create a policy for both the talker and the listener to only use the chatter topic.

Begin by downloading the sros2 repository with the sample policy files:

```bash
git clone https://github.com/ros2/sros2.git /tmp/sros2
```

Then use the `create_permission` verb while pointing to the sample policy to generate the XML permission files:

```bash
ros2 security create_permission demo_keystore \
/talker_listener/talker \
/tmp/sros2/sros2/test/policies/sample.policy.xml
ros2 security create_permission demo_keystore \
/talker_listener/listener \
/tmp/sros2/sros2/test/policies/sample.policy.xml
```

These permission files allow nodes to only publish or subscribe to the chatter topic, and enable communications required for parameters.

In one terminal with security enabled as in previous security tutorials, run the talker demo program:

```bash
ros2 run demo_nodes_cpp talker --ros-args -e /talker_listener/talker
```

In another terminal do the same with the listener program:

```bash
ros2 run demo_nodes_py listener --ros-args -e /talker_listener/listener
```

At this point, your talker and listener nodes will be communicating securely using explicit access control lists. However, the following attempt for the listener node to subscribe to a topic other than chatter will fail:

```bash
ros2 run demo_nodes_py listener --ros-args --enclave /talker_listener/listener \ 
--remap chatter:=not_chatter
```

Deployment Guidelines

Goal: Understand the best practices when deploying security artifacts into production systems.

Tutorial level: Advanced

Time: 20 minutes
Background

Typical deployment scenarios often involve shipping containerized applications, or packages, into remote systems. Special attention should be paid when deploying security enabled applications, requiring users to reason about the sensitivity of packaged files.

Complying with the DDS Security standard, the sros2 package provides a collection of utilities for managing security under ROS 2 environments in a highly modular and flexible fashion.

Basic core guidelines on how to organize the different certificates, keys and directories remains a critical factor to avoid compromising the security of the system. This includes protection-awareness and criteria for selecting the minimum set of necessary files to be deployed upon remote production systems for minimizing security exposure.

Prerequisites

- A docker installation with the compose plugin. Please refer to the installation steps detailed in Docker installation and Compose Plugin.
- (Recommended) A basic understanding on ROS 2 Security design.
- (Recommended) Previous security tutorials completion. In particular:
  - Setting up security
  - Understanding the security keystore
  - Setting access controls

General Guidelines

ROS 2 leverages DDS Security extensions to ensure security on message exchanges within the same enclave. The different signed files and certificates within an enclave are generated from the private keys and certificates of a Certificate Authority (CA) trusted entity. In fact, two different CA’s can be selected for identity and permissions, per enclave. Those CA artifacts are stored inside private/ and public/ sub-directories of a Keystore with the following folder structure:

```
keystore
    |__ enclaves
    |   |__ ...
    |   |   |__ ...
    |__ private
    |   |__ ...
    |__ public
```

A good practice for the creation and usage of a certain Certificate Authority on a typical deployment for a production system, is to:

1. Create it within the organization system intended for internal use only.
2. Generate/modify desired enclaves bearing in mind that:
   • Not all the generated enclaves should be deployed to all target devices.
   • A reasonable way to proceed would be having one enclave per application, allowing for a separation of concerns.

3. Ship public/ alongside with corresponding enclaves/ into the different remote production devices during setup.

4. Keep and protect private/ keys and/or certification requests in the organization.

It is important to note that if private/ files are lost, it won’t be possible to change access permissions, add or modify security profiles anymore.

In addition, further practices may be taken into consideration:
   • Granting read-only permissions to the enclaves/ directory contents.
   • If a PKCS#11 compliant URI is given for generating enclave’s private keys, a Hardware Security Module (HSM) could be used to store them.

The following table depicts a summary of the previous statements relating the Keystore directory with the Recommended location:

<table>
<thead>
<tr>
<th>Directory / Location</th>
<th>Organization</th>
<th>Target Device</th>
<th>Material Sensitivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>✓</td>
<td>✓</td>
<td>Low</td>
</tr>
<tr>
<td>private</td>
<td>✓</td>
<td>×</td>
<td>High</td>
</tr>
<tr>
<td>enclaves</td>
<td>✓</td>
<td>✓</td>
<td>Medium</td>
</tr>
</tbody>
</table>

Building a deployment scenario

To illustrate a simple deployment scenario, a new docker image will be built on top of the one provided by ros:<DISTRO>. Starting from the image, three containers will be created with the aim of:
   • Initializing the keystore in a local host’s shared volume.
   • Simulating two deployed remote devices that interact with each other in a secure way.

In this example, the local host serves as the organization’s system. Let us start by creating a workspace folder:

```
mkdir ~/security_gd_tutorial
cd ~/security_gd_tutorial
```

Generating the Docker Image

In order to build a new docker image, a Dockerfile is required. The one proposed for this tutorial can be retrieved with the following command:

```
# Download the Dockerfile
```

Now, build the docker image with the command:

```
# Build the base image
docker build -t ros2_security/deployment_tutorial --build-arg ROS_DISTRO=iron .
```
Understanding the compose file

A compose configuration file takes an image to create containers as services. In this tutorial, three services are defined within the configuration:

- **keystore-creator**: That, similarly to previous tutorials, it internally initializes a new keystore tree directory. This will create `enclaves/public/` and `private/`, which are explained in more detail in ROS 2 Security enclaves. The `keystore` directory is configured to be a shared volume across containers.

- **listener** and **talker**: Act as the remote device actors in this tutorial. Required Security environment variables are sourced as well as the necessary keystore files from the shared volume.

The compose configuration yaml file can be downloaded with:

```
# Download the compose file
```

Running the example

In the same working directory `~/security_gd_tutorial`, run:

```
# Start the example
docker compose -f compose.deployment.yaml up
```

This should result in the following output:

- **tutorial-listener-1**: Found security directory: `/keystore/enclaves/talker_listener/listener`
- **tutorial-talker-1**: Found security directory: `/keystore/enclaves/talker_listener/talker`
- **tutorial-listener-1**: Publishing: 'Hello World: <number>'
- **tutorial-talker-1**: I heard: [Hello World: <number>]

Examining the containers

While having the containers running that simulate the two remote devices for this tutorial, attach to each of them by opening two different terminals and enter:

```
# Terminal 1
docker exec -it tutorial-listener-1 bash
cd keystore
tree

# Terminal 2
docker exec -it tutorial-talker-1 bash
cd keystore
tree
```

A similar output to the one depicted below should be obtained:

```
# Terminal 1
keystore
    ├── enclaves
```

(continues on next page)
Note that:

- *private/* folder is not moved but left in the local host (organization).
- Each one of the deployed devices contain its own minimum enclave required for its application.

**Note:** For the sake of simplicity, the same CA is used within this enclave for both identity and permissions.
Demos

Using quality-of-service settings for lossy networks

Table of Contents

- Background
- Prerequisites
- Run the demo
  - Command line options
  - Add network traffic

Background

Please read the documentation page about QoS settings <../../Concepts/Intermediate/About-Quality-of-Service-Settings> for background information on available support in ROS 2.

In this demo, we will spawn a node that publishes a camera image and another that subscribes to the image and shows it on the screen. We will then simulate a lossy network connection between them and show how different quality of service settings handle the bad link.

Prerequisites

This tutorial assumes you have a working ROS 2 installation and OpenCV. See the OpenCV documentation for its installation instructions. You will also need the ROS package image_tools.

Linux Binaries

sudo apt-get install ros-iron-image-tools

From Source

# Clone and build the demos repo using the branch that matches your installation
git clone https://github.com/ros2/demos.git -b iron

Run the demo

Before running the demo, make sure you have a working webcam connected to your computer.

Once you’ve installed ROS 2, source your setup file:

Linux

. <path to ROS 2 install space>/setup.bash

macOS

. <path to ROS 2 install space>/setup.bash
Windows

```bash
call <path to ROS 2 install space>/local_setup.bat
```

Then run:

```bash
ros2 run image_tools showimage
```

Nothing will happen yet. `showimage` is a subscriber node that is waiting for a publisher on the `image` topic.

Note: you have to close the `showimage` process with `Ctrl-C` later. You can't just close the window.

In a separate terminal, source the install file and run the publisher node:

```bash
ros2 run image_tools cam2image
```

This will publish an image from your webcam. In case you don’t have a camera attached to your computer, there is a commandline option which publishes predefined images.

```bash
ros2 run image_tools cam2image --ros-args -p burger_mode:=True
```

In this window, you'll see terminal output:

```
PUBLISHING image #1
PUBLISHING image #2
PUBLISHING image #3
...
```

A window will pop up with the title “view” showing your camera feed. In the first window, you’ll see output from the subscriber:

```
RECEIVED image #1
RECEIVED image #2
RECEIVED image #3
...
```

**Note:** macOS users: If these examples do not work or you receive an error like `ddsi_conn_write failed -1` then you’ll need to increase your system wide UDP packet size:

```bash
$ sudo sysctl -w net.inet.udp.recvspace=209715
$ sudo sysctl -w net.inet.udp.maxdgram=65500
```

These changes will not persist a reboot. If you want the changes to persist, add these lines to `/etc/sysctl.conf` (create the file if it doesn’t exist already):

```bash
net.inet.udp.recvspace=209715
net.inet.udp.maxdgram=65500
```
Command line options

In one of your terminals, add a -h flag to the original command:

ros2 run image_tools showimage -h

Add network traffic

**Warning:** This section of the demo won’t work on RTI’s Connext DDS. When running multiple nodes in the same host, the RTI Connext DDS implementation uses shared memory along with the loopback interface. Degrading the loopback interface throughput won’t affect shared memory, thus traffic between the two nodes won’t be affected.

**Note:** This next section is Linux-specific.

However, for macOS and Windows you can achieve a similar effect with the utilities “Network Link Conditioner” (part of the xcode tool suite) and “Clumsy” (http://jagt.github.io/clumsy/index.html), respectively, but they will not be covered in this tutorial.

We are going to use the Linux network traffic control utility, tc (http://linux.die.net/man/8/tc).

```bash
sudo tc qdisc add dev lo root netem loss 5%
```

This magical incantation will simulate 5% packet loss over the local loopback device. If you use a higher resolution of the images (e.g. `--ros-args -p width:=640 -p height:=480`) you might want to try a lower packet loss rate (e.g. 1%).

Next we start the `cam2image` and `showimage`, and we’ll soon notice that both programs seem to have slowed down the rate at which images are transmitted. This is caused by the behavior of the default QoS settings. Enforcing reliability on a lossy channel means that the publisher (in this case, `cam2image`) will resend the network packets until it receives acknowledgement from the consumer (i.e. `showimage`).

Let’s now try running both programs, but with more suitable settings. First of all, we’ll use the `-p reliability:=best_effort` option to enable best effort communication. The publisher will now just attempt to deliver the network packets, and don’t expect acknowledgement from the consumer. We see now that some of the frames on the `showimage` side were dropped, so the frame numbers in the shell running `showimage` won’t be consecutive anymore:
When you’re done, remember to delete the queueing discipline:

```
sudo tc qdisc delete dev lo root netem loss 5%
```

**Managing nodes with managed lifecycles**

This page lives now directly side-by-side with the code. For more information about the lifecycle package, refer to rosindex.
Setting up efficient intra-process communication

Table of Contents

• Background
• Installing the demos
• Running and understanding the demos

Background

ROS applications typically consist of a composition of individual “nodes” which perform narrow tasks and are decoupled from other parts of the system. This promotes fault isolation, faster development, modularity, and code reuse, but it often comes at the cost of performance. After ROS 1 was initially developed, the need for efficient composition of nodes became obvious and Nodelets were developed. In ROS 2 we aim to improve on the design of Nodelets by addressing some fundamental problems that required restructuring of nodes.

In this demo we’ll be highlighting how nodes can be composed manually, by defining the nodes separately but combining them in different process layouts without changing the node’s code or limiting its abilities.

Installing the demos

See the installation instructions for details on installing ROS 2.

If you’ve installed ROS 2 from packages, ensure that you have ros-iron-intra-process-demo installed. If you downloaded the archive or built ROS 2 from source, it will already be part of the installation.

Running and understanding the demos

There are a few different demos: some are toy problems designed to highlight features of the intra process communications functionality and some are end to end examples which use OpenCV and demonstrate the ability to recombine nodes into different configurations.

The two node pipeline demo

This demo is designed to show that the intra process publish/subscribe connection can result in zero-copy transport of messages when publishing and subscribing with std::unique_ptrs.

First let’s take a look at the source:

https://github.com/ros2/demos/blob/iron/intra_process_demo/src/two_node_pipeline/two_node_pipeline.cpp

```cpp
#include <chrono>
#include <cstdint>
#include <cstdio>
#include <memory>
#include <string>
#include <utility>
```

(continues on next page)
```cpp
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/int32.hpp"

using namespace std::chrono_literals;

// Node that produces messages.
struct Producer : public rclcpp::Node
{
  Producer(const std::string & name, const std::string & output)
    : Node(name, rclcpp::NodeOptions().use_intra_process_comms(true))
    {
    // Create a publisher on the output topic.
    pub_ = this->create_publisher<std_msgs::msg::Int32>(output, 10);
    auto callback = []() -> void {
      auto pub_ptr = captured_pub.lock();
      if (!pub_ptr) {
        return;
      }
      static int32_t count = 0;
      std_msgs::msg::Int32::UniquePtr msg(new std_msgs::msg::Int32());
      msg->data = count++;
      printf("Published message with value: %d, and address: 0x%" PRIXPTR "\n",
             msg->data, reinterpret_cast<std::uintptr_t>(msg.get()));
      pub_ptr->publish(std::move(msg));
    };
    timer_ = this->create_wall_timer(1s, callback);
  }

  rclcpp::Publisher<std_msgs::msg::Int32>::SharedPtr pub_;  
  rclcpp::TimerBase::SharedPtr timer_;
};

// Node that consumes messages.
struct Consumer : public rclcpp::Node
{
  Consumer(const std::string & name, const std::string & input)
    : Node(name, rclcpp::NodeOptions().use_intra_process_comms(true))
    {
    // Create a subscription on the input topic which prints on receipt of new messages.
    sub_ = this->create_subscription<std_msgs::msg::Int32>(
        input,
        10,
        [](std_msgs::msg::Int32::UniquePtr msg) {
          printf("Received message with value: %d, and address: 0x%" PRIXPTR "\n",
                 msg->data, reinterpret_cast<std::uintptr_t>(msg.get()));
        });
  }
};
```

(continues on next page)
As you can see by looking at the `main` function, we have a producer and a consumer node, we add them to a single threaded executor, and then call spin.

If you look at the “producer” node’s implementation in the `Producer` struct, you can see that we have created a publisher which publishes on the “number” topic and a timer which periodically creates a new message, prints out its address in memory and its content’s value and then publishes it.

The “consumer” node is a bit simpler, you can see its implementation in the `Consumer` struct, as it only subscribes to the “number” topic and prints the address and value of the message it receives.

The expectation is that the producer will print out an address and value and the consumer will print out a matching address and value. This demonstrates that intra process communication is indeed working and unnecessary copies are avoided, at least for simple graphs.

Let’s run the demo by executing `ros2 run intra_process_demo two_node_pipeline` executable (don’t forget to source the setup file first):

```
$ ros2 run intra_process_demo two_node_pipeline
Published message with value: 0, and address: 0x7fb02303fa0
Published message with value: 1, and address: 0x7fb020cf0520
Received message with value: 1, and address: 0x7fb020cf0520
Published message with value: 2, and address: 0x7fb020e12900
Received message with value: 2, and address: 0x7fb020e12900
Received message with value: 3, and address: 0x7fb020cf0520
Published message with value: 3, and address: 0x7fb020e12900
Received message with value: 4, and address: 0x7fb020e12900
Received message with value: 4, and address: 0x7fb020e12900
Published message with value: 5, and address: 0x7fb02303cea0
Received message with value: 5, and address: 0x7fb02303cea0
[...]
```

One thing you’ll notice is that the messages tick along at about one per second. This is because we told the timer to fire at about once per second.
Also you may have noticed that the first message (with value 0) does not have a corresponding “Received message …” line. This is because publish/subscribe is “best effort” and we do not have any “latching” like behavior enabled. This means that if the publisher publishes a message before the subscription has been established, the subscription will not receive that message. This race condition can result in the first few messages being lost. In this case, since they only come once per second, usually only the first message is lost.

Finally, you can see that “Published message…” and “Received message …” lines with the same value also have the same address. This shows that the address of the message being received is the same as the one that was published and that it is not a copy. This is because we’re publishing and subscribing with std::unique_ptr which allow ownership of a message to be moved around the system safely. You can also publish and subscribe with const & and std::shared_ptr, but zero-copy will not occur in that case.

The cyclic pipeline demo

This demo is similar to the previous one, but instead of the producer creating a new message for each iteration, this demo only ever uses one message instance. This is achieved by creating a cycle in the graph and “kicking off” communication by externally making one of the nodes publish before spinning the executor:

https://github.com/ros2/demos/blob/iron/intra_process_demo/src/cyclic_pipeline/cyclic_pipeline.cpp

```cpp
#include <chrono>
#include <cinttypes>
#include <cstdio>
#include <memory>
#include <string>
#include <utility>
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/int32.hpp"

using namespace std::chrono_literals;

// This node receives an Int32, waits 1 second, then increments and sends it.
struct IncrementerPipe : public rclcpp::Node
{
    IncrementerPipe(const std::string & name, const std::string & in, const std::string & out)
        : Node(name, rclcpp::NodeOptions().use_intra_process_comms(true))
    {
        // Create a publisher on the output topic.
        pub = this->create_publisher<std_msgs::msg::Int32>(out, 10);
        std::weak_ptr<std::remove_pointer<decltype(pub.get())>::type> captured_pub = pub;
        // Create a subscription on the input topic.
        sub = this->create_subscription<std_msgs::msg::Int32>(
            in, 10,
            [captured_pub](std_msgs::msg::Int32::UniquePtr msg) {
                auto pub_ptr = captured_pub.lock();
                if (!pub_ptr) {
                    return;
                }
                printf("Received message with value: %d, and address: 0x" PRIXPTR "\n", msg->
```
(continues on next page)
```cpp
reinterpret_cast<std::uintptr_t>(msg.get()));
printf(" sleeping for 1 second...
if (!rclcpp::sleep_for(1s)) {
    return;  // Return if the sleep failed (e.g. on :kbd:`ctrl-c`).
} printf(" done.
    "Incrementing and sending with value: %d, and address: 0x" PRIXPTR "\n", msg->
data,
    reinterpret_cast<std::uintptr_t>(msg.get()));
    pub_ptr->publish(std::move(msg));  // Send the message along to the output
}));
}
rclcpp::Publisher<std_msgs::msg::Int32>::SharedPtr pub;
rclcpp::Subscription<std_msgs::msg::Int32>::SharedPtr sub;

int main(int argc, char * argv[]) {
    setvbuf(stdout, NULL, _IONBF, BUFSIZ);
rclcpp::init(argc, argv);
rclcpp::executors::SingleThreadedExecutor executor;

    // Create a simple loop by connecting the in and out topics of two IncrementerPipe's.
    // The expectation is that the address of the message being passed between them never
    // changes.
    auto pipe1 = std::make_shared<IncrementerPipe>("pipe1", "topic1", "topic2");
    auto pipe2 = std::make_shared<IncrementerPipe>("pipe2", "topic2", "topic1");
rclcpp::sleep_for(1s);  // Wait for subscriptions to be established to avoid race
    // conditions.
    // Publish the first message (kicking off the cycle).
    std::unique_ptr<std_msgs::msg::Int32> msg(new std_msgs::msg::Int32());
    msg->data = 42;
    printf("Published first message with value: %d, and address: 0x" PRIXPTR "\n", msg->data,
        reinterpret_cast<std::uintptr_t>(msg.get()));
    pipe1->pub_ptr->publish(std::move(msg));

    executor.add_node(pipe1);
    executor.add_node(pipe2);
    executor.spin();
rclcpp::shutdown();

    return 0;
}
```

Unlike the previous demo, this demo uses only one Node, instantiated twice with different names and configurations. The graph ends up being pipe1 -> pipe2 -> pipe1... in a loop.
The line `pipe1->pub->publish(msg);` kicks the process off, but from then on the messages are passed back and forth between the nodes by each one calling `publish` within its own subscription callback.

The expectation here is that the nodes pass the message back and forth, once a second, incrementing the value of the message each time. Because the message is being published and subscribed to as a `unique_ptr` the same message created at the beginning is continuously used.

To test those expectations, let’s run it:

```bash
$ ros2 run intra_process_demo cyclic_pipeline
```

You should see ever increasing numbers on each iteration, starting with 42... because 42, and the whole time it reuses the same message, as demonstrated by the pointer addresses which do not change, which avoids unnecessary copies.

**The image pipeline demo**

In this demo we’ll use OpenCV to capture, annotate, and then view images.

**Note:** If you are on macOS and these examples do not work or you receive an error like `ddsi_conn_write failed -1`, then you’ll need to increase your system wide UDP packet size:

```bash
$ sudo sysctl -w net.inet.udp.recvspace=209715
$ sudo sysctl -w net.inet.udp.maxdgram=65500
```

These changes will not persist after a reboot.
Simple pipeline

First we’ll have a pipeline of three nodes, arranged as such: `camera_node -> watermark_node -> image_view_node`

The `camera_node` reads from camera device 0 on your computer, writes some information on the image and publishes it. The `watermark_node` subscribes to the output of the `camera_node` and adds more text before publishing it too. Finally, the `image_view_node` subscribes to the output of the `watermark_node`, writes more text to the image and then visualizes it with `cv::imshow`.

In each node the address of the message which is being sent, or which has been received, or both, is written to the image. The watermark and image view nodes are designed to modify the image without copying it and so the addresses imprinted on the image should all be the same as long as the nodes are in the same process and the graph remains organized in a pipeline as sketched above.

**Note:** On some systems (we’ve seen it happen on Linux), the address printed to the screen might not change. This is because the same unique pointer is being reused. In this situation, the pipeline is still running.

Let's run the demo by executing the following executable:

```bash
ros2 run intra_process_demo image_pipeline_all_in_one
```

You should see something like this:
You can pause the rendering of the image by pressing the spacebar and you can resume by pressing the spacebar again. You can also press q or ESC to exit.

If you pause the image viewer, you should be able to compare the addresses written on the image and see that they are the same.

### Pipeline with two image viewers

Now let’s look at an example just like the one above, except it has two image view nodes. All the nodes are still in the same process, but now two image view windows should show up. (Note for macOS users: your image view windows might be on top of each other). Let’s run it with the command:

```
ros2 run intra_process_demo image_pipeline_with_two_image_view
```

Just like the last example, you can pause the rendering with the spacebar and continue by pressing the spacebar a second time. You can stop the updating to inspect the pointers written to the screen.

As you can see in the example image above, we have one image with all of the pointers the same and then another image with the same pointers as the first image for the first two entries, but the last pointer on the second image is different. To understand why this is happening consider the graph’s topology:

```
camera_node -> watermark_node -> image_view_node
                -> image_view_node2
```

The link between the `camera_node` and the `watermark_node` can use the same pointer without copying because there is only one intra process subscription to which the message should be delivered. But for the link between the `watermark_node` and the two image view nodes the relationship is one to many, so if the image view nodes were using `unique_ptr` callbacks then it would be impossible to deliver the ownership of the same pointer to both. It can be, however, delivered to one of them. Which one would get the original pointer is not defined, but instead is simply the last to be delivered.

Note that the image view nodes are not subscribed with `unique_ptr` callbacks. Instead they are subscribed with `const shared_ptr`s. This means the system delivers the same `shared_ptr` to both callbacks. When the first intraprocess subscription is handled, the internally stored `unique_ptr` is promoted to a `shared_ptr`. Each of the callbacks will receive shared ownership of the same message.
Pipeline with interprocess viewer

One other important thing to get right is to avoid interruption of the intra process zero-copy behavior when interprocess subscriptions are made. To test this we can run the first image pipeline demo, `image_pipeline_all_in_one`, and then run an instance of the stand alone `image_view_node` (don’t forget to prefix them with `ros2 run intra_process_demo` in the terminal). This will look something like this:

![Image showing the process IDs and addresses for the pipeline and interprocess view](image)

It's hard to pause both images at the same time so the images may not line up, but the important thing to notice is that the `image_pipeline_all_in_one` image view shows the same address for each step. This means that the intra process zero-copy is preserved even when an external view is subscribed as well. You can also see that the interprocess image view has different process IDs for the first two lines of text and the process ID of the standalone image viewer in the third line of text.

Recording and playing back data with rosbag using the ROS 1 bridge

This tutorial is a follow up to the *Bridge communication between ROS 1 and ROS 2* demo as can be found [here](#), and in the following it is assumed you have completed that tutorial already.

The ros1_bridge can be built from [source](#) for these examples.

What follows is a series of additional examples, like that ones that come at the end of the aforementioned *Bridge communication between ROS 1 and ROS 2* demo.

Recording topic data with rosbag and ROS 1 Bridge

In this example, we’ll be using the `cam2image` demo program that comes with ROS 2 and a Python script to emulate a simple turtlebot-like robot's sensor data so that we can bridge it to ROS 1 and use rosbag to record it.

First we’ll run a ROS 1 `roscore` in a new shell:

```
# Shell A:
. /opt/ros/kinetic/setup.bash
# Or, on OSX, something like:
# . ~/ros_catkin_ws/install_isolated/setup.bash
roscore
```
Then we’ll run the ROS 1 <=> ROS 2 dynamic_bridge with the `--bridge-all-topics` option (so we can do rostopic list and see them) in another shell:

```bash
# Shell B:
. /opt/ros/kinetic/setup.bash
# Or, on OSX, something like:
. ~/ros_catkin_ws/install_isolated/setup.bash
. /opt/ros/ardent/setup.bash
# Or, if building ROS 2 from source:
# . <workspace-with-bridge>/install/setup.bash
export ROS_MASTER_URI=http://localhost:11311
ros2 run ros1_bridge dynamic_bridge --bridge-all-topics
```

Remember to replace `<workspace-with-bridge>` with the path to where you either extracted the ROS 2 binary or where you built ROS 2 from source.

Now we can start up the ROS 2 programs that will emulate our turtlebot-like robot. First we’ll run the `cam2image` program with the `-b` option so it doesn’t require a camera to work:

```bash
# Shell C:
. /opt/ros/ardent/setup.bash
# Or, if building ROS 2 from source:
# . <workspace-with-bridge>/install/setup.bash
ros2 run image_tools cam2image -- -b
```

TODO: use namespaced topic names

Then we’ll run a simple Python script to emulate the `odom` and `imu_data` topics from a Kobuki base. I would use the more accurate `~sensors/imu_data` topic name for the imu data, but we don’t have namespace support just yet in ROS 2 (it’s coming!). Place this script in a file called `emulate_kobuki_node.py`:

```python
#!/usr/bin/env python3

import sys
import time
import rclpy
from nav_msgs.msg import Odometry
from sensor_msgs.msg import Imu

def main():
    rclpy.init(args=sys.argv)

    node = rclpy.create_node('emulate_kobuki_node')

    imu_publisher = node.create_publisher(Imu, 'imu_data')
    odom_publisher = node.create_publisher(Odometry, 'odom')

    imu_msg = Imu()
    odom_msg = Odometry()
    counter = 0

    while True:
        (continues on next page)
```
counter += 1
now = time.time()
if (counter % 50) == 0:
    odom_msg.header.stamp.sec = int(now)
    odom_msg.header.stamp.nanosec = int(now * 1e9) % 1000000000
    odom_publisher.publish(odom_msg)
if (counter % 100) == 0:
    imu_msg.header.stamp.sec = int(now)
    imu_msg.header.stamp.nanosec = int(now * 1e9) % 1000000000
    imu_publisher.publish(imu_msg)
counter = 0
time.sleep(0.001)

if __name__ == '__main__':
sys.exit(main())

You can run this python script in a new ROS 2 shell:

```
# Shell D:
./opt/ros/ardent/setup.bash
# Or, if building ROS 2 from source:
# . <workspace-with-bridge>/install/setup.bash
python3 emulate_kobuki_node.py
```

Now that all the data sources and the dynamic bridge are running, we can look at the available topics in a new ROS 1 shell:

```
# Shell E:
./opt/ros/kinetic/setup.bash
# Or, on OSX, something like:
# . ~/ros_catkin_ws/install_isolated/setup.bash
rostopic list
```

You should see something like this:

```
% rostopic list
/image
/imu_data
/odom
/rosout
/rosout_agg
```

We can now record this data with `robag record` in the same shell:

```
# Shell E:
robag record /image /imu_data /odom
```

After a few seconds you can `Ctrl-c` the `robag` command and do an `ls -lh` to see how big the file is, you might see something like this:
% ls -lh
 total 0
 -rw-rw-r-- 1 william william 12M Feb 23 16:59 2017-02-23-16-59-47.bag

Though the file name will be different for your bag (since it is derived from the date and time).

**Playing back topic data with rosbag and ROS 1 Bridge**

Now that we have a bag file you can use any of the ROS 1 tools to introspect the bag file, like `rosbag info <bag file>`, `rostopic list -b <bag file>`, or `rqt_bag <bag file>`. However, we can also playback bag data into ROS 2 using `rosbag play` and the ROS 1 <=> ROS 2 `dynamic_bridge`.

First close out all the shells you opened for the previous tutorial, stopping any running programs.

Then in a new shell start the `roscore`:

```bash
# Shell P:
. /opt/ros/kinetic/setup.bash
# Or, on OSX, something like:
# . ~/ros_catkin_ws/install_isolated/setup.bash
roscore
```

Then run the `dynamic_bridge` in another shell:

```bash
# Shell Q:
. /opt/ros/kinetic/setup.bash
# Or, on OSX, something like:
# . ~/ros_catkin_ws/install_isolated/setup.bash
. /opt/ros/ardent/setup.bash
# Or, if building ROS 2 from source:
# . <workspace-with-bridge>/install/setup.bash
export ROS_MASTER_URI=http://localhost:11311
ros2 run ros1_bridge dynamic_bridge --bridge-all-topics
```

Then play the bag data back with `rosbag play` in another new shell, using the `--loop` option so that we don’t have to keep restarting it for short bags:

```bash
# Shell R:
. /opt/ros/kinetic/setup.bash
# Or, on OSX, something like:
# . ~/ros_catkin_ws/install_isolated/setup.bash
rosbag play --loop path/to/bag_file
```

Make sure to replace `path/to/bag_file` with the path to the bag file you want to playback.

Now that the data is being played back and the bridge is running we can see the data coming across in ROS 2.

```bash
# Shell S:
. /opt/ros/ardent/setup.bash
# Or, if building ROS 2 from source:
# . <workspace-with-bridge>/install/setup.bash
ros2 topic list
topic echo /odom
```
You should see something like:

```
% ros2 topic list
/clock
/image
/imu_data
/odom
/parameter_events
```

You can also see the image being played from the bag by using the `showimage` tool:

```
ros2 run image_tools showimage
```

### Understanding real-time programming

<table>
<thead>
<tr>
<th>Table of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Background</td>
</tr>
<tr>
<td>• Install and run the demo</td>
</tr>
<tr>
<td>– Run the tests</td>
</tr>
<tr>
<td>– What the heck just happened?</td>
</tr>
<tr>
<td>– Adjust permissions for memory locking</td>
</tr>
<tr>
<td>– Output overview</td>
</tr>
<tr>
<td>– Latency</td>
</tr>
<tr>
<td>– Setting permissions for the scheduler</td>
</tr>
<tr>
<td>– Plotting results</td>
</tr>
</tbody>
</table>

#### Background

Real-time computing is a key feature of many robotics systems, particularly safety- and mission-critical applications such as autonomous vehicles, spacecrafts, and industrial manufacturing. We are designing and prototyping ROS 2 with real-time performance constraints in mind, since this is a requirement that was not considered in the early stages of ROS 1 and it is now intractable to refactor ROS 1 to be real-time friendly.

This document outlines the requirements of real-time computing and best practices for software engineers. In short:

To make a real-time computer system, our real-time loop must update periodically to meet deadlines. We can only tolerate a small margin of error on these deadlines (our maximum allowable jitter). To do this, we must avoid nondeterministic operations in the execution path, things like: pagefault events, dynamic memory allocation/deallocation, and synchronization primitives that block indefinitely.

A classic example of a controls problem commonly solved by real-time computing is balancing an inverted pendulum. If the controller blocked for an unexpectedly long amount of time, the pendulum would fall down or go unstable. But if the controller reliably updates at a rate faster than the motor controlling the pendulum can operate, the pendulum will successfully adapt react to sensor data to balance the pendulum.

Now that you know everything about real-time computing, let’s try a demo!
Install and run the demo

The real-time demo was written with Linux operating systems in mind, since many members of the ROS community doing real-time computing use Xenomai or RT_PREEMPT as their real-time solutions. Since many of the operations done in the demo to optimize performance are OS-specific, the demo only builds and runs on Linux systems. So, if you are an OSX or Windows user, don’t try this part!

Also this must be built from source using a the static DDS API. Currently the only supported implementation is Connext.

First, follow the instructions to build ROS 2 from source using Connext DDS as the middleware.

Run the tests

Before you run make sure you have at least 8Gb of RAM free. With the memory locking, swap will not work anymore.

Source your ROS 2 setup.bash.

Run the demo binary, and redirect the output. You may want to use sudo in case you get permission error:

```bash
pendulum_demo > output.txt
```

What the heck just happened?

First, even though you redirected stdout, you will see some output to the console (from stderr):

```
mlockall failed: Cannot allocate memory
Couldn't lock all cached virtual memory.
Pagefaults from reading pages not yet mapped into RAM will be recorded.
```

After the initialization stage of the demo program, it will attempt to lock all cached memory into RAM and prevent future dynamic memory allocations using mlockall. This is to prevent pagefaults from loading lots of new memory into RAM. (See the realtime design article for more information.)

The demo will continue on as usual when this occurs. At the bottom of the output.txt file generated by the demo, you’ll see the number of pagefaults encountered during execution:

```
rttest statistics:
- Minor pagefaults: 20
- Major pagefaults: 0
```

If we want those pagefaults to go away, we’ll have to…
Adjust permissions for memory locking

Add to /etc/security/limits.conf (as sudo):

```bash
<your username>    - memlock   <limit in kB>
```

A limit of -1 is unlimited. If you choose this, you may need to accompany it with `ulimit -l unlimited` (as root) after editing the file.

After saving the file, log out and log back in. Then rerun the `pendulum_demo` invocation.

You’ll either see zero pagefaults in your output file, or an error saying that a bad_alloc exception was caught. If this happened, you didn’t have enough free memory available to lock the memory allocated for the process into RAM. You’ll need to install more RAM in your computer to see zero pagefaults!

Output overview

To see more output, we have to run the `pendulum_logger` node.

In one shell with your `install/setup.bash` sourced, invoke:

```bash
pendulum_logger
```

You should see the output message:

```
Logger node initialized.
```

In another shell with `setup.bash` sourced, invoke `pendulum_demo` again.

As soon as this executable starts, you should see the other shell constantly printing output:

```
Commanded motor angle: 1.570796
Actual motor angle: 1.570796
Mean latency: 210144.000000 ns
Min latency: 4805 ns
Max latency: 578137 ns
Minor pagefaults during execution: 0
Major pagefaults during execution: 0
```

The demo is controlling a very simple inverted pendulum simulation. The pendulum simulation calculates its position in its own thread. A ROS node simulates a motor encoder sensor for the pendulum and publishes its position. Another ROS node acts as a simple PID controller and calculates the next command message.

The logger node periodically prints out the pendulum’s state and the runtime performance statistics of the demo during its execution phase.

After the `pendulum_demo` is finished, you’ll have to CTRL-C out of the logger node to exit.
Latency

At the pendulum_demo execution, you’ll see the final statistics collected for the demo:

<table>
<thead>
<tr>
<th>rttest statistics:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Minor pagefaults: 0</td>
</tr>
<tr>
<td>- Major pagefaults: 0</td>
</tr>
<tr>
<td>Latency (time after deadline was missed):</td>
</tr>
<tr>
<td>- Min: 3354 ns</td>
</tr>
<tr>
<td>- Max: 2752187 ns</td>
</tr>
<tr>
<td>- Mean: 19871.8 ns</td>
</tr>
<tr>
<td>- Standard deviation: 1.35819e+08</td>
</tr>
</tbody>
</table>

PendulumMotor received 985 messages
PendulumController received 987 messages

The latency fields show you the minimum, maximum, and average latency of the update loop in nanoseconds. Here, latency means the amount of time after the update was expected to occur.

The requirements of a real-time system depend on the application, but let’s say in this demo we have a 1kHz (1 millisecond) update loop, and we’re aiming for a maximum allowable latency of 5% of our update period.

So, our average latency was really good in this run, but the maximum latency was unacceptable because it actually exceeded our update loop! What happened?

We may be suffering from a non-deterministic scheduler. If you’re running a vanilla Linux system and you don’t have the RT_PREEMPT kernel installed, you probably won’t be able to meet the real-time goal we set for ourselves, because the Linux scheduler won’t allow you to arbitrarily pre-empt threads at the user level.

See the realtime design article for more information.

The demo attempts to set the scheduler and thread priority of the demo to be suitable for real-time performance. If this operation failed, you’ll see an error message: “Couldn’t set scheduling priority and policy: Operation not permitted”. You can get slightly better performance by following the instructions in the next section:

**Setting permissions for the scheduler**

Add to /etc/security/limits.conf (as sudo):

| <your username> | rtprio 98 |

The range of the rtprio (real-time priority) field is 0-99. However, do NOT set the limit to 99 because then your processes could interfere with important system processes that run at the top priority (e.g. watchdog). This demo will attempt to run the control loop at priority 98.

**Plotting results**

You can plot the latency and pagefault statistics that are collected in this demo after the demo runs.

Because the code has been instrumented with rttest, there are useful command line arguments available:
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>-i</td>
<td>Specify how many iterations to run the real-time loop</td>
<td>1000</td>
</tr>
<tr>
<td>-u</td>
<td>Specify the update period with the default unit being microseconds. Use the suffix “s” for seconds, “ms” for milliseconds, “us” for microseconds, and “ns” for nanoseconds.</td>
<td>1ms</td>
</tr>
<tr>
<td>-f</td>
<td>Specify the name of the file for writing the collected data.</td>
<td></td>
</tr>
</tbody>
</table>

Run the demo again with a filename to save results:

```bash
pendulum_demo -f pendulum_demo_results
```

Then run the `rttest_plot` script on the resulting file:

```bash
ros2 run rttest rttest_plot pendulum_demo_results
```

This script will produce a number of files:

- `pendulum_demo_results_plot_latency.svg`
- `pendulum_demo_results_plot_latency_hist.svg`
- `pendulum_demo_results_plot_majflts.svg`
- `pendulum_demo_results_plot_minflts.svg`

You can view these plots in an image viewer of your choice.

**Experimenting with a dummy robot**

In this demo, we present a simple demo robot with all components from publishing joint states over publishing fake laser data until visualizing the robot model on a map in RViz.

**Launching the demo**

We assume your ROS 2 installation dir as `~/.ros2_ws`. Please change the directories according to your platform.

To start the demo, we execute the demo bringup launch file, which we are going to explain in more details in the next section.

```bash
source ~/.ros2_ws/install/setup.bash
dummy_robot_bringup dummy_robot_bringup_launch.py
```

You should see some prints inside your terminal along the lines of the following:

```
[INFO] [launch]: process[dummy_map_server-1]: started with pid [25812]
[INFO] [launch]: process[dummy_state_publisher-2]: started with pid [25813]
[INFO] [launch]: process[dummy_joint_states-3]: started with pid [25814]
[INFO] [launch]: process[dummy_laser-4]: started with pid [25815]
Initialize urdf model from file: /home/mikael/work/ros2/bouncy_ws/install_debug_isolated/...dummy_robot_bringup/share/dummy_robot_bringup/launch/single_rrbot.urdf
Parsing robot urdf xml string.
Link single_rrbot_link1 had 1 children
Link single_rrbot_link2 had 1 children
```

(continues on next page)
If you now open in a next terminal your RViz, you’ll see your robot.

```bash
$ source <ROS2_INSTALL_FOLDER>/setup.bash
$ rviz2
```

This opens RViz2. Assuming you have your dummy_robot_bringup still launched, you can now add the TF display plugin and configure your global frame to `world`. Once you did that, you should see a similar picture:
What's happening?

If you have a closer look at the launch file, we start a couple of nodes at the same time.

- `dummy_map_server`
- `dummy_laser`
- `dummy_joint_states`
- `robot_state_publisher`

The first two packages are relatively simple. The `dummy_map_server` constantly publishes an empty map with a periodic update. The `dummy_laser` does basically the same; publishing dummy fake laser scans.

The `dummy_joint_states` node is publishing fake joint state data. As we are publishing a simple RRbot with only two joints, this node publishes joint states values for these two joints.

The `robot_state_publisher` is doing the actual interesting work. It parses the given URDF file, extracts the robot model and listens to the incoming joint states. With this information, it publishes TF values for our robot which we visualize in RViz.

Hooray!
Logging

Table of Contents

- Using log statements in code
  - Basic logging
  - Logging only the first time
  - Logging all but the first time
  - Logging throttled
  - Logging throttled all but the first time
- Logging demo
- Logging directory configuration
- Logger level configuration: programmatically
- Logger level configuration: externally
  - Using the logger config component
- Logger level configuration: command line
  - Console output formatting
  - Console output colorizing
  - Default stream for console output
  - Line buffered console output

See the logging page <../../Concepts/Intermediate/About-Logging> for details on available functionality.

Using log statements in code

Basic logging

The following code will output a log message from a ROS 2 node at DEBUG severity:

C++

```cpp
// printf style
RCLCPP_DEBUG(node->get_logger(), "My log message %d", 4);

// C++ stream style
RCLCPP_DEBUG_STREAM(node->get_logger(), "My log message " << 4);
```

Python

```python
node.get_logger().debug('My log message %d' % (4))
```

Note that in both cases, no trailing newline is added, as the logging infrastructure will automatically add one.
Logging only the first time

The following code will output a log message from a ROS 2 node at INFO severity, but only the first time it is hit:

C++

```c++
// printf style
RCLCPP_INFO_ONCE(node->get_logger(), "My log message %d", 4);

// C++ stream style
RCLCPP_INFO_STREAM_ONCE(node->get_logger(), "My log message " << 4);
```

Python

```python
num = 4
node.get_logger().info(f"My log message {num}", once=True)
```

Logging all but the first time

The following code will output a log message from a ROS 2 node at WARN severity, but not the very first time it is hit:

C++

```c++
// printf style
RCLCPP_WARN_SKIPFIRST(node->get_logger(), "My log message %d", 4);

// C++ stream style
RCLCPP_WARN_STREAM_SKIPFIRST(node->get_logger(), "My log message " << 4);
```

Python

```python
num = 4
node.get_logger().warning('My log message {0}'.format(num), skip_first=True)
```

Logging throttled

The following code will output a log message from a ROS 2 node at ERROR severity, but no more than once per second. The interval parameter specifying milliseconds between messages should have an integer data type so it can be converted to a rcutils_duration_value_t (an int64_t):

C++

```c++
// printf style
RCLCPP_ERROR_THROTTLE(node->get_logger(), *node->get_clock(), 1000, "My log message %d", 4);

// C++ stream style
RCLCPP_ERROR_STREAM_THROTTLE(node->get_logger(), *node->get_lock(), 1000, "My log message " << 4);
```

(continues on next page)
// For now, use the nanoseconds() method to use an existing rclcpp::Duration value, see
→ https://github.com/ros2/rclcpp/issues/1929
RCLCPP_ERROR_STREAM_THROTTLE(node->get_logger(), *node->get_clock(), msg_interval.
→ nanoseconds() / 1000000, "My log message " << 4);

Python
num = 4
node.get_logger().error(f'My log message {num}', throttle_duration_sec=1)

Logging throttled all but the first time

The following code will output a log message from a ROS 2 node at DEBUG severity, no more than once per second, skipping the very first time it is hit:
C++

// printf style
RCLCPP_DEBUG_SKIPFIRST_THROTTLE(node->get_logger(), *node->get_clock(), 1000, "My log␣
→ message %d", 4);

RCLCPP_DEBUG_SKIPFIRST_THROTTLE(node->get_logger(), *node->get_clock(), 1000, "My log␣
→ message " << 4);

Python
num = 4
node.get_logger().debug(f'My log message {num}', skip_first=True, throttle_duration_␣
→ sec=1.0)

Logging demo

In this demo, different types of log calls are shown and the severity level of different loggers is configured locally and externally.

Start the demo with:

ros2 run logging_demo logging_demo_main

Over time you will see output from various log calls with different properties. To start with you will only see output from log calls with severity INFO and above (WARN, ERROR, FATAL). Note that the first message will only be logged once, though the line is reached on each iteration, as that is a property of the log call used for that message.
Logging directory configuration

The logging directory can be configured through two environment variables: $ROS_LOG_DIR and $ROS_HOME. The logic is as follows:

- Use $ROS_LOG_DIR if $ROS_LOG_DIR is set and not empty.
- Otherwise, use $ROS_HOME/log, using ~/.ros for $ROS_HOME if not set or if empty.

For example, to set the logging directory to ~/my_logs:

**Linux**

```bash
export ROS_LOG_DIR=~/my_logs
ros2 run logging_demo logging_demo_main
```

**macOS**

```bash
export ROS_LOG_DIR=~/my_logs
ros2 run logging_demo logging_demo_main
```

**Windows**

```bash
set "ROS_LOG_DIR=~/my_logs"
ros2 run logging_demo logging_demo_main
```

You will then find the logs under ~/my_logs/.

Alternatively, you can set $ROS_HOME and the logging directory will be relative to it ($ROS_HOME/log). $ROS_HOME is intended to be used by anything that needs a base directory. Note that $ROS_LOG_DIR has to be either unset or empty.

For example, with $ROS_HOME set to ~/my_ros_home:

**Linux**

```bash
export ROS_HOME=~/my_ros_home
ros2 run logging_demo logging_demo_main
```

**macOS**

```bash
export ROS_HOME=~/my_ros_home
ros2 run logging_demo logging_demo_main
```

**Windows**

```bash
set "ROS_HOME=~/my_ros_home"
ros2 run logging_demo logging_demo_main
```

You will then find the logs under ~/my_ros_home/log/.
Vulcanexus Documentation, Release 1.0.0

Logger level configuration: programmatically
After 10 iterations the level of the logger will be set to DEBUG, which will cause additional messages to be logged.
Some of these debug messages cause additional functions/expressions to be evaluated, which were previously skipped
as DEBUG log calls were not enabled. See the source code of the demo for further explanation of the calls used, and see
the rclcpp logging documentation for a full list of supported logging calls.
Logger level configuration: externally
ROS 2 nodes have services available to configure the logging level externally at runtime. These services are disabled
by default. The following code shows how to enable the logger service while creating the node.
Linux
// Create a node with logger service enabled
auto node = std::make_shared<rclcpp::Node>("NodeWithLoggerService",␣
˓→rclcpp::NodeOptions().enable_logger_service(true))
Python
# Create a node with logger service enabled
node = Node('NodeWithLoggerService', enable_logger_service=True)
If you run one of the nodes as configured above, you will find 2 services when running ros2 service list:
$ ros2 service list
...
/NodeWithLoggerService/get_logger_levels
/NodeWithLoggerService/set_logger_levels
...
• get_logger_levels
Use this service to get logger levels for specified logger names.
Run ros2 service call to get logger levels for NodeWithLoggerService and rcl.
$ ros2 service call /NodeWithLoggerService/get_logger_levels rcl_interfaces/
˓→srv/GetLoggerLevels '{names: ["NodeWithLoggerService", "rcl"]}'
requester: making request: rcl_interfaces.srv.GetLoggerLevels_
˓→Request(names=['NodeWithLoggerService', 'rcl'])
response:
rcl_interfaces.srv.GetLoggerLevels_Response(levels=[rcl_interfaces.msg.
˓→LoggerLevel(name='NodeWithLoggerService', level=0), rcl_interfaces.msg.
˓→LoggerLevel(name='rcl', level=0)])
• set_logger_levels
Use this service to set logger levels for specified logger names.
Run ros2 service call to set logger levels for NodeWithLoggerService and rcl.

1072

Chapter 4. Contributing to the documentation


There is also demo code showing how to set or get the logger level via the logger service.

- **rclcpp:** [demo code](#)

  ```
  $ ros2 run demo_nodes_cpp use_logger_service
  ```

- **rclpy:** [demo code](#)

  ```
  $ ros2 run demo_nodes_py use_logger_service
  ```

**Warning:** Currently, there is a limitation that `get_logger_levels` and `set_logger_levels` services are not thread-safe. This means that you need to ensure that only one thread is calling the services at a time. Please see the details in [https://github.com/ros2/rcutils/issues/397](https://github.com/ros2/rcutils/issues/397)

### Using the logger config component

The server that responds to the logger configuration requests has been developed as a component so that it may be added to an existing composition-based system. For example, if you are using a [container to run your nodes](#), to be able to configure your loggers you only need to request that it additionally load the `logging_demo::LoggerConfig` component into the container.

As an example, if you want to debug the `composition::Talker` demo, you can start the talker as normal with:

Shell 1:

```
ros2 run rclcpp_components component_container
```

Shell 2:

```
ros2 component load /ComponentManager composition composition::Talker
```

And then when you want to enable debug logging, load the `LoggerConfig` component with:

Shell 2:

```
ros2 component load /ComponentManager logging_demo logging_demo::LoggerConfig
```

And finally, configure all unset loggers to the debug severity by addressing the empty-named logger. Note that loggers that have been specifically configured to use a particular severity will not be affected by this call.

Shell 2:

```
4.8. ROS 2 Documentation 1073
```
You should see debug output from any previously unset loggers in the process start to appear, including from the ROS 2 core.

**Logger level configuration: command line**

As of the Bouncy ROS 2 release, the severity level for loggers that have not had their severity set explicitly can be configured from the command line. Restart the demo including the following command line argument:

```
ros2 run logging_demo logging_demo_main --ros-args --log-level debug
```

This configures the default severity for any unset logger to the debug severity level. You should see debug output from loggers from the demo itself and from the ROS 2 core.

The severity level for individual loggers can be configured from the command-line. Restart the demo including the following command line arguments:

```
ros2 run logging_demo logging_demo_main --ros-args --log-level logger_usage_demo:=debug
```

**Console output formatting**

If you would like more or less verbose formatting, you can use `RCUTILS_CONSOLE_OUTPUT_FORMAT` environment variable. For example, to additionally get the timestamp and location of the log calls, stop the demo and restart it with the environment variable set:

**Linux**

```
export RCUTILS_CONSOLE_OUTPUT_FORMAT="[{severity} {time}] [{name}]: {message} ({function_˓→name}() at {file_name}:{line_number})"
```

**macOS**

```
export RCUTILS_CONSOLE_OUTPUT_FORMAT="[{severity} {time}] [{name}]: {message} ({function_˓→name}() at {file_name}:{line_number})"
```

**Windows**

```
# set "RCUTILS_CONSOLE_OUTPUT_FORMAT=[{severity} {time}] [{name}]: {message} ({function_˓→name}() at {file_name}:{line_number})"
```

ros2 run logging_demo logging_demo_main

You should see the timestamp in seconds and the function name, filename and line number additionally printed with each message. The ``time`` option is only supported as of the ROS 2 Bouncy release.
Console output colorizing

By default, the output is colorized when it's targeting a terminal. If you would like to force enabling or disabling it, you can use the `RCUTILS_COLORIZED_OUTPUT` environment variable. For example:

Linux

```bash
export RCUTILS_COLORIZED_OUTPUT=0  # 1 for forcing it
```

macOS

```bash
export RCUTILS_COLORIZED_OUTPUT=0  # 1 for forcing it
```

Windows

```bash
# set "RCUTILS_COLORIZED_OUTPUT=0"
ros2 run logging_demo logging_demo_main
```

You should see that debug, warn, error and fatal logs aren't colorized now.

**Note:** In Linux and MacOS forcing colorized output means that if you redirect the output to a file, the ansi escape color codes will appear on it. In windows the colorization method relies on console APIs. If it is forced you will get a new warning saying that colorization failed. The default behavior already checks if the output is a console or not, so forcing colorization is not recommended.

Default stream for console output

In Foxy and later, the output from all debug levels goes to stderr by default. It is possible to force all output to go to stdout by setting the `RCUTILS_LOGGING_USE_STDOUT` environment variable to 1. For example:

Linux

```bash
export RCUTILS_LOGGING_USE_STDOUT=1
```

macOS

```bash
export RCUTILS_LOGGING_USE_STDOUT=1
```

Windows

```bash
set "RCUTILS_LOGGING_USE_STDOUT=1"
```

Line buffered console output

By default, all logging output is unbuffered. You can force it to be buffered by setting the `RCUTILS_LOGGING_BUFFERED_STREAM` environment variable to 1. For example:

Linux

```bash
export RCUTILS_LOGGING_BUFFERED_STREAM=1
```

macOS
export RCUTILS_LOGGING_BUFFERED_STREAM=1

Windows

set "RCUTILS_LOGGING_BUFFERED_STREAM=1"

Then run:

ros2 run logging_demo logging_demo_main

Creating a content filtering subscription

Goal: Create a content filtering subscription.

Tutorial level: Advanced

Time: 15 minutes

Table of Contents

- Overview
- RMW Support
  - Installing the demo
  - Temperature filtering demo
- Related content

Overview

ROS 2 applications typically consist of topics to transmit data from publishers to subscriptions. Basically, subscriptions receive all published data from publishers on the topic. But sometimes, a subscription might be interested in only a subset of the data which is being sent by publishers. A content filtering subscription allows to receive only the data of interest for the application.

In this demo, we’ll be highlighting how to create a content filtering subscription and how they work.

RMW Support

Content filtering subscriptions require RMW implementation support.

<table>
<thead>
<tr>
<th>RMW</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>rmw_fastrtps</td>
<td>supported</td>
</tr>
<tr>
<td>rmw_connextdds</td>
<td>supported</td>
</tr>
<tr>
<td>rmw_cyclonedds</td>
<td>not supported</td>
</tr>
</tbody>
</table>

Currently all RMW implementations that support content filtering subscriptions are DDS based. That means that the supported filtering expressions and parameters are also dependent on DDS, you can refer to DDS specification Annex B - Syntax for Queries and Filters for details.
Installing the demo

See the *installation instructions* for details on installing ROS 2.

If you’ve installed ROS 2 from packages, ensure that you have `ros-iron-demo-nodes-cpp` installed. If you downloaded the archive or built ROS 2 from source, it will already be part of the installation.

Temperature filtering demo

This demo shows how a content filtering subscription can be used to only receive temperature values that are out of the acceptable temperature range, detecting emergencies. The content filtering subscription filters out the uninteresting temperature data, so that the subscription callback is not issued.

ContentFilteringPublisher:

https://github.com/ros2/demos/blob/iron/demo_nodes_cpp/src/topics/content_filtering_publisher.cpp

```cpp
#include <chrono>
#include <cstdio>
#include <memory>
#include <utility>
#include "rclcpp/rclcpp.hpp"
#include "rclcpp_components/register_node_macro.hpp"
#include "std_msgs/msg/float32.hpp"
#include "demo_nodes_cpp/visibility_control.h"

using namespace std
::chrono_literals;

namespace demo_nodes_cpp
{
  // The simulated temperature data starts from -100.0 and ends at 150.0 with a step size of 10.0
  constexpr std::array<float, 3> TEMPERATURE_SETTING = {-100.0f, 150.0f, 10.0f};

  // Create a ContentFilteringPublisher class that subclasses the generic rclcpp::Node base class.
  // The main function below will instantiate the class as a ROS node.
  class ContentFilteringPublisher : public rclcpp::Node
  {
    public:
      DEMO_NODES_CPP_PUBLIC
      explicit ContentFilteringPublisher(const rclcpp::NodeOptions & options)
        : Node("content_filtering_publisher", options)
      {
        // Create a function for when messages are to be sent.
        setvbuf(stdout, NULL, _IONBF, BUFSIZ);
        auto publish_message =
        [this]() -> void
        {
          msg_ = std::make_unique<std_msgs::msg::Float32>();
          msg_->data = temperature_;
        }
```
temperature_ += TEMPERATURE_SETTING[2];
if (temperature_ > TEMPERATURE_SETTING[1]) {
    temperature_ = TEMPERATURE_SETTING[0];
}
RCLCPP_INFO(this->get_logger(), "Publishing: '%f'", msg_->data);
// Put the message into a queue to be processed by the middleware.
// This call is non-blocking.
pub_ -> publish(std::move(msg_));
};
// Create a publisher with a custom Quality of Service profile.
// Uniform initialization is suggested so it can be trivially changed to
// rclcpp::KeepAll{} if the user wishes.
// (rclcpp::KeepLast(7) -> rclcpp::KeepAll() fails to compile)
rclcpp::QoS qos(rclcpp::KeepLast(7));
pub_ = this->create_publisher<std_msgs::msg::Float32>("temperature", qos);
// Use a timer to schedule periodic message publishing.
timer_ = this->create_wall_timer(1s, publish_message);
}

private:
  float temperature_ = TEMPERATURE_SETTING[0];
  std::unique_ptr<std_msgs::msg::Float32> msg_;
  rclcpp::Publisher<std_msgs::msg::Float32>::SharedPtr pub_;
  rclcpp::TimerBase::SharedPtr timer_;
};
// namespace demo_nodes_cpp

The content filter is defined in the subscription side, publishers don’t need to be configured in any special way to allow content filtering. The ContentFilteringPublisher node publishes simulated temperature data starting from -100.0 and ending at 150.0 with a step size of 10.0 every second.

We can run the demo by running the ros2 run demo_nodes_cpp content_filtering_publisher executable (don’t forget to source the setup file first):

```
$ ros2 run demo_nodes_cpp content_filtering_publisher
[INFO] [1651094594.822753479] [content_filtering_publisher]: Publishing: '-100.000000'
[INFO] [1651094595.822723857] [content_filtering_publisher]: Publishing: '-90.000000'
[INFO] [1651094596.822752996] [content_filtering_publisher]: Publishing: '-80.000000'
[INFO] [1651094597.822752475] [content_filtering_publisher]: Publishing: '-70.000000'
[INFO] [1651094598.822721485] [content_filtering_publisher]: Publishing: '-60.000000'
[INFO] [1651094599.822696188] [content_filtering_publisher]: Publishing: '-50.000000'
[INFO] [1651094600.822699217] [content_filtering_publisher]: Publishing: '-40.000000'
[INFO] [1651094601.822744113] [content_filtering_publisher]: Publishing: '-30.000000'
[INFO] [1651094602.822694805] [content_filtering_publisher]: Publishing: '-20.000000'
[INFO] [1651094603.822735805] [content_filtering_publisher]: Publishing: '-10.000000'
[INFO] [1651094604.822722094] [content_filtering_publisher]: Publishing: '0.000000'
[INFO] [1651094605.822699960] [content_filtering_publisher]: Publishing: '10.000000'
[INFO] [1651094606.822748946] [content_filtering_publisher]: Publishing: '20.000000'
[INFO] [1651094607.822694017] [content_filtering_publisher]: Publishing: '30.000000'
[INFO] [1651094608.822788798] [content_filtering_publisher]: Publishing: '40.000000'
[INFO] [1651094609.822692417] [content_filtering_publisher]: Publishing: '50.000000'
```
ContentFilteringSubscriber:

https://github.com/ros2/demos/blob/iron/demo_nodes_cpp/src/topics/content_filtering_subscriber.cpp

```cpp
#include "rclcpp/rclcpp.hpp"
#include "rclcpp_components/register_node_macro.hpp"
#include "rcpputils/join.hpp"

#include "std_msgs/msg/float32.hpp"
#include "demo_nodes_cpp/visibility_control.h"

namespace demo_nodes_cpp {
    "DemoNodesCpp_Public"
    explicit ContentFilteringSubscriber(const rclcpp::NodeOptions & options) : Node("content_filtering_subscriber", options) {
        setvbuf(stdout, NULL, _IONBF, BUFSIZ);
        // Create a callback function for when messages are received.
        auto callback = [this](const std_msgs::msg::Float32 & msg) -> void {
            if (msg.data < EMERGENCY_TEMPERATURE[0] || msg.data > EMERGENCY_TEMPERATURE[1]) {
                RCLCPP_INFO(this->get_logger(), "I receive an emergency temperature data: [%f]", msg.data);
            } else {
                RCLCPP_INFO(this->get_logger(), "I receive a temperature data: [%f]", msg.data);
            }
        };
    }
};
```

(continues on next page)
To enable content filtering, applications can set the filtering expression and the expression parameters in `SubscriptionOptions`. The application can also check if content filtering is enabled on the subscription.

In this demo, the `ContentFilteringSubscriber` node creates a content filtering subscription that receives a message only if the temperature value is less than -30.0 or greater than 100.0.

As commented before, content filtering subscription support depends on the RMW implementation. Applications can use the `is_cft_enabled` method to check if content filtering is actually enabled on the subscription.

To test content filtering subscription, let’s run it:

```
$ ros2 run demo_nodes_cpp content_filtering_subscriber
[INFO] [1651094590.682660703] [content_filtering_subscriber]: subscribed to topic "/
---temperature" with content filter options "data < %0 OR data > %1, {-30.000000, 100.
---000000}"
[INFO] [1651094590.682660703] [content_filtering_subscriber]: subscribed to topic "/
---temperature" with content filter options "data < %0 OR data > %1, {-30.000000, 100.
---000000}"
[INFO] [1651094594.823805294] [content_filtering_subscriber]: I receive an emergency...
---temperature data: [-90.000000]
[INFO] [1651094595.823419993] [content_filtering_subscriber]: I receive an emergency...
---temperature data: [-90.000000]
```

(continues on next page)
You should see a message showing the content filtering options used and logs for each message received only if the temperature value is less than -30.0 or greater than 100.0.

If content filtering is not supported by the RMW implementation, the subscription will still be created without content filtering enabled. We can try that by executing $\text{RMW\_IMPLEMENTATION=}\text{rmw\_cyclonedds\_cpp}$ $\text{ros2 run demo\_nodes\_cpp content\_filtering\_subscriber}$.

```
$ \text{RMW\_IMPLEMENTATION=}\text{rmw\_cyclonedds\_cpp} \text{ros2 run demo\_nodes\_cpp content\_filtering\_subscriber}

[WARN] [1651096637.893842072] [content\_filtering\_subscriber]: Content filter is not enabled since it is not supported
```

(continues on next page)
You can see the message *Content filter is not enabled* because underlying RMW implementation does not support the feature, but the demo still successfully creates the normal subscription to receive all temperature data.

**Related content**

- content filtering examples that covers all interfaces for content filtering subscription.
- content filtering design PR

**External resources**

- Bridging communication between ROS 1 and ROS 2
- Motion planning for a MoveIt 2 arm
- Using Turtlebot 3 (community-contributed)
  - Getting started
  - Simulating
  - Navigating in simulation
  - Learning SLAM in simulation
About

This article describes how to get ROS2 running on IBM Cloud using Docker files. It first gives a brief overview of docker images and how they work locally and then explores IBM Cloud and how the user can deploy their containers on it. Afterwards, a short description of how the user can use their own custom packages for ROS2 from github on IBM Cloud is provided. A walkthrough of how to create a cluster and utilize Kubernetes on IBM Cloud is provided and finally the Docker image is deployed on the cluster. Originally published here and here.

ROS2 on IBM Cloud

In this tutorial, we show how you can easily integrate and run ROS2 on IBM Cloud with your custom packages.

ROS2 is the new generation of ROS which gives more control over multi-robot formations. With the advancements of cloud computing, cloud robotics are becoming more important in today’s age. In this tutorial, we will go through a short introduction on running ROS2 on IBM Cloud. By the end of the tutorial, you will be able to create your own packages in ROS2 and deploy them to the cloud using docker files.

The following instructions assume you’re using Linux and have been tested with Ubuntu 18.04 (Bionic Beaver).
Step 1: Setting up your system

Before we go into how the exact process works, let’s first make sure all the required software is properly installed. We’ll point you towards the appropriate sources to set up your system and only highlight the details that pertain to our use-case.

a) Docker files?

Docker files are a form of containers that can run separate from your system, this way, you can set-up potentially hundreds of different projects without affecting one another. You can even set-up different versions of Linux on one machine, without the need for virtual machine. Docker files have an advantage of saving space and only utilizing your system resources when running. In addition, dockers are versatile and transferable. They contain all the required prerequisites to run separately, meaning that you can easily use a docker file for a specific system or service without any cumbersome steps!

Excited yet? Let’s start off by installing docker to your system by following the following link. From the tutorial, you should have done some sanity checks to make sure docker is properly set-up. Just in case, however, let’s run the following command once again that uses the hello-world docker image:

```
$ sudo docker run hello-world
```

You should obtain the following output:

```
Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub. (amd64)
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/
```

For more examples and ideas, visit:
https://docs.docker.com/get-started/
b) ROS2 Image

ROS announced image containers for several ROS distributions in January 2019. More detailed instructions on the use of ROS2 docker images can be found here.

Let’s skip through that and get to real-deal right away; creating a local ROS2 docker. We’ll create our own Dockerfile (instead of using a ready Image) since we’ll need this method for deployment on IBM Cloud. First, we create a new directory which will hold our Dockerfile and any other files we need later on and navigate to it. Using your favorite $EDITOR of choice, open a new file named Dockerfile (make sure the file naming is correct):

```
$ mkdir ~/ros2_docker
$ cd ~/ros2_docker
$ $EDITOR Dockerfile
```

Insert the following in the Dockerfile, and save it (also found here):

```
FROM ros:foxy

# install ros package
RUN apt-get update && apt-get install -y \
    ros-${ROS_DISTRO}-demo-nodes-cpp \
    ros-${ROS_DISTRO}-demo-nodes-py && \
    rm -rf /var/lib/apt/lists/* && mkdir /ros2_home

WORKDIR /ros2_home

# launch ros package
CMD ["ros2", "launch", "demo_nodes_cpp", "talker_listener_launch.py"]
```

- `FROM`: creates a layer from the ros:foxy Docker image
- `RUN`: builds your container by installing vim into it and creating a directory called /ros2_home
- `WORKDIR`: informs the container where the working directory should be for it

Of course, you are free to change the ROS distribution (foxy is used here) or change the directory name. The above docker file sets up ROS-foxy and installs the demo nodes for C++ and Python. Then it launches a file which runs a talker and a listener node. We will see it in action in just a few, but they act very similar to the publisher-subscriber example found in the ROS wiki

Now, we are ready to build the docker image to run ROS2 in it (yes, it is THAT easy!).

**Note**: if you have errors due to insufficient privileges or permission denied, try running the command with sudo privileges:

```
$ docker build .

# You will see a bunch of lines that execute the docker file instructions followed by:
Successfully built 0dc6ce7cb487
```

0dc6ce7cb487 will most probably be different for you, so keep note of it and copy it somewhere for reference. You can always go back and check the docker images you have on your system using:

```
$ sudo docker ps -as
```
Now, run the docker file using:

```bash
$ docker run -it 0dc6ce7cb487
[INFO] [launch]: All log files can be found below /root/.ros/log/2020-10-28-02-41-45-177546-0b5d9ed123be-1
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [talker-1]: process started with pid [28]
[INFO] [listener-2]: process started with pid [30]
[talker-1] [INFO] [1603852907.249886590] [talker]: Publishing: 'Hello World: 1'
[listener-2] [INFO] [1603852907.249886590] [listener]: I heard: [Hello World: 1]
[talker-1] [INFO] [1603852908.249786312] [talker]: Publishing: 'Hello World: 2'
[listener-2] [INFO] [1603852908.250453386] [listener]: I heard: [Hello World: 2]
[talker-1] [INFO] [1603852909.249845718] [talker]: Publishing: 'Hello World: 3'
[listener-2] [INFO] [1603852909.250152324] [listener]: I heard: [Hello World: 3]
[talker-1] [INFO] [1603852910.249845718] [talker]: Publishing: 'Hello World: 4'
[listener-2] [INFO] [1603852910.250509355] [listener]: I heard: [Hello World: 4]
[talker-1] [INFO] [1603852911.249506058] [talker]: Publishing: 'Hello World: 5'
[listener-2] [INFO] [1603852911.250152324] [listener]: I heard: [Hello World: 5]
[talker-1] [INFO] [1603852912.24956670] [talker]: Publishing: 'Hello World: 6'
[listener-2] [INFO] [1603852912.250212678] [listener]: I heard: [Hello World: 6]
```

If it works correctly, you should see something similar to what is shown above. As can be seen, there are two ROS nodes (a publisher and a subscriber) running and their output is provided to us through ROS INFO.

### Step 2: Running the image on IBM Cloud

The following steps assume you have an IBM cloud account and have ibmcloud CLI installed. If not, please check this link out to get that done first.

We also need to make sure that the CLI plug-in for the IBM Cloud Container Registry is installed by running the command

```bash
$ ibmcloud plugin install container-registry
```

Afterwards, login to your ibmcloud account through the terminal:

```bash
$ ibmcloud login --sso
```

From here, let’s create a container registry name-space. Make sure you use a unique name that is also descriptive as to what it is. Here, I used `ros2nasr`.

```bash
$ ibmcloud cr namespace-add ros2nasr
```

IBM cloud has a lot of shortcuts that would help us get our container onto the cloud right away. The command below builds the container and tags it with the name `ros2foxy` and the version of 1. Make sure you use the correct registry name you created and you are free to change the container name as you wish. The . at the end indicates that the `Dockerfile` is in the current directory (and it is important), if not, change it to point to the directory containing the `Dockerfile`.

```bash
$ ibmcloud cr build --tag registry.bluemix.net/ros2nasr/ros2foxy:1 .
```

You can now make sure that the container has been pushed to the registry you created by running the following command
$ ibmcloud cr image-list  
Listing images...  

<table>
<thead>
<tr>
<th>REPOSITORY</th>
<th>TAG</th>
<th>DIGEST</th>
<th>NAMESPACE</th>
<th>CREATED</th>
<th>SIZE</th>
<th>SECURITY STATUS</th>
<th>STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>us.icr.io/ros2nasr/ros2foxy</td>
<td>1</td>
<td>031be29301e6</td>
<td>ros2nasr</td>
<td>36 seconds ago</td>
<td>120 MB</td>
<td>No Issues</td>
<td>OK</td>
</tr>
</tbody>
</table>

Next, it is important to log-in to your registry to run the docker image. Again, if you face a permission denied error, perform the command with sudo privileges. Afterwards, run your docker file as shown below.

$ ibmcloud cr login  
Logging in to 'registry.ng.bluemix.net'...  
Logged in to 'registry.ng.bluemix.net'.  
Logging in to 'us.icr.io'...  
Logged in to 'us.icr.io'.  

OK  

$ docker run -v -it registry.ng.bluemix.net/ros2nasr/ros2foxy:1  

Where ros2nasr is the name of the registry you created and ros2foxy:1 is the tag of the docker container and the version as explained previously.

You should now see your docker file running and providing similar output to that you saw when you ran it locally on your machine.

**Step 3: Using Custom ROS2 Packages**

So now we have the full pipeline working, from creating the Dockerfile, all the way to deploying it and seeing it work on IBM Cloud. But, what if we want to use a custom set of packages we (or someone else) created?

Well that all has to do with how you set-up your Dockerfile. Let’s use the example provided by ROS2 here. Create a new directory with a new Dockerfile (or overwrite the existing one) and add the following in it (or download the file here)

```bash
ARG FROM_IMAGE=ros:foxy
ARG OVERLAY_WS=/opt/ros/overlay_ws

# multi-stage for caching
FROM $FROM_IMAGE AS cacher

# clone overlay source
ARG OVERLAY_WS
WORKDIR $OVERLAY_WS/src
RUN echo "\n repositories:  
   ros2/demos:  
      type: git  
      url: https://github.com/ros2/demos.git  
      version: $({ROS_DISTRO})  
"
```

(continues on next page)
Going through the lines shown, we can see how we can add custom packages from github in 4 steps:

1. Create an overlay with custom packages cloned from Github:

ARG OVERLAY_WS
WORKDIR $OVERLAY_WS/src
RUN echo 

repositories: 

Going through the lines shown, we can see how we can add custom packages from github in 4 steps:

1. Create an overlay with custom packages cloned from Github:
ros2/demos:
  type: git
  url: https://github.com/ros2/demos.git
  version: ${ROS_DISTRO}
" > ../overlay.repos
RUN vcs import ./ < ../overlay.repos

2. Install package dependencies using rosdep

```bash
# install overlay dependencies
ARG OVERLAY_WS
WORKDIR $OVERLAY_WS
COPY --from=cacher /tmp/$OVERLAY_WS/src ./src
RUN . /opt/ros/$ROS_DISTRO/setup.sh && \
    apt-get update && rosdep install -y \
    --from-paths \
    src/ros2/demos/demo_nodes_cpp \
    src/ros2/demos/demo_nodes_py \
    --ignore-src \
    && rm -rf /var/lib/apt/lists/*
```

3. Build the packages you need

```bash
# build overlay source
COPY --from=cacher $OVERLAY_WS/src ./src
ARG OVERLAY_MIXINS="release"
RUN . /opt/ros/$ROS_DISTRO/setup.sh && \
    colcon build \
    --packages-select \
    demo_nodes_cpp \
    demo_nodes_py \
    --mixin $OVERLAY_MIXINS
```

4. Running the launch file

```bash
# run launch file
CMD ["ros2", "launch", "demo_nodes_cpp", "talker_listener_launch.py"]
```

Likewise, we can change the packages used, install their dependencies, and then run them.

**Back to IBM Cloud**

With this Dockerfile, we can follow the same steps we did before to deploy it on IBM Cloud. Since we already have our registry created, and we’re logged in to IBM Cloud, we directly build our new Dockerfile. Notice how I kept the tag the same but changed the version, this way I can update the docker image created previously. (You are free to create a completely new one if you want)

```
$ ibmcloud cr build --tag registry.bluemix.net/ros2nasr/ros2foxy:2 .
```

Then, make sure you are logged in to the registry and run the new docker image:

```
$ ibmcloud cr login
Logging in to 'registry.ng.bluemix.net'...
Logged in to 'registry.ng.bluemix.net'.
```

(continues on next page)
Logging in to 'us.icr.io'...
Logged in to 'us.icr.io'.

OK

$ docker run -v -it registry.ng.bluemix.net/ros2nasr/ros2foxy:2

You should see, again, the same output. However, this time we did it through custom packages from github, which allows us to utilize our personally created packages for ROS2 on IBM Cloud.

**Extra: Deleting Docker Images**

As you may find yourself in need of deleting a specific docker image(s) from IBM Cloud, this is how you should go about it!

1. List all the images you have and find all the ones that share the *IMAGE* name corresponding to *registry.ng.bluemix.net/ros2nasr/ros2foxy:2* (in my case). Then delete them using their *NAMES*

```
$ docker rm your_docker_NAMES
```

2. Delete the docker image from IBM Cloud using its *IMAGE* name

```
$ docker rmi registry.ng.bluemix.net/ros2nasr/ros2foxy:2
```

**Step 4: Kubernetes**

**a) Creating the Cluster**

Create a cluster using the Console. The instructions are found [here](#). The settings used are detailed below. These are merely suggestions and can be changed if you need to. However, make sure you understand the implications of your choices:

1. Plan: *Standard*
2. Orchestration Service: *Kubernetes v1.18.10*
3. Infrastructure: *Classic*
4. Location:
   - Resource group: *Default*
   - Geography: *North America* (you are free to change this)
   - Availability: *Single zone* (you are free to change this but make sure you understand the impact of your choices by checking the IBM Cloud documentation.)
   - Worker Zone: *Toronto 01* (choose the location that is physically closest to you)
5. Worker Pool:
   - Virtual - shared, Ubuntu 18
   - Memory: 16 GB
   - Worker nodes per zone: 1
6. Master service endpoint: *Both private & public endpoints*

7. Resource details (Totally flexible):
   - Cluster name: `mycluster-tor01-rosibm`
   - Tags: `version:1`

After you create your cluster, you will be redirected to a page which details how you can set up the CLI tools and access your cluster. Please follow these instructions (or check the instructions here) and wait for the progress bar to show that the worker nodes you created are ready by indicating *Normal* next to the cluster name. You can also reach this screen from the IBM Cloud Console inside the Kubernetes.

**b) Deploying your Docker Image** *Finally!*

1. Create a deployment configuration yaml file named `ros2-deployment.yaml` using your favorite $EDITOR and insert the following in it:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: <deployment>
spec:
  replicas: <number_of_replicas>
  selector:
    matchLabels:
      app: <app_name>
  template:
    metadata:
      labels:
        app: <app_name>
    spec:
      containers:
      - name: <app_name>
        image: <region>.icr.io/<namespace>/<image>:<tag>
```

You should replace the tags shown between “<” “>” as described here. The file in my case would look something like this:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ros2-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ros2-ibmcloud
  template:
    metadata:
      labels:
        app: ros2-ibmcloud
    spec:
      containers:
```

(continues on next page)
- name: ros2-ibmcloud
  image: us.icr.io/ros2nasr/ros2foxy:2

Deploy the file using the following command

```
$ kubectl apply -f ros2-deployment.yaml
deployment.apps/ros2-deployment created
```

Now your docker image is fully deployed on your cluster!

**Step 5: Using CLI for your Docker Image**

1. Navigate to your cluster through the IBM Cloud console Kubernetes.
2. Click on Kubernetes dashboard on the top right corner of the page.

You should now be able to see a full list of all the different parameters of your cluster as well as its CPU and Memory Usage.

3. Navigate to Pods and click on your deployment.
4. On the top right corner, click on Exec into pod

Now you are inside your docker image! You can source your workspace (if needed) and run ROS2! For example:

```
root@ros2-deployment-xxxxxxxx:/opt/ros/overlay_ws# . install/setup.sh
root@ros2-deployment-xxxxxxxx:/opt/ros/overlay_ws# ros2 launch demo_nodes_cpp talker_listener_launch.py
```

**Final Remarks**

At this point, you are capable of creating your own docker image using ROS2 packages on github. It is also possible, with little changes to utilize local ROS2 packages as well. This could be the topic of another article. However, you are encouraged to check out the following Dockerfile which uses a local copy of the demos repository. Similarly, you can use your own local package.

**Using Eclipse Oxygen with rviz2 [community-contributed]**

Table of Contents

- Setup
- Eclipse-indexer
- Debugging with eclipse
Setup

This tutorial assumes Eclipse Oxygen, git, and Egit (http://www.eclipse.org/egit/download/) are already installed. Throughout the tutorial we name the eclipse workspace the same name as the ros2 package, but this is not required. HINT: We use nested projects and one Eclipse Workspace for each ROS-2 package.

Create a C++ Project.
Choose the ROS 2 package name as the Project Name. Choose a Makefile Project and Other Toolchain.
Click on Finish
Our project should be shown in the “Project Explorer”.

4.8. ROS 2 Documentation
Inside our Project create a folder called “src”.

Vulcanexus Documentation, Release 1.0.0

Chapter 4. Contributing to the documentation
Import a git repository.
Put in the repository URL.
IMPORTANT: Use the source folder of the project we created before as the destination folder.

HINT: If you ran into problems choosing the destination folder path, the Eclipse Dialog needs a name in the name field.
Import using the new project wizard.
Create a General->Project.
Use the git repository name as the project name. IMPORTANT: Use the folder we cloned the git repository in as the “Location”.
The git project and the new project should be visible in the Project Explorer view. The same files are listed multiple times, but only one project is linked with Egit.
Repeat this procedure again. Import git repository pluginlib.
IMPORTANT: Use a folder inside the source folder as “Destination->Directory”.
IMPORTANT: Use the folder we cloned the git repository in as the location for the new project.
Run the same procedure with the tinyxml2_vendor git repository.
IMPORTANT: Again use a folder inside the source folder.
IMPORTANT: Use the location of the folder we cloned as the new project folder.
Now all four Projects should be visible in the Project Explorer view.
Clicking in the top right corner for the Project Explorer view allows us to change the Project Presentation to Hierarchical view. Now it looks like a ROS-2 project as it is on the hard drive. But this view loses the linkage to Egit, so use the Flat Project Presentation. The Egit linkage is good if you want to see e.g. which author wrote which code-line, etc.

Go to “C/C++ build”-section and put “ament” into “Build command”.

4.8. ROS 2 Documentation
Go to “Behavior” tab and unselect “clean” and put “build” into Build textbox.
Before “Build project” will work, we need to close Eclipse. Open a shell and source the ROS-2 setup.bash file, then cd into the directory of the eclipse project (here: /home/ubuntu/rviz2_ws/rviz2_ws) and start Eclipse from inside this directory.
Now code completion, egit annotations, eclipse C/C++ Tools, etc. should all work.

Eclipse-indexer

Opening the main.cpp of rviz2 may show a lot of “unresolved inclusion” warnings. To fix this, go to Project->Properties->C++ General->Path and Symbols. Click on the “References” tab and select “ros2_ws”.
Go to C/C++ General -> Path-and-Symbols, click on the “Source locations” tab and click on “Link folder”. Choose the location of qt5 includes.

The next image should be shown. It is a good idea to add excludes to the source locations, so that some directories (like “Build” and “Install”) don’t get indexed.

Go to C++ General -> Preprocessor includes, select “CDT GCC Built in compiler settings [Shared]” and enter in the “command to get compiler specs” text box the following:
Go to “C/C++-General->Indexer” and select the following in the image. E.g “index unused headers as c files” to resolve e.g. QApplication, because the QApplication headers content is only “#include “qapplication.h”.”
After running the indexer (which happens later, so you will see this also later), you can see what it added.

After that right-click on the rviz2 project and select “Indexer->Rebuild”, which will start rebuilding the index (there is an icon in the lower right showing progress). Once the index is finished rebuilding, it should be able to resolve all...
Debugging with eclipse

Go to “C/C++-Build” and add to the build command:

-D_CMAKE_BUILD_TYPE=Debug
Then in eclipse go to “Run->Debug Configurations” and add the following and click on “Debug”.

```
build-cmake-args "DCMAKE_BUILD_TYPE=Debug"
```

Configure Workspace Settings...
Building a real-time Linux kernel [community-contributed]

This tutorial begins with a clean Ubuntu 20.04.1 install on Intel x86_64. Actual kernel is 5.4.0-54-generic, but we will install the Latest Stable RT_PREEMPT Version. To build the kernel you need at least 30GB free disk space.

Check https://wiki.linuxfoundation.org/realtime/start for the latest stable version, at the time of writing this is “Latest Stable Version 5.4-rt”. If we click on the link, we get the exact version. Currently it is patch-5.4.78-rt44.patch.gz.

We create a directory in our home dir with

```
mkdir ~/kernel
```

and switch into it with

```
cd ~/kernel
```

We can go with a browser to https://mirrors.edge.kernel.org/pub/linux/kernel/v5.x/ and see if the version is there. You can download it from the site and move it manually from /Downloads to the /kernel folder, or download it using wget by right clicking the link using “copy link location”. Example:

```
wget https://mirrors.edge.kernel.org/pub/linux/kernel/v5.x/linux-5.4.78.tar.gz
```

unpack it with

```
tar -xzf linux-5.4.78.tar.gz
```

download rt_preempt patch matching the Kernel version we just downloaded over at http://cdn.kernel.org/pub/linux/kernel/projects/rt/5.4/
wget http://cdn.kernel.org/pub/linux/kernel/projects/rt/5.4/older/patch-5.4.78-rt44.-patch.gz

unpack it with

```
gunzip patch-5.4.78-rt44.patch.gz
```

Then switch into the linux directory with

```
cd linux-5.4.78/
```

and patch the kernel with the realtime patch

```
patch -p1 < ../patch-5.4.78-rt44.patch
```

We simply want to use the config of our Ubuntu installation, so we get the Ubuntu config with

```
cp /boot/config-5.4.0-54-generic .config
```

Open Software & Updates. in the Ubuntu Software menu tick the ‘Source code’ box

We need some tools to build kernel, install them with

```
sudo apt-get build-dep linux
sudo apt-get install libncurses-dev flex bison openssl libssl-dev dkms libelf-dev
   libudev-dev libpci-dev libiberty-dev autoconf fakeroot
```

To enable all Ubuntu configurations, we simply use

```
yes '' | make oldconfig
```

Then we need to enable rt_preempt in the kernel. We call

```
make menuconfig
```

and set the following

```
# Enable CONFIG_PREEMPT_RT
   -> General Setup
      -> Preemption Model (Fully Preemptible Kernel (Real-Time))
         (X) Fully Preemptible Kernel (Real-Time)

# Enable CONFIG_HIGH_RES_TIMERS
   -> General setup
      -> Timers subsystem
         [*] High Resolution Timer Support

# Enable CONFIG_NO_HZ_FULL
   -> General setup
      -> Timers subsystem
         -> Timer tick handling (Full dynticks system (tickless))
            (X) Full dynticks system (tickless)

# Set CONFIG_HZ_1000 (note: this is no longer in the General Setup menu, go back twice)
   -> Processor type and features
```

(continues on next page)
Save and exit menuconfig. Now we're going to build the kernel which will take quite some time. (10-30min on a modern cpu)

```
make -j `nproc` deb-pkg
```

After the build is finished check the debian packages

```
ls ../*deb
../linux-headers-5.4.78-rt41_5.4.78-rt44-1_amd64.deb ../linux-image-5.4.78-rt44-dbg_5.4.78-rt44-1_amd64.deb
../linux-image-5.4.78-rt41_5.4.78-rt44-1_amd64.deb ../linux-libc-dev_5.4.78-rt44-1_amd64.deb
```

Then we install all kernel debian packages

```
sudo dpkg -i ../*.deb
```

Now the real time kernel should be installed. Reboot the system and check the new kernel version

```
sudo reboot
uname -a
Linux ros2host 5.4.78-rt44 #1 SMP PREEMPT_RT Fri Nov 6 10:37:59 CET 2020 x86_64 xx
```

**Building a package with Eclipse 2021-06**

You cannot create a ROS 2 package with eclipse, you need to create it with commandline tools. Follow the Create a package tutorial.

After you created your project, you can edit the source code and build it with eclipse.

We start eclipse and select a eclipse-workspace.
We create a C++ project
We see that we got C++ includes.
We now import our ROS 2 project. The code is still in the old place.
Click the Advanced in the Options and check the **Create links in workspace**.
We see in the source code that the C++ includes got resolved but not the ROS 2 ones.
Add include paths of needed packages. (e.g. /opt/ros/iron/include/rclcpp, /opt/ros/iron/include/std_msgs, etc.)
We now see that the ROS 2 includes got resolved too.

Adding Builder colcon, so that we can build with right-click on project and “Build project”.

4.8. ROS 2 Documentation
With `PYTHONPATH` you can also build python projects.
Chapter 4. Contributing to the documentation
Right-click on the project and select “Build Project”.

```cpp
#include "rcpp/rclcpp.h"
#include "std_msgs/msg/string.hpp"
using namespace std::chrono_literals;

25/* This example creates a subclass of Node and uses std::bind() to register a
26   member function as a callback from the timer. */
27
28
29class MinimalPublisher : public rclcpp::Node
30{
31    public:
32       MinimalPublisher() : Node("minimal_publisher"), count(0) {
33           // initializer list
34           publisher = this->create_publisher<std_msgs::msg::String>("topic", 10);
35           timer = this->create_wall_timer(500ms, std::bind(&MinimalPublisher::timer_callback, this));
36
37    private:
38
39    public:
40    void timer_callback(const std::chrono::milliseconds elapsed)
41    {
42        this->publish_string();
43    }
44
45    private:
46    rclcpp::Publisher< std::string >::SharedPtr publisher;
47    rclcpp::TimerBase::SharedPtr timer;
48
49    int count;
50
51    };
```

Starting `my_node` (Program `my_node`) at 19,28-16 AM - 9:23 AM (pid: 72968)
Finished `my_node` (Program `my_node`) at 19,28-16 AM - 9:23 AM (pid: 72968)
Summary: 1 package finished [6.18s]
Examples

- Python and C++ minimal examples.

4.8.4 How-to Guides

How-to Guides provide direct and modular answers to “How-to” questions regarding key aspects of ROS 2. They contain succinct steps to help you accomplish important tasks quickly.

How-to Guides are meant for users who already have some knowledge of ROS 2 and just want to find out how to implement something specific. They will not go in-depth by providing background information or teaching how a concept ties into the greater ROS 2 ecosystem.

If you are new and looking to learn the ropes, start with the Tutorials for a more well-rounded progression through ROS 2.

Installation troubleshooting

Troubleshooting techniques for installation are sorted by the platforms they apply to.

<table>
<thead>
<tr>
<th>Platforms</th>
</tr>
</thead>
<tbody>
<tr>
<td>- General</td>
</tr>
<tr>
<td>- Linux</td>
</tr>
<tr>
<td>- macOS</td>
</tr>
<tr>
<td>- Windows</td>
</tr>
</tbody>
</table>

General

General troubleshooting techniques apply to all platforms.

Enable multicast

In order to communicate successfully via DDS, the used network interface has to be multicast enabled. We’ve seen in past experiences that this might not necessarily be enabled by default (on Ubuntu or OSX) when using the loopback adapter. See the original issue or a conversation on ros-answers. You can verify that your current setup allows multicast with the ROS 2 tool:

In Terminal 1:
```
ros2 multicast receive
```

In Terminal 2:
```
ros2 multicast send
```

If the first command did not return a response similar to:

```
Received from xx.xxx.xxx.xx:43751: 'Hello World!'`
```
then you will need to update your firewall configuration to allow multicast using `ufw`.

```bash
sudo ufw allow in proto udp to 224.0.0.0/4
sudo ufw allow in proto udp from 224.0.0.0/4
```

You can check if the multicast flag is enabled for your network interface using the `ifconfig` tool and looking for MULTICAST in the flags section:

```
en0: flags=4163<...,MULTICAST>
...```

### Import failing without library present on the system

Sometimes `rclpy` fails to be imported because the expected C extension libraries are not found. If so, compare the libraries present in the directory with the one mentioned in the error message. Assuming a file with a similar name exists (same prefix like `_rclpy_.` and same suffix like `.so` but a different Python version / architecture) you are using a different Python interpreter than which was used to build the C extension. Be sure to use the same Python interpreter as the one used to build the binary.

For example, such a mismatch can crop up after an update of the OS. Then, rebuilding the workspace may fix the issue.

### Linux

#### Internal compiler error

If you experience an ICE when trying to compile on a memory constrained platform like a Raspberry PI you might want to build single threaded (prefix the build invocation with `MAKEFLAGS=-j1`).

#### Out of memory

The `ros1_bridge` in its current form requires 4Gb of free RAM to compile. If you don’t have that amount of RAM available it’s suggested to use `COLCON_IGNORE` in that folder and skip its compilation.

#### Multiple host interference

If you’re running multiple instances on the same network you may get interference. To avoid this you can set the environment variable `ROS_DOMAIN_ID` to a different integer, the default is zero. This will define the DDS domain id for your system.

#### Exception sourcing setup.bash

If you encounter exceptions when trying to source the environment after building from source, try to upgrade `colcon` related packages using

```
colcon version-check  # check if newer versions available
sudo apt install python3-colcon* --only-upgrade  # upgrade installed colcon packages to latest version
```
macOS

Segmentation fault when using pyenv

pyenv seems to default to building Python with .a files, but that causes issues with rclpy, so it’s recommended to build Python with Frameworks enabled on macOS when using pyenv:
https://github.com/pyenv/pyenv/wiki#how-to-build-cpython-with-framework-support-on-os-x

Library not loaded; image not found

If you are seeing library loading issues at runtime (either running tests or running nodes), such as the following:

```bash
--cpython-37m-darwin.so, 2): Library not loaded: @rpath/librcl_interfaces__rosidl__
typesupport_c.dylib
--cpython-37m-darwin.so
Reason: image not found
```

Then you probably have System Integrity Protection enabled. Follow these instructions to disable System Integrity Protection (SIP).

Qt build error: unknown type name 'Q_ENUM'

If you see build errors related to Qt, e.g.:

```bash
In file included from /usr/local/opt/qt/lib/QtGui.framework/Headers/qguiapplication.h:46:
/usr/local/opt/qt/lib/QtGui.framework/Headers/qinputmethod.h:87:5: error:
unknown type name 'Q_ENUM'
Q_ENUM(Action)
^
```

you may be using qt4 instead of qt5: see https://github.com/ros2/ros2/issues/441

Missing symbol when opencv (and therefore libjpeg, libtiff, and libpng) are installed with Homebrew

If you have opencv installed you might get this:

```bash
dyld: Symbol not found: __cg_jpeg_resync_to_restart
Referenced from: /System/Library/Frameworks/ImageIO.framework/Versions/A/ImageIO
Expected in: /usr/local/lib/libJPEG.dylib
in /System/Library/Frameworks/ImageIO.framework/Versions/A/ImageIO
/bin/sh: line 1: 25274 Trace/BPT trap: 5 /usr/local/bin/cmake
```

If so, to build you’ll have to do this:

```bash
$ brew unlink libpng libtiff libjpeg
```

But this will break opencv, so you’ll also need to update it to continue working:
$ sudo install_name_tool -change /usr/local/lib/libjpeg.8.dylib /usr/local/opt/jpeg/lib/libjpeg.8.dylib /usr/local/lib/libopencv_highgui.2.4.dylib
$ sudo install_name_tool -change /usr/local/lib/libpng16.16.dylib /usr/local/opt/libpng/lib/libpng16.16.dylib /usr/local/lib/libopencv_highgui.2.4.dylib
$ sudo install_name_tool -change /usr/local/lib/libtiff.5.dylib /usr/local/opt/libtiff/lib/libtiff.5.dylib /usr/local/lib/libopencv_highgui.2.4.dylib
$ sudo install_name_tool -change /usr/local/lib/libjpeg.8.dylib /usr/local/opt/jpeg/lib/libjpeg.8.dylib /usr/local/Cellar/libtiff/4.0.4/lib/libtiff.5.dylib

The first command is necessary to avoid things built against the system libjpeg (etc.) from getting the version in /usr/local/lib. The others are updating things built by Homebrew so that they can find the version of libjpeg (etc.) without having them in /usr/local/lib.

**Xcode-select error: tool xcodebuild requires Xcode, but active developer directory is a command line instance**

If you recently installed Xcode, you may encounter this error:

```
Xcode: xcode-select: error: tool 'xcodebuild' requires Xcode, but active developer directory '/Library/Developer/CommandLineTools' is a command line instance
```

To resolve this error, you will need to:

1. Double check that you have the command line tool installed:

   ```
   $ xcode-select --install
   ```

2. Accept the terms and conditions of Xcode by typing in terminal:

   ```
   $ sudo xcodebuild -license accept
   ```

3. Ensure Xcode app is in the /Applications directory (NOT /Users/{user}/Applications)

4. Point xcode-select to the Xcode app Developer directory using the following command:

   ```
   $ sudo xcode-select -s /Applications/Xcode.app/Contents/Developer
   ```

**qt_gui_cpp error: SIP binding generator NOT available**

When building qt_gui_cpp there may be errors look like the following:

```
--- stderr: qt_gui_cpp

CMake Error at src/CMakeLists.txt:10 (message):
  No Python binding generator found.
---
Failed <<< qt_gui_cpp [ Exited with code 1 ]
```

To fix this issue, follow these steps to install dependencies for RQt.
rosdep install error homebrew: Failed to detect successful installation of [qt5]

While following the Creating a workspace tutorial, you might encounter the following error stating that rosdep fails to install Qt5.

```
$ rosdep install -i --from-path src --rosdistro iron -y
executing command [brew install qt5]
Warning: qt 5.15.0 is already installed and up-to-date
To reinstall 5.15.0, run `brew reinstall qt`
ERROR: the following rosdeps failed to install
       homebrew: Failed to detect successful installation of [qt5]
```

This error seems to stem from a linking issue and can be resolved by running the following command.

```
$ cd /usr/local/Cellar
$ sudo ln -s qt qt5
```

Running the rosdep command should now execute normally:

```
$ rosdep install -i --from-path src --rosdistro iron -y
#All required rosdeps installed successfully
```

Windows

Import failing even with library present on the system

Sometimes rclpy fails to be imported because of some missing DLLs on your system. If so, make sure to install all the dependencies listed in the “Installing prerequisites” sections of the installation instructions).

If you are installing from binaries, you may need to update your dependencies: they must be the same version as those used to build the binaries.

If you are still having issues, you can use the Dependencies tool to determine which dependencies are missing on your system. Use the tool to load the corresponding .pyd file, and it should report unavailable DLL modules. Be sure that the current workspace is sourced before you execute the tool, otherwise there will be unresolved ROS DLL files. Use this information to install additional dependencies or adjust your path as necessary.

CMake error setting modification time

If you run into the CMake error file INSTALL cannot set modification time on ... when installing files it is likely that an anti virus software or Windows Defender are interfering with the build. E.g. for Windows Defender you can list the workspace location to be excluded to prevent it from scanning those files.
260 character path limit

The input line is too long.
The syntax of the command is incorrect.

Depending on your directory hierarchy, you may see path length limit errors when building ROS 2 from source or your own libraries.

To allow deeper path lengths:

Run `regedit.exe`, navigate to `Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\FileSystem`, and set `LongPathsEnabled` to 0x00000001 (1).

Hit the windows key and type `Edit Group Policy`. Navigate to `Local Computer Policy > Computer Configuration > Administrative Templates > System > Filesystem`. Right click `Enable Win32 long paths`, click `Edit`. In the dialog, select `Enabled` and click `OK`.

Close and open your terminal to reset the environment and try building again.

CMake packages unable to find asio, tinyxml2, tinyxml, or eigen

We’ve seen that sometimes the chocolatey packages for asio, tinyxml2, etc. do not add important registry entries and CMake will be unable to find them when building ROS 2. We’ve not yet been able to identify the root cause, but uninstalling the chocolatey packages (with `-n` if the uninstall fails the first time), and then reinstalling them will fix the issue.

patch.exe opens a new command window and asks for administrator

This will also cause the build of packages which need to use patch to fail, even you allow it to use administrator rights.

- `choco uninstall patch; colcon build --cmake-clean-cache` - This is a bug in the GNU Patch For Windows package. If this package is not installed, the build process will instead use the version of Patch distributed with git.

Failed to load Fast RTPS shared library

Fast RTPS requires `msvcr20.dll`, which is part of the Visual C++ Redistributable Packages for Visual Studio 2013. Although it is usually installed by default in Windows 10, we know that some Windows 10-like versions don’t have it installed by default (e.g.: Windows Server 2019). In case you don’t have it installed, you can download it from here.

Binary installation specific

- If your example does not start because of missing DLLs, please verify that all libraries from external dependencies such as OpenCV are located inside your PATH variable.
- If you forget to call the `local_setup.bat` file from your terminal, the demo programs will most likely crash immediately.
Developing a ROS 2 package

Table of Contents

- Prerequisites
- Creating a package
  - C++ Packages
  - Python Packages
  - Combined C++ and Python Packages

This tutorial will teach you how to create your first ROS 2 application. It is intended for developers who want to learn how to create custom packages in ROS 2, not for people who want to use ROS 2 with its existing packages.

Prerequisites

- Install ROS
  - Install colcon
  - Setup your workspace by sourcing your ROS 2 installation.

Creating a package

All ROS 2 packages begin by running the command

```
ros2 pkg create <pkg-name> --dependencies [deps]
```

in your workspace (usually ~/ros2_ws/src).

To create a package for a specific client library:

C++

```
ros2 pkg create <pkg-name> --dependencies [deps] --build-type ament_cmake
```

Python

```
ros2 pkg create <pkg-name> --dependencies [deps] --build-type ament_python
```

You can then update the package.xml with your package info such as dependencies, descriptions, and authorship.
C++ Packages

You will mostly use the add_executable() CMake macro along with

```
ament_target_dependencies(<executable-name> [dependencies])
```

to create executable nodes and link dependencies.

To install your launch files and nodes, you can use the install() macro placed towards the end of the file but before the ament_package() macro.

An example for launch files and nodes:

```
# Install launch files
install(
    DIRECTORY launch
    DESTINATION share/${PROJECT_NAME}
)

# Install nodes
install(
    TARGETS [node-names]
    DESTINATION lib/${PROJECT_NAME}
)
```

Python Packages

ROS2 follows Python’s standard module distribution process that uses setuptools. For Python packages, the setup.py file complements a C++ package’s CMakeLists.txt. More details on distribution can be found in the official documentation.

In your ROS 2 package, you should have a setup.cfg file which looks like:

```
[develop]
script_dir=$base/lib/<package-name>

[install]
install_scripts=$base/lib/<package-name>
```

and a setup.py file that looks like:

````
import os
from glob import glob
from setuptools import find_packages, setup

package_name = 'my_package'

setup(
    name=package_name,
    version='0.0.0',
    # Packages to export
    packages=find_packages(exclude=['test']),
    # Files we want to install, specifically launch files
    data_files=[
        # Install marker file in the package index
    ]
)
```

(continues on next page)
Combined C++ and Python Packages

When writing a package with both C++ and Python code, the setup.py file and setup.cfg file are not used. Instead, use ament_cmake_python.

ament_cmake user documentation

ament_cmake is the build system for CMake based packages in ROS 2 (in particular, it will be used for most if not all C/C++ projects). It is a set of scripts enhancing CMake and adding convenience functionality for package authors. Knowing the basics of CMake will be very helpful, an official tutorial can be found here.
– Adding Dependencies
– Building a Library
– Compiler and linker options
– Building libraries on Windows

• Testing and Linting
  – Linting
  – Testing

• Extending ament
  – Adding a function/macro to ament
  – Adding to extension points
  – Adding extension points

• Adding resources
  – The ament index explained
  – Querying the ament index
  – Adding to the ament index

Basics

A basic CMake outline can be produced using `ros2 pkg create <package_name>` on the command line. The basic build information is then gathered in two files: the `package.xml` and the `CMakeLists.txt`. The `package.xml` must contain all dependencies and a bit of metadata to allow colcon to find the correct build order for your packages, to install the required dependencies in CI as well as provide the information for a release with `bloom`. The `CMakeLists.txt` contains the commands to build and package executables and libraries and will be the main focus of this document.

Basic project outline

The basic outline of the `CMakeLists.txt` of an ament package contains:

```cmake
cmake_minimum_required(VERSION 3.5)
project(my_project)
ament_package()
```

The argument to `project` will be the package name and must be identical to the package name in the `package.xml`. The project setup is done by `ament_package()` and this call must occur exactly once per package. `ament_package()` installs the `package.xml`, registers the package with the ament index, and installs config (and possibly target) files for CMake so that it can be found by other packages using `find_package`. Since `ament_package()` gathers a lot of information from the `CMakeLists.txt` it should be the last call in your `CMakeLists.txt`. Although it is possible to follow calls to `ament_package()` by calls to `install` functions copying files and directories, it is simpler to just keep `ament_package()` the last call.

`ament_package` can be given additional arguments:
• **CONFIG_EXTRAS**: a list of CMake files (.cmake or .cmake.in templates expanded by `configure_file()`) which should be available to clients of the package. For an example of when to use these arguments, see the discussion in Adding resources. For more information on how to use template files, see the official documentation.

• **CONFIG_EXTRAS_POST**: same as CONFIG_EXTRAS, but the order in which the files are added differs. While CONFIG_EXTRAS files are included before the files generated for the `ament_export_*` calls the files from CONFIG_EXTRAS_POST are included afterwards.

Instead of adding to `ament_package`, you can also add to the variable `${PROJECT_NAME}_CONFIG_EXTRAS` and `${PROJECT_NAME}_CONFIG_EXTRAS_POST` with the same effect. The only difference is again the order in which the files are added with the following total order:

• files added by CONFIG_EXTRAS
• files added by appending to `${PROJECT_NAME}_CONFIG_EXTRAS`
• files added by appending to `${PROJECT_NAME}_CONFIG_EXTRAS_POST`
• files added by CONFIG_EXTRAS_POST

### Adding files and headers

There are two main targets to build: libraries and executables which are built by `add_library` and `add_executable` respectively.

With the separation of header files and implementation in C/C++, it is not always necessary to add both files as argument to `add_library`/ `add_executable`.

The following best practice is proposed:

• if you are building a library, put all headers which should be usable by clients and therefore must be installed into a subdirectory of the `include` folder named like the package, while all other files (.c/.cpp and header files which should not be exported) are inside the `src` folder.

• only `.cpp` files are explicitly referenced in the call to `add_library` or `add_executable`

• allow to find headers via

  ```
  target_include_directories(my_target
    PUBLIC
      <$<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
      <$<INSTALL_INTERFACE:include>>
  )
  ```

  This adds all files in the folder `${CMAKE_CURRENT_SOURCE_DIR}/include` to the public interface during build time and all files in the include folder (relative to `${CMAKE_INSTALL_DIR}`) when being installed.

  In principle, using generator expressions here is not necessary if both folders are called `include` and top-level with respect to `${CMAKE_CURRENT_SOURCE_DIR}` and `${CMAKE_INSTALL_DIR}`, but it is very common.
Adding Dependencies

There are two ways to link your packages against a new dependency.

The first and recommended way is to use the ament macro `ament_target_dependencies`. As an example, suppose we want to link `my_target` against the linear algebra library Eigen3.

```cmake
find_package(Eigen3 REQUIRED)
ament_target_dependencies(my_target Eigen3)
```

It includes the necessary headers and libraries and their dependencies to be correctly found by the project. It will also ensure that the include directories of all dependencies are ordered correctly when using overlay workspaces.

The second way is to use `target_link_libraries`.

The recommended way in modern CMake is to only use targets, exporting and linking against them. CMake targets are namedpaced, similar to C++. For instance, Eigen3 defines the target `Eigen3::Eigen`.

At least until Crystal Clemmys target names are not supported in the `ament_target_dependencies` macro. Sometimes it will be necessary to call the `target_link_libraries` CMake function. In the example of Eigen3, the call should then look like

```cmake
find_package(Eigen3 REQUIRED)
target_link_libraries(my_target Eigen3::Eigen)
```

This will also include necessary headers, libraries and their dependencies, but in contrast to `ament_target_dependencies` it might not correctly order the dependencies when using overlay workspaces.

**Note:** It should never be necessary to `find_package` a library that is not explicitly needed but is a dependency of another dependency that is explicitly needed. If that is the case, file a bug against the corresponding package.

Building a Library

When building a reusable library, some information needs to be exported for downstream packages to easily use it.

```cmake
ament_export_targets(my_libraryTargets HAS_LIBRARY_TARGET)
ament_export_dependencies(some_dependency)

install(
  DIRECTORY include/
  DESTINATION include
)

install(
  TARGETS my_library
  EXPORT my_libraryTargets
  LIBRARY DESTINATION lib
  ARCHIVE DESTINATION lib
  RUNTIME DESTINATION bin
  INCLUDES DESTINATION include
)
```

Here, we assume that the folder `include` contains the headers which need to be exported. Note that it is not necessary to put all headers into a separate folder, only those that should be included by clients.
Here is what’s happening in the snippet above:

- The `ament_export_targets` macro exports the targets for CMake. This is necessary to allow your library’s clients to use the `target_link_libraries(client my_library::my_library)` syntax. `ament_export_targets` can take an arbitrary list of targets named as EXPORT in an install call and an additional option `HAS_LIBRARY_TARGET`, which adds potential libraries to environment variables.

- The `ament_export_dependencies` exports dependencies to downstream packages. This is necessary so that the user of the library does not have to call `find_package` for those dependencies, too.

- The first install commands installs the header files which should be available to clients.

```
Warning: Calling `ament_export_targets`, `ament_export_dependencies`, or other ament commands from a CMake subdirectory will not work as expected. This is because the CMake subdirectory has no way of setting necessary variables in the parent scope where `ament_package` is called.
```

- The last large install command installs the library. Archive and library files will be exported to the lib folder, runtime binaries will be installed to the bin folder and the path to installed headers is `include`.

```
Note: Windows DLLs are treated as runtime artifacts and installed into the `RUNTIME DESTINATION` folder. It is therefore advised to not leave out the `RUNTIME` install even when developing libraries on Unix based systems.
```

- Regarding the `include` directory, the install command only adds information to CMake, it does not actually install the includes folder. This is done by copying the headers via `install(DIRECTORY <dir> DESTINATION <dest>)` as described above.

- The `EXPORT` notation of the install call requires additional attention: It installs the CMake files for the `my_library` target. It is named exactly like the argument in `ament_export_targets` and could be named like the library. However, this will then prohibit using the `ament_target_dependencies` way of including your library. To allow for full flexibility, it is advised to prepend the export target with something like `<target>Targets`.

- All install paths are relative to `CMAKE_INSTALL_PREFIX`, which is already set correctly by colcon/ament

There are two additional functions which can be used but are superfluous for target based installs:

```
ament_export_include_directories(include)
ament_export_libraries(my_library)
```

The first macro marks the directory of the exported include directories (this is achieved by `INCLUDES DESTINATION` in the target install call). The second macro marks the location of the installed library (this is done by the `HAS_LIBRARY_TARGET` argument in the call to `ament_export_targets`).

Some of the macros can take different types of arguments for non-target exports, but since the recommended way for modern Make is to use targets, we will not cover them here. Documentation of these options can be found in the source code itself.
Compiler and linker options

ROS 2 targets compilers which comply with the C++14 and C99 standard until at least Crystal Clemmys. Newer versions might be targeted in the future and are referenced here. Therefore it is customary to set the corresponding CMake flags:

```cpp
if(NOT CMAKE_C_STANDARD)
  set(CMAKE_C_STANDARD 99)
endif()
if(NOT CMAKE_CXX_STANDARD)
  set(CMAKE_CXX_STANDARD 14)
endif()
```

To keep the code clean, compilers should throw warnings for questionable code and these warnings should be fixed. It is recommended to at least cover the following warning levels:

- For Visual Studio, the default W1 warnings are kept
- For GCC and Clang: `-Wall -Wextra -Wpedantic` are required and `-Wshadow -Werror` are advisable (the latter makes warnings errors).

Although modern CMake advises to add compiler flags on a target basis, i.e. call

```cpp
target_compile_options(my_target PRIVATE -Wall)
```

it is at the moment recommended to use the directory level function `add_compile_options(-Wall)` to not clutter the code with target-based compile options for all executables and tests.

Building libraries on Windows

Since Linux, Mac and Windows are all officially supported platforms, to have maximum impact any package should also build on Windows. The Windows library format enforces symbol visibility: Every symbol which should be used from a client has to be explicitly exported by the library (and data symbols need to be implicitly imported).

To keep this compatible with Clang and GCC builds, it is advised to use the logic in the GCC wiki. To use it for a package called `my_library`:

- Copy the logic in the link into a header file called `visibility_control.hpp`.
- Replace DLL by `MY_LIBRARY` (for an example, see visibility control of `rviz_rendering`).
- Use the macros “`MY_LIBRARY_PUBLIC`” for all symbols you need to export (i.e. classes or functions).
- In the project `CMakeLists.txt` use:

```cpp
target_compile_definitions(my_library PRIVATE "MY_LIBRARY_BUILDING_LIBRARY")
```

For more details, see `Windows Symbol Visibility in the Windows Tips and Tricks document.`
Testing and Linting

In order to separate testing from building the library with colcon, wrap all calls to linters and tests in a conditional:

```cpp
if(BUILD_TESTING)
    find_package(ament_cmake_gtest REQUIRED)
    ament_add_gtest(<tests>)
endif()
```

Linting

It's advised to use the combined call from `ament_lint_auto`:

```cpp
find_package(ament_lint_auto REQUIRED)
ament_lint_auto_find_test_dependencies()
```

This will run linters as defined in the `package.xml`. It is recommended to use the set of linters defined by the package `ament_lint_common`. The individual linters included there, as well as their functions, can be seen in the `ament_lint_common` docs.

Linters provided by `ament` can also be added separately, instead of running `ament_lint_auto`. One example of how to do so can be found in the `ament_cmake_lint_cmake` documentation.

Testing

Ament contains CMake macros to simplify setting up GTests. Call:

```cpp
find_package(ament_cmake_gtest)
ament_add_gtest(some_test <test_sources>)
```

to add a GTest. This is then a regular target which can be linked against other libraries (such as the project library). The macros have additional parameters:

- **APPEND_ENV**: append environment variables. For instance you can add to the ament prefix path by calling:

  ```cpp
  find_package(ament_cmake_gtest REQUIRED)
  ament_add_gtest(some_test <test_sources> APPEND_ENV PATH=some/additional/path/for/testing/resources)
  ```

- **APPEND_LIBRARY_DIRS**: append libraries so that they can be found by the linker at runtime. This can be achieved by setting environment variables like `PATH` on Windows and `LD_LIBRARY_PATH` on Linux, but this makes the call platform specific.

- **ENV**: set environment variables (same syntax as `APPEND_ENV`).

- **TIMEOUT**: set a test timeout in second. The default for GTests is 60 seconds. For example:

  ```cpp
  ament_add_gtest(some_test <test_sources> TIMEOUT 120)
  ```

- **SKIP_TEST**: skip this test (will be shown as “passed” in the console output).

- **SKIP_LINKING_MAIN_LIBRARIES**: Don’t link against GTest.

- **WORKING_DIRECTORY**: set the working directory for the test.
The default working directory otherwise is the `CMAKE_CURRENT_BINARY_DIR`, which is described in the CMake documentation.

Similarly, there is a CMake macro to set up GTest including GMock:

```cmake
find_package(ament_cmake_gmock REQUIRED)
ament_add_gmock(some_test <test_sources>)
```

It has the same additional parameters as `ament_add_gtest`.

## Extending ament

It is possible to register additional macros/functions with `ament_cmake` and extend it in several ways.

### Adding a function/macro to `ament`

Extending `ament` will often times mean that you want to have some functions available to other packages. The best way to provide the macro to client packages is to register it with `ament`.

This can be done by appending the `$\{PROJECT_NAME\}_CONFIG_EXTRAS` variable, which is used by `ament_package()` via

```cmake
list(APPEND $\{PROJECT_NAME\}_CONFIG_EXTRAS
  path/to/file.cmake"
  other/path/to/file.cmake"
)
```

Alternatively, you can directly add the files to the `ament_package()` call:

```cmake
ament_package(CONFIG_EXTRAS
  path/to/file.cmake
  other/path/to/file.cmake
)
```

### Adding to extension points

In addition to simple files with functions that can be used in other packages, you can also add extensions to `ament`. Those extensions are scripts which are executed with the function which defines the extension point. The most common use-case for `ament` extensions is probably registering rosidl message generators: When writing a generator, you normally want to generate all messages and services with your generator also without modifying the code for the message/service definition packages. This is possible by registering the generator as an extension to `rosidl_generate_interfaces`.

As an example, see

```cmake
ament_register_extension(
  "rosidl_generate_interfaces"
  "rosidl_generator_cpp"
  "rosidl_generator_cpp_generate_interfaces.cmake")
```

which registers the macro `rosidl_generator_cpp_generate_interfaces.cmake` for the package `rosidl_generator_cpp` to the extension point `rosidl_generate_interfaces`. When the extension point gets executed, this will trigger the execution of the script `rosidl_generator_cpp_generate_interfaces.cmake` here. In particular, this will call the generator whenever the function `rosidl_generate_interfaces` gets executed.
The most important extension point for generators, aside from `rosidl_generate_interfaces`, is `ament_package`, which will simply execute scripts with the `ament_package()` call. This extension point is useful when registering resources (see below).

`ament_register_extension` is a function which takes exactly three arguments:

- `extension_point`: The name of the extension point (most of the time this will be one of `ament_package` or `rosidl_generate_interfaces`)
- `package_name`: The name of the package containing the CMake file (i.e. the project name of the project where the file is written to)
- `cmake_filename`: The CMake file executed when the extension point is run

**Note:** It is possible to define custom extension points in a similar manner to `ament_package` and `rosidl_generate_interfaces`, but this should hardly be necessary.

### Adding extension points

Very rarely, it might be interesting to define a new extension point to `ament`.

Extension points can be registered within a macro so that all extensions will be executed when the corresponding macro is called. To do so:

- Define and document a name for your extension (e.g. `my_extension_point`), which is the name passed to the `ament_register_extension` macro when using the extension point.
- In the macro/function which should execute the extensions call:

```
ament_execute_extensions(my_extension_point)
```

Ament extensions work by defining a variable containing the name of the extension point and filling it with the macros to be executed. Upon calling `ament_execute_extensions`, the scripts defined in the variable are then executed one after another.

### Adding resources

Especially when developing plugins or packages which allow plugins it is often essential to add resources to one ROS package from another (e.g. a plugin). Examples can be plugins for tools using the pluginlib.

This can be achieved using the ament index (also called “resource index”).

### The ament index explained

For details on the design and intentions, see [here](#).

In principle, the ament index is contained in a folder within the install/share folder of your package. It contains shallow subfolders named after different types of resources. Within the subfolder, each package providing said resource is referenced by name with a “marker file”. The file may contain whatever content necessary to obtain the resources, e.g. relative paths to the installation directories of the resource, it may also be simply empty.

To give an example, consider providing display plugins for RViz: When providing RViz plugins in a project named `my_rviz_displays` which will be read by the pluginlib, you will provide a `plugin_description.xml` file, which will be installed and used by the pluginlib to load the plugins. To achieve this, the `plugin_description.xml` is registered as a resource in the `resource_index` via
When running `colcon build`, this installs a file `my_rviz_displays` into a subfolder `rviz_common__pluginlib__plugin` into the resource index. Pluginlib factories within `rviz_common` will know to gather information from all folders named `rviz_common__pluginlib__plugin` for packages that export plugins. The marker file for pluginlib factories contains an install-folder relative path to the `plugins_description.xml` file (and the name of the library as marker file name). With this information, the pluginlib can load the library and know which plugins to load from the `plugin_description.xml` file.

As a second example, consider the possibility to let your own RViz plugins use your own custom meshes. Meshes get loaded at startup time so that the plugin owner does not have to deal with it, but this implies RViz has to know about the meshes. To achieve this, RViz provides a function:

```
register_rviz_ogre_media_exports(DIRECTORIES <my_dirs>)
```

This registers the directories as an ogre_media resource in the ament index. In short, it installs a file named after the project which calls the function into a subfolder called `rviz_ogre_media_exports`. The file contains the install folder relative paths to the directories listed in the macros. On startup time, RViz can now search for all folders called `rviz_ogre_media_exports` and load resources in all folders provided. These searches are done using `ament_index_cpp` (or `ament_index_py` for Python packages).

In the following sections we will explore how to add your own resources to the ament index and provide best practices for doing so.

### Querying the ament index

If necessary, it is possible to query the ament index for resources via CMake. To do so, there are three functions:

**ament_index_has_resource**: obtain a prefix path to the resource if it exists with the following parameters:

- **var**: output parameter: fill this variable with FALSE if the resource does not exist or the prefix path to the resource otherwise
- **resource_type**: The type of the resource (e.g. `rviz_common__pluginlib__plugin`)
- **resource_name**: The name of the resource which usually amounts to the name of the package having added the resource of type `resource_type` (e.g. `rviz_default_plugins`)

**ament_index_get_resource**: Obtain the content of a specific resource, i.e. the contents of the marker file in the ament index.

- **var**: output parameter: filled with the content of the resource marker file if it exists.
- **resource_type**: The type of the resource (e.g. `rviz_common__pluginlib__plugin`)
- **resource_name**: The name of the resource which usually amounts to the name of the package having added the resource of type `resource_type` (e.g. `rviz_default_plugins`)
- **PREFIX_PATH**: The prefix path to search for (usually, the default `ament_index_get_prefix_path()` will be enough).

Note that `ament_index_get_resource` will throw an error if the resource does not exist, so it might be necessary to check using `ament_index_has_resource`.

**ament_index_get_resources**: Get all packages which registered resources of a specific type from the index

- **var**: Output parameter: filled with a list of names of all packages which registered a resource of `resource_type`
- **resource_type**: The type of the resource (e.g. `rviz_common__pluginlib__plugin`)
• **PREFIX_PATH**: The prefix path to search for (usually, the default `ament_index_get_prefix_path()` will be enough).

### Adding to the ament index

Defining a resource requires two bits of information:

• a name for the resource which must be unique,

• a layout of the marker file, which can be anything and could also be empty (this is true for instance for the “package” resource marking a ROS 2 package)

For the RViz mesh resource, the corresponding choices were:

• `rviz_ogre_media_exports` as name of the resource,

• install path relative paths to all folders containing resources. This will already enable you to write the logic for using the corresponding resource in your package.

To allow users to easily register resources for your package, you should furthermore provide macros or functions such as the pluginlib function or `rviz_ogre_media_exports` function.

To register a resource, use the ament function `ament_index_register_resource`. This will create and install the marker files in the resource_index. As an example, the corresponding call for `rviz_ogre_media_exports` is the following:

```cmake
ament_index_register_resource(rviz_ogre_media_exports CONTENT ${OGRE_MEDIA_RESOURCE_FILE} ...
)```

This installs a file named like `${PROJECT_NAME}` into a folder `rviz_ogre_media_exports` into the resource_index with content given by variable `${OGRE_MEDIA_RESOURCE_FILE}`. The macro has a number of parameters that can be useful:

• the first (unnamed) parameter is the name of the resource, which amounts to the name of the folder in the resource_index

• **CONTENT**: The content of the marker file as string. This could be a list of relative paths, etc. **CONTENT** cannot be used together with **CONTENT_FILE**.

• **CONTENT_FILE**: The path to a file which will be used to create the marker file. The file can be a plain file or a template file expanded with `configure_file()`. **CONTENT_FILE** cannot be used together with **CONTENT**.

• **PACKAGE_NAME**: The name of the package/library exporting the resource, which amounts to the name of the marker file. Defaults to `${PROJECT_NAME}`.

• **AMENT_INDEX_BINARY_DIR**: The base path of the generated ament index. Unless really necessary, always use the default `${CMAKE_BINARY_DIR}/ament_cmake_index`.

• **SKIP_INSTALL**: Skip installing the marker file.

Since only one marker file exists per package, it is usually a problem if the CMake function/macro gets called twice by the same project. However, for large projects it might be best to split up calls registering resources.

Therefore, it is best practice to let a macro registering a resource such as `register_rviz_ogre_media_exports.cmake` only fill some variables. The real call to `ament_index_register_resource` can then be added within an ament extension to `ament_package`. Since there must only ever be one call to `ament_package` per project, there will always only be one place where the resource gets registered. In the case of `rviz_ogre_media_exports` this amounts to the following strategy:

• The macro `register_rviz_ogre_media_exports` takes a list of folders and appends them to a variable called `OGRE_MEDIA_RESOURCE_FILE`. 

---

1154 Chapter 4. Contributing to the documentation
Another macro called `register_rviz_ogre_media_exports_hook` calls `ament_index_register_resource` if `${OGRE_MEDIARESOURCE_FILE}` is non-empty.

The `register_rviz_ogre_media_exports_hook.cmake` file is registered as an ament extension in a third file `register_rviz_ogre_media_exports_hook-extras.cmake` via calling

```
ament_register_extension("ament_package" "rviz_rendering"
"register_rviz_ogre_media_exports_hook.cmake"
)
```

The files `register_rviz_ogre_media_exports.cmake` and `register_rviz_ogre_media_exports_hook-extra.cmake` are registered as `CONFIG_EXTRA` with `ament_package()`.

### ament_cmake_python user documentation

`ament_cmake_python` is a package that provides CMake functions for packages of the `ament_cmake` build type that contain Python code. See the `ament_cmake user documentation` for more information.

**Note:** Pure Python packages should use the `ament_python` build type in most cases. To create an `ament_python` package, see *Creating your first ROS 2 package*. `ament_cmake_python` should only be used in cases where that is not possible, like when mixing C/C++ and Python code.

### Basics

#### Basic project outline

The outline of a package called “my_project” with the `ament_cmake` build type that uses `ament_cmake_python` looks like:

```
my_project
  CMakeLists.txt
  package.xml
  my_project
    __init__.py
    my_script.py
```

The `__init__.py` file can be empty, but it is needed to make Python treat the directory containing it as a package. There can also be a `src` or `include` directory alongside the `CMakeLists.txt` which holds C/C++ code.
Using ament_cmake_python

The package must declare a dependency on ament_cmake_python in its package.xml.

```xml
<buildtool_depend>ament_cmake_python</buildtool_depend>
```

The CMakeLists.txt should contain:

```cmake
find_package(ament_cmake_python REQUIRED)
# ...
ament_python_install_package(${PROJECT_NAME})
```

The argument to `ament_python_install_package()` is the name of the directory alongside the CMakeLists.txt that contains the Python file. In this case, it is `my_project`, or `${PROJECT_NAME}`.

**Warning:** Calling `rosidl_generate_interfaces` and `ament_python_install_package` in the same CMake project does not work. See this Github issue for more info. It is best practice to instead separate out the message generation into a separate package.

Then, another Python package that correctly depends on `my_project` can use it as a normal Python module:

```python
from my_project.my_script import my_function
```

Assuming `my_script.py` contains a function called `my_function()`.

Using ament_cmake_pytest

The package ament_cmake_pytest is used to make tests discoverable to cmake. The package must declare a test dependency on ament_cmake_pytest in its package.xml.

```xml
<test_depend>ament_cmake_pytest</test_depend>
```

Say the package has a file structure like below, with tests in the tests folder.

```
.  
  CMakeLists.txt
  my_project
    - my_script.py
  package.xml
  tests
    - test_a.py
    - test_b.py
```

The CMakeLists.txt should contain:

```cmake
if(BUILD_TESTING)
  find_package(ament_cmake_pytest REQUIRED)
  set(_pytest_tests
tests/test_a.py
tests/test_b.py
  # Add other test files here
)
```

(continues on next page)
Vulcanexus Documentation, Release 1.0.0

(continued from previous page)

```bash
foreach(_test_path ${_pytest_tests})
    get_filename_component(_test_name ${_test_path} NAME_WE)
    ament_add_pytest_test(${_test_name} ${_test_path}
        APPEND_ENV PYTHONPATH=${CMAKE_CURRENT_BINARY_DIR}
        TIMEOUT 60
        WORKING_DIRECTORY ${CMAKE_SOURCE_DIR}
    )
endforeach()
endif()
```

Compared to the usage of ament_python, which supports automatic test discovery, ament_cmake_pytest must be called with the path to each test file. The timeout can be reduced as needed.

Now, you can invoke your tests with the standard colcon testing commands.

**Migrating launch files from ROS 1 to ROS 2**

### Table of Contents

- **Background**
- **Migrating tags from ROS 1 to ROS 2**
- **New tags in ROS 2**
- **Replacing an include tag**
- **Substitutions**
- **Type inference rules**

This guide describes how to write XML launch files for an easy migration from ROS 1.

**Background**

A description of the ROS 2 launch system and its Python API can be found in *Launch System tutorial*.

**Migrating tags from ROS 1 to ROS 2**

**launch**

- Available in ROS 1.
- `launch` is the root element of any ROS 2 launch XML file.
node

- Available in ROS 1.
- Launches a new node.
- Differences from ROS 1:
  - `type` attribute is now `exec`.
  - `ns` attribute is now `namespace`.
  - The following attributes aren’t available: `machine`, `respawn_delay`, `clear_params`.

Example

```xml
<launch>
  <node pkg="demo_nodes_cpp" exec="talker"/>
  <node pkg="demo_nodes_cpp" exec="listener"/>
</launch>
```

dparam

- Available in ROS 1.
- Used for passing a parameter to a node.
- There’s no global parameter concept in ROS 2. For that reason, it can only be used nested in a `node` tag. Some attributes aren’t supported in ROS 2: `type`, `textfile`, `binfile`, `executable`, `command`.

Example

```xml
<launch>
  <node pkg="demo_nodes_cpp" exec="parameter_event">
    <param name="foo" value="5"/>
  </node>
</launch>
```

Type inference rules

Here are some examples of how to write parameters:

```xml
<node pkg="my_package" exec="my_executable" name="my_node">
  <!--A string parameter with value "1"-->
  <param name="a_string" value="1"/>
  <!--A integer parameter with value 1-->
  <param name="an_int" value="1"/>
  <!--A float parameter with value 1.0-->
  <param name="a_float" value="1.0"/>
  <!--A string parameter with value "asd"-->
  <param name="another_string" value="asd"/>
</node>
```
Parameter grouping

In ROS 2, \texttt{param} tags are allowed to be nested. For example:

```xml
<node pkg="my_package" exec="my_executable" name="my_node" ns="/an_absoulute_ns">
  <param name="group1">
    <param name="group2">
      <param name="my_param" value="1"/>
    </param>
  </param>
  <param name="another_param" value="2"/>
</node>
```

That will create two parameters:

- A \texttt{group1.group2.my_param} of value 1, hosted by node \texttt{/an_absolute_ns/my_node}.
- A \texttt{group1.another_param} of value 2 hosted by node \texttt{/an_absolute_ns/my_node}.

It’s also possible to use full parameter names:

```xml
<node pkg="my_package" exec="my_executable" name="my_node" ns="/an_absoulute_ns">
  <param name="group1.group2.my_param" value="1"/>
  <param name="group1.another_param" value="2"/>
</node>
```

\texttt{rosparm}

- Available in ROS 1.
- Loads parameters from a yaml file.
- It has been replaced with a \texttt{from} attribute in \texttt{param} tags.
Example

```xml
<node pkg="my_package" exec="my_executable" name="my_node" ns="/an_absolute_ns">
  <param from="/path/to/file"/>
</node>
```

**remap**

- Available in ROS 1.
- Used to pass remapping rules to a node.
- It can only be used within node tags.

Example

```xml
<launch>
  <node pkg="demo_nodes_cpp" exec="talker">
    <remap from="chatter" to="my_topic"/>
  </node>
  <node pkg="demo_nodes_cpp" exec="listener">
    <remap from="chatter" to="my_topic"/>
  </node>
</launch>
```

**include**

- Available in ROS 1.
- Allows including another launch file.
- Differences from ROS 1:
  - Available in ROS 1, included content was scoped. In ROS 2, it's not. Nest includes in group tags to scope them.
  - ns attribute is not supported. See example of push_ros_namespace tag for a workaround.
  - arg tags nested in an include tag don't support conditionals (if or unless).
  - There is no support for nested env tags. set_env and unset_env can be used instead.
  - Both clear_params and pass_all_args attributes aren't supported.
Examples

See *Replacing an include tag*.

**arg**

- Available in ROS 1.
- **arg** is used for declaring a launch argument, or to pass an argument when using **include** tags.
- Differences from ROS 1:
  - value attribute is not allowed. Use **let** tag for this.
  - doc is now **description**.
  - When nested within an **include** tag, **if** and **unless** attributes aren’t allowed.

**Example**

```xml
<launch>
  <arg name="topic_name" default="chatter"/>
  <node pkg="demo_nodes_cpp" exec="talker">
    <remap from="chatter" to="$(var topic_name)"/>
  </node>
  <node pkg="demo_nodes_cpp" exec="listener">
    <remap from="chatter" to="$(var topic_name)"/>
  </node>
</launch>
```

**Passing an argument to the launch file**

In the XML launch file above, the **topic_name** defaults to the name **chatter**, but can be configured on the command-line. Assuming the above launch configuration is in a file named `mylaunch.xml`, a different topic name can be used by launching it with the following:

```
ros2 launch mylaunch.xml topic_name:=custom_topic_name
```

There is some additional information about passing command-line arguments in *Using Substitutions*.

**env**

- Available in ROS 1.
- Sets an environment variable.
- It has been replaced with **env**, **set_env** and **unset_env**:
  - **env** can only be used nested in a **node** or **executable** tag. **if** and **unless** tags aren’t supported.
  - **set_env** can be nested within the root tag **launch** or in **group** tags. It accepts the same attributes as **env**, and also **if** and **unless** tags.
  - **unset_env** unsets an environment variable. It accepts a **name** attribute and conditionals.
Example

```xml
<launch>
  <set_env name="MY_ENV_VAR" value="MY_VALUE" if="CONDITION_A"/>
  <set_env name="ANOTHER_ENV_VAR" value="ANOTHER_VALUE" unless="CONDITION_B"/>
  <set_env name="SOME_ENV_VAR" value="SOME_VALUE"/>
  <node pkg="MY_PACKAGE" exec="MY_EXECUTABLE" name="MY_NODE">
    <env name="NODE_ENV_VAR" value="SOME_VALUE"/>
  </node>
  <unset_env name="MY_ENV_VAR" if="CONDITION_A"/>
  <node pkg="ANOTHER_PACKAGE" exec="ANOTHER_EXECUTABLE" name="ANOTHER_NODE"/>
  <unset_env name="ANOTHER_ENV_VAR" unless="CONDITION_B"/>
  <unset_env name="SOME_ENV_VAR"/>
</launch>
```

group

- Available in ROS 1.
- Allows limiting the scope of launch configurations. Usually used together with let, include and push_ros_namespace tags.
- Differences from ROS 1:
  - There is no ns attribute. See the new push_ros_namespace tag as a workaround.
  - clear_params attribute isn’t available.
  - It doesn’t accept remap nor param tags as children.

Example

launch-prefix configuration affects both executable and node tags’ actions. This example will use time as a prefix if use_time_prefix_in_talker argument is 1, only for the talker.

```xml
<launch>
  <arg name="use_time_prefix_in_talker" default="0"/>
  <group>
    <let name="launch-prefix" value="time" if="$(var use_time_prefix_in_talker)"/>
    <node pkg="demo_nodes_cpp" exec="talker"/>
  </group>
  <node pkg="demo_nodes_cpp" exec="listener"/>
</launch>
```
machine

It is not supported at the moment.

test

It is not supported at the moment.

New tags in ROS 2

set_env and unset_env

See env tag description.

push_ros_namespace

include and group tags don’t accept an ns attribute. This action can be used as a workaround:

```xml
<group>
  <push_ros_namespace namespace="my_ns"/>
  <!--Nodes here are namedpaced with "my_ns".-->
  <!--If there is an include action here, its nodes will also be namespaced.-->
  <push_ros_namespace namespace="another_ns"/>
  <!--Nodes here are namedpaced with "another_ns/my_ns".-->
  <push_ros_namespace namespace="/absolute_ns"/>
  <!--Nodes here are namedpaced with "/absolute_ns".-->
  <!--The following node receives an absolute namespace, so it will ignore the others.
   previously pushed.-->
  <!--The full path of the node will be /asd/my_node.-->
  <node pkg="my_pkg" exec="my_executable" name="my_node" ns="/asd"/>
</group>

<!--Nodes outside the group action won't be namedpaced.-->
```

let

It's a replacement of arg tag with a value attribute.

```xml
<let name="foo" value="asd"/>
```
executable

It allows running any executable.

Example

```xml
<executable cmd="ls -las" cwd="/var/log" name="my_exec" launch-prefix="something" output="screen" shell="true">
  <env name="LD_LIBRARY" value="/lib/some.so"/>
</executable>
```

Replacing an include tag

In order to include a launch file under a namespace as in ROS 1 then the include tags must be nested in a group tag.

```xml
<group>
  <include file="another_launch_file"/>
</group>
```

Then, instead of using the ns attribute, add the push_ros_namespace action tag to specify the namespace:

```xml
<group>
  <push_ros_namespace namespace="my_ns"/>
  <include file="another_launch_file"/>
</group>
```

Nesting include tags under a group tag is only required when specifying a namespace

Substitutions

Documentation about ROS 1’s substitutions can be found in roslaunch XML wiki. Substitutions syntax hasn’t changed, i.e. it still follows the $(substitution-name arg1 arg2 ...) pattern. There are, however, some changes w.r.t. ROS 1:

- `env` and `optenv` tags have been replaced by the `env` tag. `$(env <NAME>)` will fail if the environment variable doesn’t exist. `$(env <NAME> '')` does the same as ROS 1’s `$(optenv <NAME>)`. `$(env <NAME> <DEFAULT>)` does the same as ROS 1’s `$(env <NAME> <DEFAULT>)` or `$(optenv <NAME> <DEFAULT>)`.
- `find` has been replaced with `find-pkg-share` (substituting the share directory of an installed package). Alternatively `find-pkg-prefix` will return the root of an installed package.
- There is a new `exec-in-pkg` substitution. E.g.: `$(exec-in-pkg <package_name> <exec_name>)`.
- There is a new `find-exec` substitution.
- `arg` has been replaced with `var`. It looks at configurations defined either with `arg` or `let` tag.
- `eval` and `dirname` substitutions haven’t changed.
- `anon` substitution is not supported.
Type inference rules

The rules that were shown in Type inference rules subsection of param tag applies to any attribute. For example:

```xml
<!--Setting a string value to an attribute expecting an int will raise an error.-->
<tag1 attr-expecting-an-int="'1'"/>
<!--Correct version.-->
<tag1 attr-expecting-an-int="1"/>
<!--Setting an integer in an attribute expecting a string will raise an error.-->
<tag2 attr-expecting-a-str="1"/>
<!--Correct version.-->
<tag2 attr-expecting-a-str="'1'"/>
<!--Setting a list of strings in an attribute expecting a string will raise an error.-->
<tag3 attr-expecting-a-str="asd, bsd" str-attr-sep=", "/
<!--Correct version.-->
<tag3 attr-expecting-a-str="don't use a separator"/>
```

Some attributes accept more than a single type, for example value attribute of param tag. It’s usual that parameters that are of type int (or float) also accept an str, that will be later substituted and tried to convert to an int (or float) by the action.

Using Python, XML, and YAML for ROS 2 Launch Files

Table of Contents

- Launch file examples
- Using the Launch files from the command line
- Python, XML, or YAML: Which should I use?

ROS 2 launch files can be written in Python, XML, and YAML. This guide shows how to use these different formats to accomplish the same task, as well as has some discussion on when to use each format.

Launch file examples

Below is a launch file implemented in Python, XML, and YAML. Each launch file performs the following actions:

- Setup command line arguments with defaults
- Include another launch file
- Include another launch file in another namespace
- Start a node and setting its namespace
- Start a node, setting its namespace, and setting parameters in that node (using the args)
- Create a node to remap messages from one topic to another

Python

```python
# example_launch.py
import os
```
from ament_index_python import get_package_share_directory
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch.actions import GroupAction
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch.substitutions import LaunchConfiguration
from launch.substitutions import TextSubstitution
from launch_ros.actions import Node
from launch_ros.actions import PushRosNamespace
from launch_xml.launch_description_sources import XMLLaunchDescriptionSource
from launch_yaml.launch_description_sources import YAMLLaunchDescriptionSource

def generate_launch_description():
    # args that can be set from the command line or a default will be used
    background_r_launch_arg = DeclareLaunchArgument("background_r", default_value=TextSubstitution(text="0"))
    background_g_launch_arg = DeclareLaunchArgument("background_g", default_value=TextSubstitution(text="255"))
    background_b_launch_arg = DeclareLaunchArgument("background_b", default_value=TextSubstitution(text="0"))
    chatter_py_ns_launch_arg = DeclareLaunchArgument("chatter_py_ns", default_value=TextSubstitution(text="chatter/py/ns"))
    chatter_xml_ns_launch_arg = DeclareLaunchArgument("chatter_xml_ns", default_value=TextSubstitution(text="chatter/xml/ns"))
    chatter_yaml_ns_launch_arg = DeclareLaunchArgument("chatter_yaml_ns", default_value=TextSubstitution(text="chatter/yaml/ns"))

    # include another launch file
    launch_include = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            os.path.join(get_package_share_directory('demo_nodes_cpp'), 'launch/topics/talker_listener_launch.py'))
    )

    # include a Python launch file in the chatter_py_ns namespace
    launch_py_include_with_namespace = GroupAction(
        actions=[
            # push_ros_namespace to set namespace of included nodes
            PushRosNamespace('chatter_py_ns'),
            IncludeLaunchDescription(
                PythonLaunchDescriptionSource(
                    os.path.join(get_package_share_directory('demo_nodes_cpp'), 'launch/topics/talker_listener_launch.py'))
            )
        ]
    )
os.path.join(
    get_package_share_directory('demo_nodes_cpp'),
    'launch/topics/talker_listener_launch.py'))

# include a xml launch file in the chatter_xml_ns namespace
launch_xml_include_with_namespace = GroupAction(
    actions=[
        # push_ros_namespace to set namespace of included nodes
        PushRosNamespace('chatter_xml_ns'),
        IncludeLaunchDescription(
            XMLLaunchDescriptionSource(
                os.path.join(
                    get_package_share_directory('demo_nodes_cpp'),
                    'launch/topics/talker_listener_launch.xml'))
            ),
        ]
    )

# include a yaml launch file in the chatter_yaml_ns namespace
launch_yaml_include_with_namespace = GroupAction(
    actions=[
        # push_ros_namespace to set namespace of included nodes
        PushRosNamespace('chatter_yaml_ns'),
        IncludeLaunchDescription(
            YAMLLaunchDescriptionSource(
                os.path.join(
                    get_package_share_directory('demo_nodes_cpp'),
                    'launch/topics/talker_listener_launch.yaml'))
            ),
        ]
    )

# start a turtlesim_node in the turtlesim1 namespace
 turtlesim_node = Node(
    package='turtlesim',
    namespace='turtlesim1',
    executable='turtlesim_node',
    name='sim'
  )

# start another turtlesim_node in the turtlesim2 namespace
# and use args to set parameters
 turtlesim_node_with_parameters = Node(
    package='turtlesim',
    namespace='turtlesim2',
    executable='turtlesim_node',
    name='sim',
    parameters=[[
        "background_r": LaunchConfiguration('background_r'),
    ]]  
)
"background_g": LaunchConfiguration('background_g'),
"background_b": LaunchConfiguration('background_b'),
]
)

# perform remap so both turtles listen to the same command topic
forward_turtlesim_commands_to_second_turtlesim_node = Node(
    package='turtlesim',
    executable='mimic',
    name='mimic',
    remappings=[
        ('/input/pose', '/turtlesim1/turtle1/pose'),
        ('/output/cmd_vel', '/turtlesim2/turtle1/cmd_vel'),
    ]
)

return LaunchDescription(
    background_r_launch_arg,
    background_g_launch_arg,
    background_b_launch_arg,
    chatter_py_ns_launch_arg,
    chatter_xml_ns_launch_arg,
    chatter_yaml_ns_launch_arg,
    launch_include,
    launch_py_include_with_namespace,
    launch_xml_include_with_namespace,
    launch_yaml_include_with_namespace,
    turtlesim_node,
    turtlesim_node_with_parameters,
    forward_turtlesim_commands_to_second_turtlesim_node,
]
)

XML

</launch>

<!-- include another launch file -->
<include file="$(find-pkg-share demo_nodes_cpp)/launch/topics/talker_listener_launch.py" />

<!-- include a Python launch file in the chatter_py_ns namespace-->
<group>

<!-- push_ros_namespace to set namespace of included nodes -->
<push_ros_namespace namespace="$(var chatter_py_ns)" />
<include file="$(find-pkg-share demo_nodes_cpp)/launch/topics/talker_listener.launch.py" />
</group>

<!-- include a xml launch file in the chatter_xml_ns namespace-->
<group>
<!-- push_ros_namespace to set namespace of included nodes -->
<push_ros_namespace namespace="$(var chatter_xml_ns)" />
<include file="$(find-pkg-share demo_nodes_cpp)/launch/topics/talker_listener.launch.xml" />
</group>

<!-- include a yaml launch file in the chatter_yaml_ns namespace-->
<group>
<!-- push_ros_namespace to set namespace of included nodes -->
<push_ros_namespace namespace="$(var chatter_yaml_ns)" />
<include file="$(find-pkg-share demo_nodes_cpp)/launch/topics/talker_listener.launch.yaml" />
</group>

<!-- start a turtlesim_node in the turtlesim1 namespace -->
<node pkg="turtlesim" exec="turtlesim_node" name="sim" namespace="turtlesim1" />

<!-- start another turtlesim_node in the turtlesim2 namespace and use args to set parameters -->
<node pkg="turtlesim" exec="turtlesim_node" name="sim" namespace="turtlesim2">
  <param name="background_r" value="$(var background_r)" />
  <param name="background_g" value="$(var background_g)" />
  <param name="background_b" value="$(var background_b)" />
</node>

<!-- perform remap so both turtles listen to the same command topic -->
<node pkg="turtlesim" exec="mimic" name="mimic">
  <remap from="/input/pose" to="/turtlesim1/turtle1/pose" />
  <remap from="/output/cmd_vel" to="/turtlesim2/turtle1/cmd_vel" />
</node>
</launch>

---

YAML

# example_launch.yaml

launch:

# args that can be set from the command line or a default will be used
- arg:
  name: "background_r"
  default: "0"
- arg:
  name: "background_g"
  default: "255"
- arg:
  name: "background_b"
  default: "0"
- arg:
name: "chatter_py_ns"
default: "chatter/py/ns"
- arg:
  name: "chatter_xml_ns"
default: "chatter/xml/ns"
- arg:
  name: "chatter_yaml_ns"
default: "chatter/yaml/ns"

# include another launch file
- include:
  file: "$(find-pkg-share demo_nodes_cpp)/launch/topics/talker_listener_launch.py"

# include a Python launch file in the chatter_py_ns namespace
- group:
  - push_ros_namespace:
    namespace: "$var chatter_py_ns"
  - include:
    file: "$(find-pkg-share demo_nodes_cpp)/launch/topics/talker_listener_launch.py"

# include an xml launch file in the chatter_xml_ns namespace
- group:
  - push_ros_namespace:
    namespace: "$var chatter_xml_ns"
  - include:
    file: "$(find-pkg-share demo_nodes_cpp)/launch/topics/talker_listener_launch.xml"

# include a yaml launch file in the chatter_yaml_ns namespace
- group:
  - push_ros_namespace:
    namespace: "$var chatter_yaml_ns"
  - include:
    file: "$(find-pkg-share demo_nodes_cpp)/launch/topics/talker_listener_launch.yaml"

# start a turtlesim_node in the turtlesim1 namespace
- node:
  pkg: "turtlesim"
  exec: "turtlesim_node"
  name: "sim"
  namespace: "turtlesim1"

# start another turtlesim_node in the turtlesim2 namespace and use args to set parameters
- node:
  pkg: "turtlesim"
  exec: "turtlesim_node"
  name: "sim"
  namespace: "turtlesim2"
  param: 
    - name: "background_r"
Using the Launch files from the command line

Launching

Any of the launch files above can be run with `ros2 launch`. To try them locally, you can either create a new package and use

```
ros2 launch <package_name> <launch_file_name>
```

or run the file directly by specifying the path to the launch file

```
ros2 launch <path_to_launch_file>
```

Setting arguments

To set the arguments that are passed to the launch file, you should use `key:=value` syntax. For example, you can set the value of `background_r` in the following way:

```
ros2 launch <package_name> <launch_file_name> background_r:=255
```

or

```
ros2 launch <path_to_launch_file> background_r:=255
```
Controlling the turtles

To test that the remapping is working, you can control the turtles by running the following command in another terminal:

```
ros2 run turtlesim turtle_teleop_key --ros-args --remap __ns:=/turtlesim1
```

Python, XML, or YAML: Which should I use?

**Note:** Launch files in ROS 1 were written in XML, so XML may be the most familiar to people coming from ROS 1. To see what’s changed, you can visit [Migrating launch files from ROS 1 to ROS 2](#).

For most applications the choice of which ROS 2 launch format comes down to developer preference. However, if your launch file requires flexibility that you cannot achieve with XML or YAML, you can use Python to write your launch file. Using Python for ROS 2 launch is more flexible because of following two reasons:

- Python is a scripting language, and thus you can leverage the language and its libraries in your launch files.
- `ros2/launch` (general launch features) and `ros2/launch_ros` (ROS 2 specific launch features) are written in Python and thus you have lower level access to launch features that may not be exposed by XML and YAML.

That being said, a launch file written in Python may be more complex and verbose than one in XML or YAML.

Using ROS 2 launch to launch composable nodes

In the *Composition tutorial*, you learned about composable nodes and how to use them from the command-line. In the *Launch tutorials*, you learned about launch files and how to use them to manage multiple nodes.

This guide will combine the above two topics and teach you how to write launch files for composable nodes.
Setup

See the installation instructions for details on installing ROS 2.

If you’ve installed ROS 2 from packages, ensure that you have ros-iron-image-tools installed. If you downloaded the archive or built ROS 2 from source, it will already be part of the installation.

Launch file examples

Below is a launch file that launches composable nodes in Python, XML, and YAML. The launch files all do the following:

- Instantiate a cam2image composable node with remappings, custom parameters, and extra arguments
- Instantiate a showimage composable node with remappings, custom parameters, and extra arguments

Python

```python
import launch
from launch_ros.actions import ComposableNodeContainer
from launch_ros.descriptions import ComposableNode

def generate_launch_description():
    """Generate launch description with multiple components."""
    container = ComposableNodeContainer(
        name='image_container',
        namespace='",
        package='rclcpp_components',
        executable='component_container',
        composable_node_descriptions=[
            ComposableNode(
                package='image_tools',
                plugin='image_tools::Cam2Image',
                name='cam2image',
                remappings=[('/image', '/burgerimage')],
                parameters=[{'width': 320, 'height': 240, 'burger_mode': True, 'history': 'keep_last'}],
                extra_arguments=[{'use_intra_process_comms': True}]),
            ComposableNode(
                package='image_tools',
                plugin='image_tools::ShowImage',
                name='showimage',
                remappings=[('/image', '/burgerimage')],
                parameters=[{'history': 'keep_last'}],
                extra_arguments=[{'use_intra_process_comms': True}])
        ],
        output='both',
    )

    return launch.LaunchDescription([container])
```

XML
<launch>
  <node_container pkg="rclcpp_components" exec="component_container" name="image_container"
    namespace="">
    <composable_node pkg="image_tools" plugin="image_tools::Cam2Image" name="cam2image"
      namespace="">
      <remap from="/image" to="/burgerimage" />
      <param name="width" value="320" />
      <param name="height" value="240" />
      <param name="burger_mode" value="true" />
      <param name="history" value="keep_last" />
      <extra_arg name="use_intra_process_comms" value="true" />
    </composable_node>
    <composable_node pkg="image_tools" plugin="image_tools::ShowImage" name="showimage"
      namespace="">
      <remap from="/image" to="/burgerimage" />
      <param name="history" value="keep_last" />
      <extra_arg name="use_intra_process_comms" value="true" />
    </composable_node>
  </node_container>
</launch>

YAML

launch:
  - node_container:
    pkg: rclcpp_components
    exec: component_container
    name: image_container
    namespace: ''
    composable_node:
      - pkg: image_tools
        plugin: image_tools::Cam2Image
        name: cam2image
        namespace: ''
        remap:
          - from: /image
            to: /burgerimage
        param:
          - name: width
            value: 320
          - name: height
            value: 240
          - name: burger_mode
            value: true
          - name: history
            value: keep_last
        extra_arg:
          - name: use_intra_process_comms
            value: 'true'
      - pkg: image_tools
        plugin: image_tools::ShowImage
        namespace: ''

(continues on next page)
Loading composable nodes into an existing container

Containers can sometimes be launched by other launch files or from a commandline. In that case, you need to add your components to an existing container. For this, you may use `LoadComposableNodes` to load components into a given container. The below example launches the same nodes as above.

**Python**

```python
from launch import LaunchDescription
from launch_ros.actions import LoadComposableNodes, Node
from launch_ros.descriptions import ComposableNode

def generate_launch_description():
    container = Node(
        name='image_container',
        package='rclcpp_components',
        executable='component_container',
        output='both',
    )

    load_composable_nodes = LoadComposableNodes(
        target_container='image_container',
        composable_node_descriptions=[
            ComposableNode(
                package='image_tools',
                plugin='image_tools::Cam2Image',
                name='cam2image',
                remappings=[('/image', '/burgerimage')],
                parameters=[{'width': 320, 'height': 240, 'burger_mode': True, 'history': 'keep_last'}],
                extra_arguments=[{'use_intra_process_comms': True}],
            ),
            ComposableNode(
                package='image_tools',
                plugin='image_tools::ShowImage',
                name='showimage',
                remappings=[('/image', '/burgerimage')],
                parameters=[{'history': 'keep_last'}],
                extra_arguments=[{'use_intra_process_comms': True}],
            ),
        )
```

(continues on next page)
),
],
)

return LaunchDescription([container, load_composable_nodes])

XML

<launch>
  <node pkg="rclcpp_components" exec="component_container" name="image_container">
    </node>
  <load_composable_node target="image_container">
    <composable_node pkg="image_tools" plugin="image_tools::Cam2Image" name="cam2image">
      <remap from="/image" to="/burgerimage" />
      <param name="width" value="320" />
      <param name="height" value="240" />
      <param name="burger_mode" value="true" />
      <param name="history" value="keep_last" />
      <extra_arg name="use_intra_process_comms" value="true" />
    </composable_node>
    <composable_node pkg="image_tools" plugin="image_tools::ShowImage" name="showimage" namespace="">
      <remap from="/image" to="/burgerimage" />
      <param name="history" value="keep_last" />
      <extra_arg name="use_intra_process_comms" value="true" />
    </composable_node>
  </load_composable_node>
</launch>

YAML

launch:
  - node_container:
      pkg: rclcpp_components
      exec: component_container
      name: image_container
      namespace: '
      composable_node:
        - pkg: image_tools
          plugin: image_tools::Cam2Image
          name: cam2image
          namespace: '
          remap:
            - from: /image
              to: /burgerimage
          param:
            - name: width
              value: 320
            - name: height
              value: 240
            - name: burger_mode

(continues on next page)
Using the Launch files from the command-line

Any of the launch files above can be run with `ros2 launch`. Copy the data into a local file, and then run:

```bash
ros2 launch <path_to_launch_file>
```

Intra-process communications

All of the above examples use an extra argument to setup intra-process communication between the nodes. For more information on what intra-process communications are, see the `intra-process comms tutorial`.

Python, XML, or YAML: Which should I use?

See the discussion in `Using Python, XML, and YAML for ROS 2 Launch Files` for more information.

Migrating YAML parameter files from ROS 1 to ROS 2

This guide describes how to adapt ROS 1 parameters files for ROS 2.
**YAML file example**

YAML is used to write parameters files in both ROS 1 and ROS 2. The main difference in ROS 2 is that node names must be used to address parameters. In addition to the fully qualified node name, we use the key "ros__parameters" to signal the start of parameters for the node.

For example, here is a parameters file in ROS 1:

```yaml
lidar_name: foo
lidar_id: 10
ports: [11312, 11311, 21311]
debug: true
```

Let's assume that the first two parameters are for a node named `/lidar_ns/lidar_node_name`, the next parameter is for a node named `/imu`, and the last parameter we want to set on both nodes.

We would construct our ROS 2 parameters file as follows:

```yaml
/lidar_ns:
  lidar_node_name:
    ros__parameters:
      lidar_name: foo
      id: 10

imu:
  ros__parameters:
    ports: [2438, 2439, 2440]
/**:
  ros__parameters:
    debug: true
```

Note the use of wildcards (`/**`) to indicate that the parameter `debug` should be set on any node in any namespace.

**Feature parity**

Some features of ROS 1 parameters files do not exist in ROS 2:

- Mixed types in a list is not supported yet (related issue)
- `deg` and `rad` substitutions are not supported

**Passing ROS arguments to nodes via the command-line**

All ROS nodes take a set of arguments that allow various properties to be reconfigured. Examples include configuring the name/namespace of the node, topic/service names used, and parameters on the node. All ROS-specific arguments have to be specified after a `--ros-args` flag:
ros2 run my_package node_executable --ros-args ...

For more details, see this design doc.

Name remapping

Names within a node (e.g. topics/services) can be remapped using the syntax `-r <old name>:=<new name>`. The name/namespace of the node itself can be remapped using `-r __node:=<new node name>` and `-r __ns:=<new node namespace>`.

Note that these remappings are “static” remappings, in that they apply for the lifetime of the node. “Dynamic” remapping of names after nodes have been started is not yet supported.

See this design doc for more details on remapping arguments (not all functionality is available yet).

Example

The following invocation will cause the `talker` node to be started under the node name `my_talker`, publishing on the topic named `my_topic` instead of the default of `chatter`. The namespace, which must start with a forward slash, is set to `/demo`, which means that topics are created in that namespace (`/demo/my_topic`), as opposed to globally (`/my_topic`).

```
ros2 run demo_nodes_cpp talker --ros-args -r __ns:=/demo -r __node:=my_talker -r ...
-...-r __chatter:=my_topic
```

Passing remapping arguments to specific nodes

If multiple nodes are being run within a single process (e.g. using `Composition`), remapping arguments can be passed to a specific node using its name as a prefix. For example, the following will pass the remapping arguments to the specified nodes:

```
ros2 run composition manual_composition --ros-args -r talker:__node:=my_talker -r ...
-...-r listener:__node:=my_listener
```

The following example will both change the node name and remap a topic (node and namespace changes are always applied before topic remapping):

```
ros2 run composition manual_composition --ros-args -r talker:__node:=my_talker -r my_ ...
-...-r talker:chatter:=my_topic -r listener:__node:=my_listener -r my_listener:chatter:=my_ ...
-...-r topic
```
Logger configuration

See \texttt{--log-level} argument usage in the \textit{logging page}.

Parameters

Setting parameters directly from the command line

You can set parameters directly from the command line using the following syntax:

\begin{verbatim}
ros2 run package_name executable_name \--ros-args -p param_name:=param_value
\end{verbatim}

As an example, you can run:

\begin{verbatim}
ros2 run demo_nodes_cpp parameter_blackboard \--ros-args -p some_int:=42 -p "a_string:=Hello world" -p "some_lists.some_integers:=[1, 2, 3, 4]" -p "some_lists.some_doubles:=[3.14, 2.718]"
\end{verbatim}

Other nodes will be able to retrieve the parameter values, e.g.:

\begin{verbatim}
$ ros2 param list parameter_blackboard
a_string
qos_overrides./parameter_events.publisher.depth
qos_overrides./parameter_events.publisher.durability
qos_overrides./parameter_events.publisher.history
qos_overrides./parameter_events.publisher.reliability
some_int
some_lists.some_doubles
some_lists.some_integers
use_sim_time
\end{verbatim}

Setting parameters from YAML files

Parameters can be set from the command-line in the form of yaml files. See here for examples of the yaml file syntax.

As an example, save the following as \texttt{demo_params.yaml}:

\begin{verbatim}
parameter_blackboard:
  ros__parameters:
    some_int: 42
    a_string: "Hello world"
    some_lists:
      some_integers: [1, 2, 3, 4]
      some_doubles: [3.14, 2.718]
\end{verbatim}

Then either declare the parameters within your node with \texttt{declare_parameter} or \texttt{declare_parameters} (see documentation for function signatures), or set the node to automatically declare parameters if they were passed in via a command line override.

Then run the following:
Other nodes will be able to retrieve the parameter values, e.g.:

```
$ ros2 param list parameter_blackboard
a_string
qos_overrides./parameter_events.publisher.depth
qos_overrides./parameter_events.publisher.durability
qos_overrides./parameter_events.publisher.history
qos_overrides./parameter_events.publisher.reliability
some_int
some_lists.some_doubles
some_lists.some_integers
use_sim_time
```

### Synchronous vs. asynchronous service clients

**Level:** Intermediate  
**Time:** 10 minutes

#### Contents

- Introduction  
- 1 Synchronous calls  
  - 1.1 Sync deadlock  
- 2 Asynchronous calls  
- Summary

#### Introduction

This guide is intended to warn users of the risks associated with the Python synchronous service client `call()` API. It is very easy to mistakenly cause deadlock when calling services synchronously, so we do not recommend using `call()`. We provide an example on how to use `call()` correctly for experienced users who wish to use synchronous calls and are aware of the pitfalls. We also highlight possible scenarios for deadlock that accompany it.

Because we recommend avoiding sync calls, this guide will also address the features and usage of the recommended alternative, async calls (`call_async()`).

The C++ service call API is only available in async, so the comparisons and examples in this guide pertain to Python services and clients. The definition of async given here generally applies to C++, with some exceptions.
1 Synchronous calls

A synchronous client will block the calling thread when sending a request to a service until a response has been received; nothing else can happen on that thread during the call. The call can take arbitrary amounts of time to complete. Once complete, the response returns directly to the client.

The following is an example of how to correctly execute a synchronous service call from a client node, similar to the async node in the *Simple Service and Client* tutorial.

```python
import sys
from threading import Thread

from example_interfaces.srv import AddTwoInts
import rclpy
from rclpy.node import Node

class MinimalClientSync(Node):
    def __init__(self):
        super().__init__('minimal_client_sync')
        self.cli = self.create_client(AddTwoInts, 'add_two_ints')
        while not self.cli.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('service not available, waiting again...')
        self.req = AddTwoInts.Request()

    def send_request(self):
        self.req.a = int(sys.argv[1])
        self.req.b = int(sys.argv[2])
        return self.cli.call(self.req)

    def main():
        rclpy.init()

        minimal_client = MinimalClientSync()

        spin_thread = Thread(target=rclpy.spin, args=(minimal_client,))
        spin_thread.start()

        response = minimal_client.send_request()
        minimal_client.get_logger().info(
            'Result of add_two_ints: for %d + %d = %d
            (minimal_client.req.a, minimal_client.req.b, response.sum))

        minimal_client.destroy_node()
        rclpy.shutdown()

    if __name__ == '__main__':
        main()
```

Note inside main() that the client calls rclpy.spin in a separate thread. Both send_request and rclpy.spin are
blocking, so they need to be on separate threads.

### 1.1 Sync deadlock

There are several ways that the synchronous `call()` API can cause deadlock.

As mentioned in the comments of the example above, failing to create a separate thread to spin `rclpy` is one cause of deadlock. When a client is blocking a thread waiting for a response, but the response can only be returned on that same thread, the client will never stop waiting, and nothing else can happen.

Another cause of deadlock is blocking `rclpy.spin` by calling a service synchronously in a subscription, timer callback or service callback. For example, if the synchronous client’s `send_request` is placed in a callback:

```python
def trigger_request(msg):
    response = minimal_client.send_request()  # This will cause deadlock
    minimal_client.get_logger().info(
        'Result of add_two_ints: for %d + %d = %d' %
        (minimal_client.req.a, minimal_client.req.b, response.sum))
subscription = minimal_client.create_subscription(String, 'trigger', trigger_request, 10)
rclpy.spin(minimal_client)
```

Deadlock occurs because `rclpy.spin` will not preempt the callback with the `send_request` call. In general, callbacks should only perform light and fast operations.

**Warning:** When deadlock occurs, you will not receive any indication that the service is blocked. There will be no warning or exception thrown, no indication in the stack trace, and the call will not fail.

### 2 Asynchronous calls

Async calls in `rclpy` are entirely safe and the recommended method of calling services. They can be made from anywhere without running the risk of blocking other ROS and non-ROS processes, unlike sync calls.

An asynchronous client will immediately return `future`, a value that indicates whether the call and response is finished (not the value of the response itself), after sending a request to a service. The returned `future` may be queried for a response at any time.

Since sending a request doesn’t block anything, a loop can be used to both spin `rclpy` and check `future` in the same thread, for example:

```python
while rclpy.ok():
    rclpy.spin_once(node)
    if future.done():
        # Get response
```

The Simple Service and Client tutorial for Python illustrates how to perform an async service call and retrieve the `future` using a loop.

The `future` can also be retrieved using a timer or callback, like in this example, a dedicated thread, or by another method. It is up to you, as the caller, to decide how to store `future`, check on its status, and retrieve your response.
Summary

It is not recommended to implement a synchronous service client. They are susceptible to deadlock, but will not provide any indication of issue when deadlock occurs. If you must use synchronous calls, the example in section 1 Synchronous calls is a safe method of doing so. You should also be aware of the conditions that cause deadlock outlined in section 1.1 Sync deadlock. We recommend using async service clients instead.

DDS tuning information

This page provides some guidance on parameter tunings that were found to address issues faced while using various DDS implementations on Linux in real-world situations. It is possible that the issues we identified on Linux or while using one vendor may occur for other platforms and vendors not documented here.

The recommendations below are starting points for tuning; they worked for specific systems and environments, but the tuning may vary depending on a number of factors. You may need to increase or decrease values while debugging relative to factors like message size, network topology, etc.

It is important to recognize that tuning parameters can come at a cost to resources, and may affect parts of your system beyond the scope of the desired improvements. The benefits of improving reliability should be weighed against any detriments for each individual case.

Cross-vendor tuning

**Issue:** Sending data over lossy (usually WiFi) connections becomes problematic when some IP fragments are dropped, possibly causing the kernel buffer on the receiving side to become full.

When a UDP packet is missing at least one IP fragment, the rest of the received fragments fill up the kernel buffer. By default, the Linux kernel will timeout after 30s of trying to recombine packet fragments. Since the kernel buffer is full at this point (default size is 256KB), no new fragments can come in, and so the connection will seemingly “hang” for long periods of time.

This issue is generic across all DDS vendors, so the solutions involve adjusting kernel parameters.

**Solution:** Use best-effort QoS settings instead of reliable.

Best-effort settings reduce the amount of network traffic since the DDS implementation does not have to incur the overhead of reliable communications, where publishers require acknowledgements for messages sent to subscribers and must resend samples that have not been properly received.

If the kernel buffer for IP fragments gets full, though, the symptom is still the same (blocking for 30s). This solution should improve the issue somewhat without having to adjust parameters.

**Solution:** Reduce the value of the `ipfrag_time` parameter.

```
net.ipv4.ipfrag_time / /proc/sys/net/ipv4/ipfrag_time (default 30s) : Time in seconds to keep an IP fragment in memory.
```

Reduce the value, for example, to 3s, by running:

```
sudo sysctl net.ipv4.ipfrag_time=3
```

Reducing this parameter’s value also reduces the window of time where no fragments are received. The parameter is global for all incoming fragments, so the feasibility of reducing its value needs to be considered for every environment.

**Solution:** Increase the value of the `ipfrag_high_thresh` parameter.

```
```

```
Increase the value, for example, to 128MB, by running:

```
sudo sysctl net.ipv4.ipfrag_high_thresh=134217728 # (128 MB)
```

Significantly increasing this parameter’s value is an attempt to ensure that the buffer never becomes completely full. However, the value would likely have to be significantly high to hold all data received during the time window of `ipfrag_time`, assuming every UDP packet lacks one fragment.

**Issue:** Sending custom messages with large variable-sized arrays of non-primitive types causes high serialization/deserialization overhead and CPU load. This can lead to stalling of the publisher due to excessive time spent in `publish()` and tools like `ros2 topic hz` under reporting the actual frequency of messages being received. Note that for example `builtin_interfaces/Time` is also considered a non-primitive type and will incur higher serialization overhead. Because of the increased serialization overhead, severe performance degradation can be observed when naively transitioning custom message types from ROS 1 to ROS 2.

**Workaround:** Use multiple arrays of primitives instead of a single array of custom types, or pack into byte array as done e.g. in `PointCloud2` messages. For example, instead of defining a `FooArray` message as:

```
Foo[] my_large_array
```

with `Foo` is defined as:

```
uint64 foo_1
uint32 foo_2
```

Instead, define `FooArray` as:

```
uint64[] foo_1_array
uint32[] foo_2_array
```

**Fast RTPS tuning**

**Issue:** Fast RTPS floods the network with large pieces of data or fast-published data when operating over WiFi.

See the solutions under *Cross-vendor tuning*.

**Cyclone DDS tuning**

**Issue:** Cyclone DDS is not delivering large messages reliably, despite using reliable settings and transferring over a wired network.

This issue should be addressed soon. Until then, we’ve come up with the following solution (debugged using this test program):

**Solution:** Increase the maximum Linux kernel receive buffer size and the minimum socket receive buffer size that Cyclone uses.

*Adjustments to solve for a 9MB message:*

Set the maximum receive buffer size, `rmem_max`, by running:

```
sudo sysctl -w net.core.rmem_max=2147483647
```

Or permanently set it by editing the `/etc/sysctl.d/10-cyclone-max.conf` file to contain:
Next, to set the minimum socket receive buffer size that Cyclone requests, write out a configuration file for Cyclone to use while starting, like so:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
  <Domain id="any">
    <Internal>
      <MinimumSocketReceiveBufferSize>10MB</MinimumSocketReceiveBufferSize>
    </Internal>
  </Domain>
</CycloneDDS>
```

Then, whenever you are going to run a node, set the following environment variable:

`CYCLONEDDS_URI=file:///absolute/path/to/config_file.xml`

### RTI Connext tuning

**Issue:** Connext is not delivering large messages reliably, despite using reliable settings and transferring over a wired network.

**Solution:** This Connext QoS profile, along with increasing the `rmem_max` parameter.

Set the maximum receive buffer size, `rmem_max`, by running:

```
sudo sysctl -w net.core.rmem_max=4194304
```

By tuning `net.core.rmem_max` to 4MB in the Linux kernel, the QoS profile can produce truly reliable behavior.

This configuration has been proven to reliably deliver messages via SHMEM|UDPv4, and just UDPv4 on a single machine. A multi-machine configuration was also tested with `rmem_max` at 4MB and at 20MB (two machines connected with 1Gbps ethernet), with no dropped messages and average message delivery times of 700ms and 371ms, respectively.

Without configuring the kernel’s `rmem_max`, the same Connext QoS profile took up to 12 seconds for the data to be delivered. However, it always at least managed to complete the delivery.

**Solution:** Use the Connext QoS profile without adjusting `rmem_max`.

The `ROS2TEST_QOS_PROFILES.xml` file was configured using RTI’s documentation on configuring flow controllers. It has slow, medium and fast flow controllers (seen in the Connext QoS profile link).

The medium flow controller produced the best results for our case. However, the controllers will still need to be tuned for the particular machine/network/environment they are operating in. The Connext flow controllers can be used to tune bandwidth and its aggressiveness for sending out data, though once the bandwidth of a particular setup is passed, performance will start to drop.
rosbag2: Overriding QoS Policies

Goal: Override Ros2Bag QoS profile settings for recording and playback.

Background

With the introduction of DDS in ROS 2, Quality of Service (QoS) compatibility for publisher/subscriber nodes needs to be considered when recording and playing back data. More detail on how QoS works can be found here. For the purposes of this guide, it is sufficient to know that only the reliability and durability policies affect whether publishers/subscribers are compatible and can receive data from one other.

Ros2Bag adapts its requested/offered QoS profile when recording/playing data from a topic to prevent dropped messages. During playback, Ros2bag also attempts to preserve the policy originally offered by the topic. Certain situations may require specifying explicit QoS profile settings so Ros2Bag can record/playback topics. These QoS profile overrides can be specified via the CLI using the --qos-profile-overrides-path flag.

Using QoS Overrides

The YAML schema for the profile overrides is a dictionary of topic names with key/value pairs for each QoS policy:

```yaml
- topic_name: str
  qos_policy_name: str
  ...
  qos_duration: object
    sec: int
    nsec: int
```

If a policy value is not specified, the value will fallback to the default used by Ros2Bag. If you specify a Duration based policy such as deadline or lifespan, you will need to specify both seconds and nanoseconds. Policy values are determined by the policy’s short keys which can be found using ros2topic verbs such as ros2 topic pub --help. All values are replicated below for reference.

```yaml
history: [keep_all, keep_last]
depth: int
reliability: [system_default, reliable, best_effort, unknown]
durability: [system_default, transient_local, volatile, unknown]
deadline:
  sec: int
  nsec: int
lifespan:
  sec: int
  nsec: int
liveliness: [system_default, automatic, manual_by_topic, unknown]
liveliness_lease_duration:
```
Example

Consider a topic `/talker` offering a `transient_local` Durability policy. ROS 2 publishers by default request volatile Durability.

```
ros2 topic pub -r 0.1 --qos-durability transient_local /talker std_msgs/String "data: Hello World"
```

In order for Ros2Bag to record the data, we would want to override the recording policy for that specific topic like so:

```
# durability_override.yaml
/talker:
  durability: transient_local
  history: keep_all
```

And call it from the CLI:

```
ros2 bag record -a -o my_bag --qos-profile-overrides-path durability_override.yaml
```

If we want to playback the bag file but with a different Reliability policy, we can specify one as such;

```
# reliability_override.yaml
/talker:
  reliability: best_effort
  history: keep_all
```

And call it from the CLI:

```
ros2 bag play --qos-profile-overrides-path reliability_override.yaml my_bag
```

We can see the results with `ros2 topic`

```
ros2 topic echo --qos-reliability best_effort /talker std_msgs/String
```

Working with multiple ROS 2 middleware implementations

Table of Contents

- Prerequisites
- Specifying RMW implementations
- Adding RMW implementations to your workspace
- Troubleshooting
  - Checking the Current RMW
This page explains the default RMW implementation and how to specify an alternative.

**Prerequisites**

You should have already read the *DDS and ROS middleware implementations page*.

**Specifying RMW implementations**

To have multiple RMW implementations available for use you must have installed the ROS 2 binaries and any additional dependencies for specific RMW implementations, or built ROS 2 from source with multiple RMW implementations in the workspace (the RMW implementations are included in the build by default if their compile-time dependencies are met). See *Install DDS implementations*.

Both C++ and Python nodes support an environment variable `RMW_IMPLEMENTATION` that allows the user to select the RMW implementation to use when running ROS 2 applications.

The user may set this variable to a specific implementation identifier, such as `rmw_cyclonedds_cpp`, `rmw_fastrtps_cpp`, `rmw_connextdds`, or `rmw_gurumdds_cpp`.

For example, to run the talker demo using the C++ talker and Python listener with the Connext RMW implementation:

**Linux**

```bash
RMW_IMPLEMENTATION=rmw_connextdds ros2 run demo_nodes_cpp talker
# Run in another terminal
RMW_IMPLEMENTATION=rmw_connextdds ros2 run demo_nodes_py listener
```

**macOS**

```bash
RMW_IMPLEMENTATION=rmw_connextdds ros2 run demo_nodes_cpp talker
# Run in another terminal
RMW_IMPLEMENTATION=rmw_connextdds ros2 run demo_nodes_py listener
```

**Windows**

```bash
set RMW_IMPLEMENTATION=rmw_connextdds
ros2 run demo_nodes_cpp talker
REM run in another terminal
set RMW_IMPLEMENTATION=rmw_connextdds
ros2 run demo_nodes_py listener
```
Adding RMW implementations to your workspace

Suppose that you have built your ROS 2 workspace with only Fast DDS installed and therefore only the Fast DDS RMW implementation built. The last time your workspace was built, any other RMW implementation packages, `rmw_connextd` for example, were probably unable to find installations of the relevant DDS implementations. If you then install an additional DDS implementation, Connext for example, you will need to re-trigger the check for a Connext installation that occurs when the Connext RMW implementation is being built. You can do this by specifying the `--cmake-clean-cache` flag on your next workspace build, and you should see that the RMW implementation package then gets built for the newly installed DDS implementation.

It is possible to run into a problem when “rebuilding” the workspace with an additional RMW implementation using the `--cmake-clean-cache` option where the build complains about the default RMW implementation changing. To resolve this, you can either set the default implementation to what is was before with the `RMW_IMPLEMENTATION` CMake argument or you can delete the build folder for packages that complain and continue the build with `--packages-start <package name>`.

Troubleshooting

Checking the Current RMW

To check the RMW that is currently in use you simply check the `RMW_IMPLEMENTATION` environment variable. On Linux systems `printenv` prints the full list of environment variables. Other operating systems will have other procedures for viewing environment variables. If `RMW_IMPLEMENTATION` is not in the environment it is safe to assume you are using the default for your ROS distro, otherwise the current RMW is the value listed. The default RMW for each ROS Distro can be found in `REP-2000`.

Ensuring use of a particular RMW implementation

If the `RMW_IMPLEMENTATION` environment variable is set to an RMW implementation for which support is not installed, you will see an error message similar to the following if you have only one implementation installed:

```
Expected RMW implementation identifier of 'rmw_connextd' but instead found 'rmw_fastrtps_cpp', exiting with 102.
```

If you have support for multiple RMW implementations installed and you request use of one that is not installed, you will see something similar to:

```
Error getting RMW implementation identifier / RMW implementation not installed (expected_identifier of 'rmw_connextd'), exiting with 1.
```

If this occurs, double check that your ROS 2 installation includes support for the RMW implementation that you have specified in the `RMW_IMPLEMENTATION` environment variable.

If you want to switch between RMW implementations, verify that the ROS 2 daemon process is not running with the previous RMW implementation to avoid any issues between nodes and command line tools such as `ros2 node`. For example, if you run:

```
RMW_IMPLEMENTATION=rmw_connextd ros2 run demo_nodes_cpp talker
```

and

```
ros2 node list
```
it will generate a daemon with a Fast DDS implementation:

```
21318  22.0  0.6  535896  55044 pts/8  Sl  16:14  0:00 /usr/bin/python3 /opt/ros/iron/bin/_ros2_daemon --rmw-implementation rmw_fastrtps_cpp --ros-domain-id 0
```

Even if you run the command line tool again with the correct RMW implementation, the daemon's RMW implementation will not change and the ROS 2 command line tools will fail.

To solve this, simply stop the daemon process:

```
ros2 daemon stop
```

and rerun the ROS 2 command line tool with the correct RMW implementation.

**RTI Connext on OSX: Failure due to insufficient shared memory kernel settings**

If you receive an error message similar to below when running RTI Connext on OSX:

```
[D0062|ENABLE]DDS_DomainParticipantPresentationreserve_participant_index_entryports:!
   --enable reserve participant index
[D0062|ENABLE]DDS_DomainParticipantreserve_participant_index_entryports:Unusable shared
   --memory transport. For a more in-depth explanation of the possible problem and
   --solution, please visit https://community.rti.com/kb/osx510.
```

This error is caused by an insufficient number or size of shared memory segments allowed by the operating system. As a result, the DomainParticipant is unable to allocate enough resources and calculate its participant index which causes the error.

You can increase the shared memory resources of your machine either temporarily or permanently.

To increase the settings temporarily, you can run the following commands as user root:

```
/usr/sbin/sysctl -w kern.sysv.shmmax=419430400
/usr/sbin/sysctl -w kern.sysv.shmmin=1
/usr/sbin/sysctl -w kern.sysv.shmmni=128
/usr/sbin/sysctl -w kern.sysv.shmseg=1024
/usr/sbin/sysctl -w kern.sysv.shmall=262144
```

To increase the settings permanently, you will need to edit or create the file `/etc/sysctl.conf`. Creating or editing this file will require root permissions. Either add to your existing `/etc/sysctl.conf` file or create a new `/etc/sysctl.conf` with the following lines:

```
kern.sysv.shmmax=419430400
kern.sysv.shmmin=1
kern.sysv.shmmni=128
kern.sysv.shmseg=1024
kern.sysv.shmall=262144
```

You will need to reboot the machine after modifying this file to have the changes take effect.

This solution is edited from the RTI Connext community forum. See the original post for more detailed explanation.
Cross-compilation

Table of Contents

- cross_compile tool
- Legacy tool instructions
  - CMake toolchain-file
  - Target file-system
  - Build process
  - Cross-compiling examples for Arm
  - Automated Cross-compilation
  - Cross-compiling against a pre-built ROS 2
  - Run on the target

For background information on cross-compilation, see the conceptual article.

This document provides you with details on how to cross-compile the ROS 2 software stack as well as provide examples for cross-compiling to systems based on the Arm cores.

cross_compile tool

Instructions to use the tool are in the cross_compile package.

Legacy tool instructions

Note: Follow the steps below only if you are using the old version (release 0.0.1) of the cross-compile tool. For all other purposes, follow the cross_compile package documentation.

Although ROS 2 is a rich software stack with a number of dependencies, it primarily uses two different types of packages:

- Python based software, which requires no cross-compilation.
- CMake based software, which provides a mechanism to do cross-compilation.

Furthermore, the ROS 2 software stack is built with Colcon which provides a mechanism to forward parameters to the CMake instance used for the individual build of each package/library that is part of the ROS 2 distribution.

When building ROS 2 natively, the developer is required to download all the dependencies (e.g. Python and other libraries) before compiling the packages that are part of the ROS 2 distribution. When cross-compiling, the same approach is required. The developer must first have the target system’s filesystem with all dependencies already installed.

The next sections of this document explain in detail the use of cmake-toolchains and the CMAKE_SYSROOT feature to cross-compile ROS 2.
**CMake toolchain-file**

A CMake toolchain-file is a file which defines variables to configure CMake for cross-compilation. The basic entries are:

- **CMAKE_SYSTEM_NAME**: the target platform, e.g. `linux`
- **CMAKE_SYSTEM_PROCESSOR**: the target architecture, e.g. `aarch64` or `arm`
- **CMAKE_SYSROOT**: the path to the target file-system
- **CMAKE_C_COMPILER**: the C cross-compiler, e.g. `aarch64-linux-gnu-gcc`
- **CMAKE_CXX_COMPILER**: the C++ cross-compiler, e.g. `aarch64-linux-gnu-g++`
- **CMAKE_FIND_ROOT_PATH**: an alternative path used by the `find_*` command to find the file-system

When cross-compiling ROS 2, the following options are required to be set:

- **CMAKE_FIND_ROOT_PATH**: the alternative path used by the `find_*` command, use it to specify the path to ROS 2 `/install` folder
- **CMAKE_FIND_ROOT_PATH_MODE_***: the search strategy for program, package, library, and include, usually: `NEVER` (look on the host-fs), `ONLY` (look on sysroot), and `BOTH` (look on both sysroot and host-fs)
- **PYTHON_SOABI**: the index name of the python libraries generated by ROS 2, e.g. `cpython-36m-aarch64-linux-gnu`
- **THREADS_PTHREAD_ARG "0" CACHE STRING "Result from TRY_RUN" FORCE**: Force the result of the `TRY_RUN` cmd to 0 (success) because binaries can not run on the host system.

The toolchain-file is provided to CMake with the `-DCMAKE_TOOLCHAIN_FILE=path/to/file` parameter. This will also set the **CMAKE_CROSSCOMPILING** variable to `true` which can be used by the software being built.

The **CMAKE_SYSROOT** is particularly important for ROS 2 as the packages need many dependencies (e.g. python, openssl, opencv, poco, eigen3, …). Setting **CMAKE_SYSROOT** to a target file-system with all the dependencies installed on it will allow CMake to find them during the cross-compilation.

---

**Note:** You can find more information on the CMake documentation page.

---

When downloading the ROS 2 source code, a generic toolchain-file is available in the repository `ros-tooling/cross_compile/cmake-toolchains` which can be downloaded separately. Further examples on using it can be found on the Cross-compiling examples for Arm section.

**Target file-system**

As mentioned previously, ROS 2 requires different libraries which needs to be provided to cross-compile.

**There are a number of ways to obtain the file-system:**

- downloading a pre-built image
- installing the dependencies on the target and exporting the file-system (e.g. with sshfs)
- using qemu + docker (or chroot) to generate the file-system on the host machine.

---

**Note:** You can find information on how to use Docker + qemu on the next Cross-compiling examples for Arm section.
**Build process**

The build process is similar to native compilation. The only difference is an extra argument to Colcon to specify the toolchain-file:

```
colcon build --merge-install \
   --cmake-force-configure \
   --cmake-args \n   -DCMAKE_TOOLCHAIN_FILE="<path_to_toolchain/toolchainfile.cmake>"
```

The toolchain-file provide to CMake the information of the cross-compiler and the target file-system. Colcon will call CMake with the given toolchain-file on every package of ROS 2.

**Cross-compiling examples for Arm**

After downloading the ROS 2 source code, you can add cross-compilation assets to the workspace via `git clone https://github.com/ros-tooling/cross_compile.git -b 0.0.1 src/ros2/cross_compile`. These are working examples on how to cross-compile for Arm cores.

The following targets are supported:

- Ubuntu-arm64: To be used with any ARMv8-A based system.
- Ubuntu-armhf: To be used with any modern ARMv7-A based system.

These are the main steps:

- Installing development tools
- Downloading ROS 2 source code
- Downloading the ROS 2 cross-compilation assets
- Preparing the sysroot
- Cross-compiling the ROS 2 software stack

The next sections explains in detail each of these steps. For a quick-setup, have a look at the Automated Cross-compilation.

**Note:** These steps were tested on an Ubuntu 18.04 (Bionic)

### 1. Install development tools

This step is similar to when building natively. The difference is that some of the libraries and tools are not required because they will be in the sysroot instead. The following packages are required:

```
sudo apt update && sudo apt install -y \
cmake \n  git \n  wget \n  python3-pip \n  qemu-user-static \n  g++-aarch64-linux-gnu \n  g++-arm-linux-gnueabihf
```

(continues on next page)
pkg-config-aarch64-linux-gnu

```bash
python3 -m pip install -U \
vcs\tool \n
colcon-common-extensions
```

**Note:** You can install vcs\tool and colcon-common-extensions via pip. This means you are not required to add extra apt repositories.

Docker is used to build the target environment. Follow the official documentation for the installation.

### 2. Download ROS 2 source code

Then create a workspace and download the ROS 2 source code:

```bash
mkdir -p ~/cc_ws/ros2_ws/src
cd ~/cc_ws/ros2_ws
vcs-import src < ros2.repos
git clone https://github.com/ros-tooling/cross_compile.git -b 0.0.1 src/ros2/cross_compile
```

```bash
cd ..
```

### 3. Prepare the sysroot

Build an arm Ubuntu image with all the ROS 2 dependencies using Docker and qemu: Copy the qemu-static binary to the workspace. It will be used to install the ROS 2 dependencies on the target file-system with docker.

```bash
mkdir qemu-user-static
cp /usr/bin/qemu-*static qemu-user-static
```

The standard `setup` process of ROS 2 is run inside an arm docker. This is possible thanks to qemu-static, which will emulate an arm machine. The base image used is an Ubuntu Bionic from Docker Hub.

```bash
docker build -t arm_ros2:latest -f ros2_ws/src/ros2/cross_compile/sysroot/Dockerfile_ubuntu_arm .
docker run --name arm_sysroot arm_ros2:latest
```

Export the resulting container to a tarball and extract it:

```bash
docker container export -o sysroot_docker.tar arm_sysroot
mkdir sysroot_docker
tar -C sysroot_docker -xf sysroot_docker.tar lib usr opt etc
docker rm arm_sysroot
```

This container can be used later as virtual target to run the created file-system and run the demo code.
4. Build

Set the variables used by the generic toolchain-file

```bash
export TARGET_ARCH=aarch64
export TARGET_TRIPLE=aarch64-linux-gnu
export CC=/usr/bin/$TARGET_TRIPLE-gcc
export CXX=/usr/bin/$TARGET_TRIPLE-g++
export CROSS_COMPILE=/usr/bin/$TARGET_TRIPLE-
export SYSROOT=~/.cc_ws/sysroot_docker
export ROS2_INSTALL_PATH=~/.cc_ws/ros2_ws/install
export PYTHON_SOABI=cpython-36m-$TARGET_TRIPLE
```

The following packages still cause errors during the cross-compilation (under investigation) and must be disabled for now.

```bash
touch
  ros2_ws/src/ros2/rviz/COLCON_IGNORE
  ros2_ws/src/ros-visualization/COLCON_IGNORE
```

The Poco pre-built has a known issue where it is searching for libz and libpcre on the host system instead of SYSROOT. As a workaround for the moment, please link both libraries into the host’s file-system.

```bash
mkdir -p /usr/lib/$TARGET_TRIPLE
ln -s `pwd`/sysroot_docker/lib/$TARGET_TRIPLE/libz.so.1 /usr/lib/$TARGET_TRIPLE/libz.so
ln -s `pwd`/sysroot_docker/lib/$TARGET_TRIPLE/libpcre.so.3 /usr/lib/$TARGET_TRIPLE/libpcre.so
```

Then, start a build with colcon specifying the toolchain-file:

```bash
cd ros2_ws

colcon build --merge-install
  --cmake-force-configure
  --cmake-args
    -DCMAKE_VERBOSE_MAKEFILE:BOOL=ON
    -DCMAKE_TOOLCHAIN_FILE="$(pwd)/src/ros2/cross_compile/cmake-toolchains/generic_
      linux.cmake"
    -DSECURITY=ON
```

Done! The install and build directories will contain the cross-compiled assets.

**Automated Cross-compilation**

All the steps above are also included into a Dockerfile and can be used for automation/CI.

First, download the dockerfile and build the image:

```bash
wget https://raw.githubusercontent.com/ros-tooling/cross_compile/master/Dockerfile_cc_for_arm
docker build -t ros2-crosscompiler:latest - < Dockerfile_cc_for_arm
```

Now run the image with: (it will take a while !)
Vulcanexus Documentation, Release 1.0.0

docker run -it --name ros2_cc \\  
   -v /var/run/docker.sock:/var/run/docker.sock \\  
   ros2-crosscompiler:latest

..note:: The -v /var/run/docker.sock allow us to use Docker inside Docker.

The result of the build will be inside the ros2_ws directory, which can be exported with:

docker cp ros2_cc:/root/cc_ws/ros2_ws .

Cross-compiling against a pre-built ROS 2

It is possible to cross-compile your packages against a pre-built ROS 2. The steps are similar to the previous Cross-compiling examples for Arm section, with the following modifications:

Instead of downloading the ROS 2 stack, just populate your workspace with your package (ros2 examples on this case) and the cross-compilation assets:

```
mkdir -p ~/cc_ws/ros2_ws/src
cd ~/cc_ws/ros2_ws/src
git clone https://github.com/ros2/examples.git
git clone https://github.com/ros-tooling/cross_compile.git -b 0.0.1
cd ..
```

Generate and export the file-system as described in 3. Prepare the sysroot, but with the provided Dockerfile_ubuntu_arm64_prebuilt. These _prebuilt Dockerfile will use the binary packages to install ROS 2 instead of building from source.

Modify the environment variable ROS2_INSTALL_PATH to point to the installation directory:

```
export ROS2_INSTALL_PATH=~/cc_ws/sysroot_docker/opt/ros/crystal
```

Source the setup.bash script on the target file-system:

```
source $ROS2_INSTALL_PATH/setup.bash
```

Then, start a build with Colcon specifying the toolchain-file:

```
colcon build \\  
   --merge-install \\  
   --cmake-force-configure \\  
   --cmake-args \\  
      -DCMAKE_VERBOSE_MAKEFILE:BOOL=ON \\  
      -DCMAKE_TOOLCHAIN_FILE="$(pwd)/src/cross_compile/cmake-toolchains/generic_linux.
   --cmake"
```
Run on the target

Copy the file-system on your target or use the previously built docker image:

```
docker run -it --rm -v `pwd`:/ros2_ws:/ros2_ws arm_ros2:latest
```

Source the environment:

```
source /ros2_ws/install/local_setup.bash
```

Run some of the C++ or python examples:

```
ros2 run demo_nodes_cpp listener &
ros2 run demo_nodes_py talker
```

Releasing a Package

First Time Release

This guide explains how to release ROS 2 packages that you have not released before. Due to numerous options available when releasing ROS packages, this guide intends to cover the most common scenario and does not cover every corner-case.

Table of Contents

- Be part of a release team
- Create a new release repository
- Install dependencies
- Set Up a Personal Access Token
- Ensure repositories are up-to-date
- Generate Changelog
- Bump the package version
- Bloom Release
- Next Steps

Be part of a release team

You must be part of a release team. If you are not part of a release team yet, follow either:

- Join a release team
- Start a new release team
Create a new release repository

You need a release repository to release a package. Follow Create a new release repository.

Install dependencies

Install tools that you will use in the upcoming steps according to your platform:

Debian (eg. Ubuntu)

```
sudo apt install python3-bloom python3-catkin-pkg
```

RPM (eg. RHEL)

```
sudo dnf install python3-bloom python3-catkin_pkg
```

Other

```
pip3 install -U bloom catkin_pkg
```

Set Up a Personal Access Token

**Warning:** If the file ~/.config/bloom exists on your computer, it is likely that you have done this before so you should skip this section.

During the release process, multiple HTTPS Git operations will be performed that require password authentication. To avoid being repeatedly asked for a password, a Personal Access Token (PAT) will be set up. If you have multi-factor authentication setup on your GitHub account, you must setup a Personal Access Token.

Create a Personal Access Token by:

1. Log in to GitHub and go to Personal access tokens.
2. Click the Generate new token button.
3. Set Note to something like Bloom token.
4. Set Expiration to No expiration.
5. Tick the public_repo and workflow checkboxes.
6. Click the Generate token button.

After you have created the token, you will end up back at the Personal access tokens page. Copy the alphanumeric token that is highlighted in green.

Save your GitHub username and PAT to a new file called ~/.config/bloom, with the format below:

```json
{
  "github_user": "<your-github-username>",
  "oauth_token": "<token-you-created-for-bloom>"
}
```
Ensure repositories are up-to-date

Make sure that:

- Your repository is hosted on a remote such as GitHub.
- You have a clone of the repository on your computer and are on the right branch.
- Both the remote repository and your clone are up-to-date.

Generate Changelog

Generate a CHANGELOG.rst file per package in your repo using the following command:

```
$ catkin_generate_changelog --all
```

Open all CHANGELOG.rst files in an editor. You will see that `catkin_generate_changelog` has auto-generated a forthcoming section with notes from commit messages:

```
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Changelog for package your_package
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Forthcoming
-----------
* you can modify this commit message
* and this
```

Clean up the list of commit messages to concisely convey the notable changes that have been made to the packages since the last release, and commit all the CHANGELOG.rst files. Do not modify the Forthcoming header.

Bump the package version

Every release of the package must have a unique version number higher than the previous release. Run:

```
$ catkin_prepare_release
```

which performs the following:

1. increases the package version in `package.xml`
2. replaces the heading Forthcoming with version (date) (eg. `0.0.1 (2022-01-08)` in `CHANGELOG.rst`
3. commits those changes
4. creates a tag (eg. `0.0.1`)
5. pushes the changes and the tag to your remote repository

Note: By default the patch version of the package is incremented, such as from `0.0.0` to `0.0.1`. To increment the minor or major version instead, run `catkin_prepare_release --bump minor` or `catkin_prepare_release --bump major`. For more details, see `catkin_prepare_release --help`.

---

1200 Chapter 4. Contributing to the documentation
Bloom Release

Run the following command, replacing `my_repo` with the name of your repository:

```
bloom-release --new-track --rosdistro iron --track iron my_repo
```

**Tip:**
- `--new-track` tells bloom to create a new track and configure it.
- `--rosdistro iron` indicates that this release is for the iron distro.
- `--track iron` indicates that you want the track name to be iron.

You will be prompted to enter information to configure a new track. In a common scenario such as:

- Your packages are in a repository called `my_repo`
- You are releasing a branch called `main`
- The repository is hosted on GitHub at `https://github.com/my_organization/my_repo.git`
- Your release repository is at `https://github.com/ros2-gbp/my_repo-release.git`

You should respond to the prompts as following:

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repository Name</td>
<td><code>my_repo</code></td>
</tr>
<tr>
<td>Upstream Repository URI</td>
<td><code>https://github.com/my_organization/my_repo.git</code></td>
</tr>
<tr>
<td>Upstream VCS Type</td>
<td></td>
</tr>
<tr>
<td>Version</td>
<td></td>
</tr>
<tr>
<td>Release Tag</td>
<td></td>
</tr>
<tr>
<td>Upstream Devel Branch</td>
<td><code>main</code></td>
</tr>
<tr>
<td>ROS Distro</td>
<td></td>
</tr>
<tr>
<td>Patches Directory</td>
<td></td>
</tr>
<tr>
<td>Release Repository Push URL</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** An empty cell in the table indicates that the default value should be used. Simply respond to the prompt by pressing Enter.

Bloom will automatically create a pull request for you against `rosdistro`.

**Next Steps**

Once your pull request has been submitted, usually within one or two days, one of the maintainers of rosdistro will review and merge your Pull Request. If your package build is successful, in 24-48 hours your packages will become available in the `ros-testing` repository, where you can test your pre-release binaries.

Approximately every two to four weeks, the distribution’s release manager manually synchronizes the contents of ros-testing into the main ROS repository. This is when your packages actually become available to the rest of the ROS community. To get updates on when the next synchronization (sync) is coming, subscribe to the Packaging and Release Management Category on ROS Discourse.
Subsequent Releases

This guide explains how to release new versions of ROS packages that have already been released before.

Table of Contents

- Be part of the release team
- Install dependencies
- Set up a Personal Access Token
- Ensure repositories are up-to-date
- Updating Changelog
- Bump the package version
- Bloom Release
- Next Steps

Be part of the release team

If you are not part of the release team that has write access to the release repository, follow Join a release team.

Install dependencies

Install tools that you will use in the upcoming steps according to your platform:

Debian (eg. Ubuntu)

```bash
sudo apt install python3-bloom python3-catkin-pkg
```

RPM (eg. RHEL)

```bash
sudo dnf install python3-bloom python3-catkin-pkg
```

Other

```bash
pip3 install -U bloom catkin_pkg
```

Set up a Personal Access Token

**Warning:** If the file `~/.config/bloom` exists on your computer, it is likely that you have done this before so you should skip this section.

During the release process, multiple HTTPS Git operations will be performed that require password authentication. To avoid being repeatedly asked for a password, a Personal Access Token (PAT) will be set up. If you have multi-factor authentication setup on your GitHub account, you **must** setup a Personal Access Token.

Create a Personal Access Token by:
1. Log in to GitHub and go to Personal access tokens.
2. Click the Generate new token button.
3. Set Note to something like Bloom token.
4. Set Expiration to No expiration.
5. Tick the public_repo and workflow checkboxes.
6. Click the Generate token button.

After you have created the token, you will end up back at the Personal access tokens page. Copy the alphanumeric token that is highlighted in green.

Save your GitHub username and PAT to a new file called ~/.config/bloom, with the format below:

```{ 
  "github_user": "<your-github-username>",
  "oauth_token": "<token-you-created-for-bloom>"
}
```

### Ensure repositories are up-to-date

Make sure that:

- Your repository is hosted on a remote such as GitHub.
- You have a clone of the repository on your computer and are on the right branch.
- Both the remote repository and your clone are up-to-date.

### Updating Changelog

For your users and for the developers, keep the changelog concise and up to date.

```bash
catkin_generate_changelog
```

Open all CHANGELOG.rst files in an editor. You will see that catkin_generate_changelog has auto-generated a forthcoming section with notes from commit messages:

```
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Changelog for package your_package
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Forthcoming
-----------
* you can modify this commit message
* and this
```

Clean up the list of commit messages to concisely convey the notable changes that have been made to the packages since the last release, and commit all the CHANGELOG.rst files. Do not modify the Forthcoming header.
**Bump the package version**

Every release of the package must have a unique version number higher than the previous release. Run:

```
catkin_prepare_release
```

which performs the following:

1. increases the package version in `package.xml`
2. replaces the heading `Forthcoming` with `version (date)` (eg. `0.0.1 (2022-01-08)`) in `CHANGELOG.rst`
3. commits those changes
4. creates a tag (eg. `0.0.1`)
5. pushes the changes and the tag to your remote repository

**Note:** By default the patch version of the package is incremented, such as from `0.0.0` to `0.0.1`. To increment the minor or major version instead, run `catkin_prepare_release --bump minor` or `catkin_prepare_release --bump major`. For more details, see `catkin_prepare_release --help`.

**Bloom Release**

Run the following command, replacing `my_repo` with the name of your repository with the packages:

```
bloom-release --rosdistro iron my_repo
```

Bloom will automatically create a pull request for you against `rosdistro`.

**Next Steps**

Once your pull request has been submitted, usually within one or two days, one of the maintainers of `rosdistro` will review and merge your Pull Request. If your package build is successful, in 24-48 hours your packages will become available in the `ros-testing` repository, where you can test your pre-release binaries.

Approximately every two to four weeks, the distribution's release manager manually synchronizes the contents of `ros-testing` into the main ROS repository. This is when your packages actually become available to the rest of the ROS community. To get updates on when the next synchronization (sync) is coming, subscribe to the Packaging and Release Management Category on ROS Discourse.

**Release Team / Repository**

<table>
<thead>
<tr>
<th>Table of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>• What is ROS 2 GBP?</td>
</tr>
<tr>
<td>• What is a release team?</td>
</tr>
<tr>
<td>– Join a release team</td>
</tr>
<tr>
<td>– Start a new release team</td>
</tr>
</tbody>
</table>
This page explains the recommended method of hosting your release repositories on ros2-gbp.

**What is ROS 2 GBP?**

ros2-gbp is a GitHub organization that hosts the release repositories for ROS packages. It also maintains a list of release teams, the list of members per release team and the list of release repositories maintained by the release teams in https://github.com/ros2-gbp/ros2-gbp-github-org. Interactions with ros2-gbp-github-org are done through raising GitHub issues. It is recommended that you request to join a release team and set up a release repository early as it can take some time for the ros2-gbp maintainers to respond to your requests.

**What is a release team?**

A release team is a GitHub team that consists of a group of people who are responsible for the release process of one or more repositories. Release teams are often made up of an organization, a working group, or even an individual, and are named after the team or group that they represent. The list of release teams and their associated release repositories are maintained at ros2-gbp-github-org.

You must be a part of the release team that you are planning on releasing the project for. If you intend to release the repository under an existing team, follow Join a release team. If you intend to start a new team, follow Start a new release team.

**Join a release team**

Fill the Update Release Team Membership issue issue template if a release team already exists for your project but you are not part of it.

**Start a new release team**

Fill the New Release Team issue issue template if no release team exists for your project yet, request for a new release team to be created.

**What is a release repository?**

A release repository is a repository that

- stores files generated from the release process, for the ROS buildfarm to use
- caches configurations from the release process to simplify subsequent releases of the repository in the future

Having a release repository separate from your source code repository is a requirement for making a release in ROS 2.
Create a new release repository

If your repository is new to the ROS community, you should first open a pull request on rosr Rosdistro adding a source entry for your repository. The review process for the rosdistro database will ensure your repository and packages conform to the REP 144 package naming conventions and other requirements before release. Once your package name has been approved and merged, fill in the Add New Release Repositories issue issue template if you don’t have a release repo for your project yet.

What if my existing release repo isn’t on ros2-gbp?

Packages released before ros2-gbp existed may have their release repositories hosted elsewhere. It is now strongly recommended for release repositories to live in this dedicated GitHub organization. If you are porting a ROS 1 package to ROS 2 and planning on releasing your packages into ROS 2 for the first time, follow standard procedure to request for a new release repository for your ROS 2 releases. If you have previously released your packages for ROS 2, when raising the Add New Release Repositories issue, specify your current release repository url, and follow standard procedure for the rest.

Note: Release repositories hosted elsewhere are still supported for stable distributions if you are not planning to release the repository into Rolling. Since stable distributions created from Rolling will start with release repositories in the ros2-gbp organization it is recommend that you use the ros2-gbp release repositories for all ROS 2 distributions to avoid fragmenting the release information.

A ros2-gbp release repository may become a hard requirement in the future and maintaining a single release repository for all ROS 2 distributions simplifies the maintenance of releases for both the Rolling distribution maintainers and package maintainers.

Release Track

Table of Contents

- What is a Track?
- Track Configurations
  - Release Repository url
  - Repository Name
  - Upstream Repository URI
  - Upstream VCS Type
  - Version
  - Release Tag
  - Upstream Devel Branch
  - ROS Distro
  - Patches Directory
  - Release Repository Push URL
What is a Track?

Bloom requires the user to enter configuration information when releasing packages for the first time. It is beneficial to store such configurations in the release repository so we don't have to manually enter configurations that won't change for subsequent releases.

Since some of the configurations will differ when releasing the package for different ROS distributions, bloom uses release tracks to store the configurations for releasing per distribution. By convention you should create tracks with the same name as the ROS distro you are releasing for.

All release track configurations are stored in tracks.yaml on the master branch of your release repository.

Track Configurations

Track configurations are explained in more detail along with the prompts from bloom.

Release Repository url

This is the url of your release repository, and should be of form https://github.com/ros2-gbp/my_repo-release.git if your release repository is hosted on ros2-gbp.

No reasonable default release repository url could be determined from previous releases. Release repository url [press enter to abort]:

Paste your release repository URL and press Enter.

Bloom may additionally ask you about initializing the new repository, as following:

Freshly initialized git repository detected.
An initial empty commit is going to be made.
Continue [Y/n]?

Simply press Enter to accept the default of yes.

Repository Name

The repository name is trivial, but it is recommended to set this to the name of your project.

Repository Name:
  upstream
  Default value, leave this as upstream if you are unsure <name>
  Name of the repository (used in the archive name)
[
  'upstream']:

Type the name of your project (eg. my_project) and press Enter.
Upstream Repository URI

The **upstream repository** is the repository where your source code is. This is most likely an https link to your project hosted on a git hosting service such as GitHub or GitLab.

Upstream Repository URI:

```html
<uri>
  Any valid URI. This variable can be templated, for example an svn url can be templated as such: "https://svn.foo.com/foo/tags/foo-:{version}"
  where the :{version} token will be replaced with the version for this release.
  [None]:
</uri>
```

Make sure you use the https address (eg. https://github.com/my_organization/my_repo.git) and not the ssh address.

Upstream VCS Type

This is the **Upstream Repository URI**’s version control system (VCS) type. You must specify the type of vcs your repository is using, from svn, git, hg or tar.

Upstream VCS Type:

```html
svn
  Upstream URI is a svn repository

git
  Upstream URI is a git repository

hg
  Upstream URI is a hg repository

tar
  Upstream URI is a tarball

[git]:
```

Most repositories will be using git, but some legacy repositories might be using hg or svn.

Version

This is the version of the package you are releasing. (eg. 1.0.3)

Version:

```html
{:ask}
  This means that the user will be prompted for the version each release.
  This also means that the upstream devel will be ignored.

{:auto}
  This means the version will be guessed from the devel branch.
  This means that the devel branch must be set, the devel branch must exist, and there must be a valid package.xml in the upstream devel branch.

<version>
  This will be the version used.
  It must be updated for each new upstream version.

[auto]:
```

Setting this to :{auto} (the default, and recommended setup) will automatically determine the version from the devel branch’s package.xml.
Setting this to :{ask} will bring up a prompt asking for the version every time you run a release with bloom.

**Release Tag**

The Release Tag refers to which tag or branch you want to import the code from.

```markdown
<table>
<thead>
<tr>
<th>Release Tag:</th>
</tr>
</thead>
<tbody>
<tr>
<td>:{version}</td>
</tr>
<tr>
<td>This means that the release tag will match the :{version} tag. This can be further templated, for example: &quot;foo-:{version}&quot; or &quot;v:{version}&quot;</td>
</tr>
<tr>
<td>This can describe any vcs reference. For git that means {tag, branch, hash}, for hg that means {tag, branch, hash}, for svn that means a revision number. For tar this value doubles as the sub directory (if the repository is in foo/ of the tar ball, putting foo here will cause the contents of foo/ to be imported to upstream instead of foo itself).</td>
</tr>
<tr>
<td>:{ask}</td>
</tr>
<tr>
<td>This means the user will be prompted for the release tag on each release.</td>
</tr>
<tr>
<td>:{none}</td>
</tr>
<tr>
<td>For svn and tar only you can set the release tag to :{none}, so that it is ignored. For svn this means no revision number is used.</td>
</tr>
</tbody>
</table>

[':{version}']:
```

Setting this to :{version} (the default, and recommended setup) will make the release tag match the version tag.

A less common setup is to set this to a branch name to always pull in that branch at the time of release from the upstream project.

Alternatively, if you want to be prompted to enter a different tag every time you do a release, enter :{ask}. :{ask} is useful if the upstream project has frequent tagged releases and you want to refer to the new tag every time you’re releasing.

**Upstream Devel Branch**

The upstream devel branch is the name of the branch in your upstream repository. If you use separate branches for each ROS distribution, this field would be different for each release track. It is used to determine the version of the package you are releasing when Version is set to :{auto}.

```markdown
<table>
<thead>
<tr>
<th>Upstream Devel Branch:</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;vcs reference&gt;</td>
</tr>
<tr>
<td>Branch in upstream repository on which to search for the version.</td>
</tr>
<tr>
<td>This is used only when version is set to ':{auto}'.</td>
</tr>
</tbody>
</table>

[None]:
```

To release from a branch called iron, enter iron. Leaving this as None would result in the version being determined from the default branch of your repository (this is not recommended).
ROS Distro

This is the distribution you’re planning on releasing the package into.

```plaintext
ROS Distro:
  <ROS distro>
    This can be any valid ROS distro, e.g. indigo, kinetic, lunar, melodic
    ['indigo']:
```

If you plan on releasing into ROS iron, enter iron.

Patches Directory

This is the directory where any additional patches to the releases are.

```plaintext
Patches Directory:
  <path in bloom branch>
    This can be any valid relative path in the bloom branch. The contents
    of this folder will be overlaid onto the upstream branch after each
    import-upstream. Additionally, any package.xml files found in the
    overlay will have the :{version} string replaced with the current
    version being released.
    :{none}
      Use this if you want to disable overlaying of files.
    [None]:
```

Adding additional patches to a release is a rarely used feature. For almost all packages, this should be left as the default None.

Release Repository Push URL

```plaintext
Release Repository Push URL:
  :{none}
    This indicates that the default release url should be used.
  <url>
    (optional) Used when pushing to remote release repositories. This is only
    needed when the release uri which is in the rosdistro file is not writable.
    This is useful, for example, when a releaser would like to use a ssh url
    to push rather than a https:// url.
    [None]:
```

Can be left as the default in most cases.

Releasing a package makes your package available on the public ROS 2 buildfarm. This will:

- Make your package available to be installed via package managers (eg. `apt` on Ubuntu) for all supported Linux platforms in a ROS distribution as described in REP 2000.
- Allow your package to have API documentation automatically generated.
- Make your package part of the ROS Index.
- (Optionally) Allow you to have automatic CI run for pull requests in your repository.

Follow one of the guides below to get your package released:
Using Python Packages with ROS 2

Goal: Explain how to interoperate with other Python packages from the ROS 2 ecosystem.

Contents

• Installing via rosdep
• Installing via a package manager
• Installing via a virtual environment

Note: A cautionary note, if you intended to use pre-packaged binaries (either deb files, or the “fat” binary distributions), the Python interpreter must match what was used to build the original binaries. If you intend to use something like virtualenv or pipenv, make sure to use the system interpreter. If you use something like conda, it is very likely that the interpreter will not match the system interpreter and will be incompatible with ROS 2 binaries.

Installing via rosdep

The fastest way to include third-party python packages is to use their corresponding rosdep keys, if available. rosdep keys can be checked via:

• https://github.com/ros/rosdistro/blob/master/rosdep/base.yaml
• https://github.com/ros/rosdistro/blob/master/rosdep/python.yaml

These rosdep keys can be added to your package.xml file, which indicates to the build system that your package (and dependent packages) depend on those keys. In a new workspace, you can also quickly install all rosdep keys with:

rosdep install -yr ./path/to/your/workspace

If there aren’t currently rosdep keys for the package that you are interested in, it is possible to add them by following the rosdep key contribution guide.

To learn more about the rosdep tool and how it works, consult the rosdep documentation.
Installing via a package manager

If you don’t want to make a rosdep key, but the package is available in your system package manager (eg apt), you can install and use the package that way:

```
sudo apt install python3-serial
```

If the package is available on The Python Package Index (PyPI) and you want to install globally on your system:

```
python3 -m pip install -U pyserial
```

If the package is available on PyPI and you want to install locally to your user:

```
python3 -m pip install -U --user pyserial
```

Installing via a virtual environment

First, create a Colcon workspace:

```
mkdir -p ~/colcon_venv/src
cd ~/colcon_venv/
```

Then setup your virtual environment:

```
# Make a virtual env and activate it
virtualenv -p python3 ./venv
source ./venv/bin/activate
# Make sure that colcon doesn’t try to build the venv
touch ./venv/COLCON_IGNORE
```

Next, install the Python packages that you want in your virtual environment:

```
python3 -m pip install gtsam pyserial... etc
```

Now you can build your workspace and run your python node that depends on packages installed in your virtual environment.

```
# Source Iron and build
source /opt/ros/iron/setup.bash
colcon build
```

Note: If you want to release your package using Bloom, you should add the packages you require to rosdep, see the rosdep key contribution guide.
Porting RQt plugins to Windows

Table of Contents

- RQt Porting examples
- Considerations for Windows 10
  - Troubles with TinyXML version 1
  - Code that uses `__cplusplus` and code that requires pluginlib
  - Locations of build artifacts (before install)
  - Compiler and linker flags

RQt Porting examples

Here is the ROS 2 port of `qt_gui_core`.
Here is the ROS 2 port of `python_qt_binding`.

Considerations for Windows 10

Troubles with TinyXML version 1

I could not successfully use TinyXML. I upgraded to TinyXML-2 where needed. It’s a pretty straight forward change. Checkout this PR for an example of porting to TinyXML-2.

Code that uses `__cplusplus` and code that requires pluginlib

In some places, notably in the ROS 2 port of pluginlib, there is use of the `__cplusplus` flag. Unfortunately on Windows Visual Studio does not set this flag correctly regardless of the C++ standard that is actually being used. See this page for more information.

To set it, you need to add the compile option `/Zc:__cplusplus`.
For example, in CMake you could do something like this:

```cpp
target_compile_options(${PROJECT_NAME} PUBLIC "/Zc:__cplusplus")
```

Locations of build artifacts (before install)

This only came up during when building `qt_gui_cpp`. In that package, a custom command depends on a target library in another part of the package. However, that library isn’t installed until build is complete. Windows builds in a `{configuration}` directory. For example:

On Linux, `qt_gui_cpp.a` would be built in: `<ros2_ws>/build/qt_gui_cpp/src/qt_gui_cpp/

But on Windows `qt_gui_cpp.lib` is built in `<ros2_ws>/build/qt_gui_cpp/src/qt_gui_cpp/Release`
For compatibility across platforms in this situation, use CMake generator expressions. However, when you need a library to link against be sure to use $<TARGET_LINKER_FILE:_target> instead of $<TARGET_FILE:_target>. The latter will find .dll files, which cannot be linked against on Windows. See an example here.

**Compiler and linker flags**

In general when porting to Windows, many packages might make use of additional compiler flags. You can find the Windows compiler flags on Microsoft’s documentation. The C++ compiler is called `cl.exe`.

For linker flags see Microsoft’s documentation. The linker program is called `link.exe`.

However, CMake actually provides many of these options in variables. This StackOverflow page contains a good example of how to see all the CMake variables available in a script.

**Running ROS 2 nodes in Docker [community-contributed]**

**Run two nodes in a single docker container**

Pull the ROS docker image with tag “iron-desktop”.

```bash
docker pull osrf/ros:iron-desktop
```

Run the image in a container in interactive mode.

```bash
$ docker run -it osrf/ros:iron-desktop
root@<container-id>:/#
```

Your best friend is the `ros2` command line help now.

```bash
root@<container-id>:/# ros2 --help
```

E.g. list all installed packages.

```bash
root@<container-id>:/# ros2 pkg list
(you will see a list of packages)
```

E.g. list all executables:

```bash
root@<container-id>:/# ros2 pkg executables
(you will see a list of <package> <executable>)
```

Run a minimal example of 2 C++ nodes (1 topic subscriber `listener`, 1 topic publisher `talker`) from the package `demo_nodes_cpp` in this container:

```bash
ros2 run demo_nodes_cpp listener &
ros2 run demo_nodes_cpp talker
```
Run two nodes in two separate docker containers

Open a terminal. Run the image in a container in interactive mode and launch a topic publisher (executable `talker` from the package `demo_nodes_cpp`) with `ros2 run`:

```
docker run -it --rm osrf/ros:iron-desktop ros2 run demo_nodes_cpp talker
```

Open a second terminal. Run the image in a container in interactive mode and launch a topic subscriber (executable `listener` from the package `demo_nodes_cpp`) with `ros2 run`:

```
docker run -it --rm osrf/ros:iron-desktop ros2 run demo_nodes_cpp listener
```

As an alternative to the command line invocation, you can create a `docker-compose.yml` file (here version 2) with the following (minimal) content:

```
version: '2'
services:
  talker:
    image: osrf/ros:iron-desktop
    command: ros2 run demo_nodes_cpp talker
  listener:
    image: osrf/ros:iron-desktop
    command: ros2 run demo_nodes_cpp listener
    depends_on:
      - talker
```

To run the containers call `docker compose up` in the same directory. You can close the containers with Ctrl+C.

Visualizing ROS 2 data with Foxglove Studio

Foxglove Studio is an open source visualization and debugging tool for your robotics data.

It is available in a variety of ways to make development as convenient as possible – it can be run as a standalone desktop app, accessed via your browser, or even self-hosted on your own domain.

View the source code on GitHub.

Installation

To use the web app, simply open Google Chrome and navigate to studio.foxglove.dev.

To use the desktop app for Linux, macOS, or Windows, download it directly from the Foxglove Studio website.
**Connect to a data source**

On opening Foxglove Studio, you will see a dialog with a list of all possible data sources. To connect to your ROS 2 stack, click “Open connection”, select the “Rosbridge (ROS 1 & 2)” tab, and configure your “WebSocket URL”.

You could also drag-and-drop any local ROS 2 .db3 files directly into the application to load them for playback.

---

**Note:** In order to load custom message definitions in your ROS 2 files, try converting them to the MCAP file format.

Check out the Foxglove Studio docs for more detailed instructions.

**Building layouts with panels**

Panels are modular visualization interfaces that can be configured and arranged into Studio layouts. You can also save your layouts for future use, for your own personal reference or with your larger robotics team.

Find the full list of available panels in the sidebar’s “Add panel” tab.

We’ve highlighted some particularly useful ones below:

**1 3D: Display visualization markers in a 3D scene**

Publish marker messages to add primitive shapes (arrows, spheres, etc.) and more complex visualizations (occupancy grids, point clouds, etc.) to your 3D panel’s scene.

Choose the topics you want to display via the topic picker on the left, and configure each topic’s visualization settings in the “Edit topic settings” menu.

Reference the docs for a full list of supported message types and some useful user interactions.
2 Diagnostics: Filter and sort diagnostics messages

Display the status of seen nodes (i.e. stale, error, warn, or OK) from topics with a `diagnostic_msgs/msg/DiagnosticArray` datatype in a running feed, and display the diagnostics data for a given `diagnostic_name/hardware_id`.

Reference the docs for more details.

3 Image: View camera feed images

Select a `sensor_msgs/msg/Image` or `sensor_msgs/msg/CompressedImage` topic to display.
Reference the docs for more details.

4 Log: View log messages

To view `rcl_interfaces/msg/Log` messages live, use the desktop app to connect to your running ROS stack. To view `rcl_interfaces/msg/Log` messages from a pre-recorded data file, you can drag-and-drop your file into either the web or desktop app.

Next, add a Log panel to your layout. If you’ve connected to your ROS stack correctly, you should now see a list of your log messages, with the ability to filter them by node name or severity level.

Reference the docs for more details.

5 Plot: Plot arbitrary values over time

Plot arbitrary values from your topics’ message paths over playback time.

Specify the topic values you want to plot along the y-axis. For the x-axis, choose between plotting the y-axis value’s timestamp, element index, or another custom topic message path.
6 Raw Messages: View incoming topic messages

Display incoming topic data in an easy-to-read collapsible JSON tree format.

Reference the docs for more details.
7 Teleop: Teleoperate your robot

Teleoperate your physical robot by publishing `geometry_msgs/msg/Twist` messages on a given topic back to your live ROS stack.

Reference the docs for more details.

8 URDF Viewer: View and manipulate your URDF model

To visualize and control your robot model in Foxglove Studio, open the web or desktop application and add a URDF Viewer panel to your layout. Then, drag and drop your URDF file into that panel to visualize your robot model.

Select any topic publishing a `JointState` message to update the visualization based on the published joint states (defaults to `/joint_states`).
Toggle to “Manual joint control” to set joint positions using the provided controls.

Reference the docs for more details.

**Other basic actions**

**1 View your ROS graph**

Using the desktop app, connect to your running ROS stack. Next, add a Topic Graph panel to your layout. If you’ve connected to your ROS stack correctly, you should now see a computational graph of your ROS nodes, topics, and services in that panel. Use the controls on the right side of the panel to select which topics to display or to toggle services.
2 View and edit your ROS params

Using the desktop app, connect to your running ROS stack. Next, add a Parameters panel to your layout. If you’ve connected to your ROS stack correctly, you should now see a live view of your current `rosparams`. You can edit these parameter values to publish `rosparam` updates back to your ROS stack.

3 Publish messages back to your live ROS stack

Using the desktop app, connect to your running ROS stack. Next, add a Publish panel to your layout. Specify the topic you want to publish on to infer its datatype and populate the text field with a JSON message template. Selecting a datatype in the dropdown of common ROS datatypes will also populate the text field with a JSON message template.

Edit the template to customize your message before hitting “Publish”.

```json
{
   "header": {
      "seq": 0,
      "stamp": {
         "sec": 0,
         "nsec": 0
      },
      "frame_id": ""
   },
   "status": [
      {
         "level": 0,
         "name": "",
         "message": "",
         "hardware_id": "",
         "values": [
            {
               "key": "",
               "value": ""
            }
         ]
      }
   ]
}
```
ROS 2 Package Maintainer Guide

Each package in the ROS 2 core has one or more maintainers that are responsible for the general health of the package. This guide gives some information about the responsibilities of a ROS 2 core package maintainer.

Table of Contents

- Reviews
- Continuous Integration
  - PR builds (https://build.ros2.org/view/Rpr)
  - CI builds (https://ci.ros2.org)
- Merging Pull Requests
- Keeping CI green
- Making releases
  - Source release
  - Binary release
- Backporting to released distributions
- Responding to issues
- Getting help

Reviews

All incoming code to ROS 2 core repositories must be reviewed. The review is looking for:

- Suitability in the package
- Correct code
- Conforms to developer guidelines:
  - Developer Guide
  - Code Style Guide
- Adds tests for the bug/feature
- Adds documentation for new features
- Clean Continuous Integration run
- Targets default branch (usually “rolling”)
- Has at least one approval from a maintainer that is not the author
Continuous Integration

All incoming code to ROS 2 core repositories must be run through Continuous Integration. ROS 2 currently has two separate CI systems, and it is required that PRs pass both of them before merging.

PR builds (https://build.ros2.org/view/Rpr)

ROS 2 PR (Pull Request) builds run automatically every time a pull request is opened. These builds run a build and test of this package, and this package only. This means that it does not build any dependencies, and it also does not build any packages that depend upon this package. These builds are good for quick feedback to see if the change passes linters, unit tests, etc. There are two major problems with them:

- These builds do not work across multiple repositories (so won’t work for adding or changing an API, etc)
- These tests only run on Linux (they won’t run on macOS or Windows)

To address these two problems, there is also the CI builds.

CI builds (https://ci.ros2.org)

CI builds do not run automatically when a pull request is opened. One of the maintainers of the package must manually request that a CI build is done by going to https://ci.ros2.org/job/ci_launcher/.

By default, running a job in this way will build and run tests for all packages (> 300 currently) on all platforms (Linux, macOS, and Windows). As a full run can take many hours and tie up the CI machines, it is recommended that all runs here restrict the number of packages that are built and tested. This can be accomplished by using the colcon arguments --packages-up-to, --packages-select, --packages-above-and-dependencies, --packages-above, amongst others. See the colcon documentation for more examples on the flags that can be used. Further documentation on how to use the CI machinery is available at https://github.com/ros2/ci/blob/master/CI_BUILDERS.md.

Merging Pull Requests

A pull request can be merged if all of the following are true:

- The DCO bot reports a passing result
- The PR build reports a passing result
- The CI build reports a passing result on all platforms
- The code has been reviewed and approved by at least one maintainer

After a PR is merged, it will automatically get built with the next nightlies. It is highly recommended to check the nightlies after merging pull requests to ensure no regressions have occurred.
Keeping CI green

The nightly jobs that run tests are typically much more comprehensive than what is done for individual pull requests. For this reason, there can be regressions that occur in the nightlies that were not seen in the CI jobs. It is a package maintainers responsibility to check for regressions in their packages at the following locations:

- https://ci.ros2.org/view/nightly
- https://ci.ros2.org/view/packaging
- https://build.ros2.org/view/Rci
- https://build.ros2.org/view/Rdev

For any problems that are found, new issues and/or pull requests on the relevant repositories should be opened.

Making releases

In order to get new features and bugfixes out to end users, the package maintainers must periodically do a release of the package (a release may also be requested on-demand from other maintainers).

As outlined in the developer guide, ROS 2 packages follow semver for version numbers.

A release in ROS terms consists of two distinct steps: making a source release, and then making a binary release.

Source release

A source release creates a changelog and a tag in the relevant repository.

The process starts by generating or updating CHANGELOG.rst files with the following command:

```bash
$ catkin_generate_changelog
```

If one or more packages in the repository don’t have contain CHANGELOG.rst, add the --all option to populate all of the previous commits for each package. The `catkin_generate_changelog` command will simply populate the files with the commit logs from the repository. Since those commit logs aren't always appropriate for a changelog, it is recommended to edit CHANGELOG.rst and edit it to make it more readable. Once editing is done, it is important to commit the updated CHANGELOG.rst file to the repository.

The next step is to bump the version in the package.xml and the changelog files with the following command:

```bash
$ catkin_prepare_release
```

This command will find all of the packages in the repository, check that the changelogs exist, check that there are no uncommitted local changes, increment the version in the package.xml files, and commit/tag the changes with a bloom-compatible tag. Using this command is the best way to ensure the release versions are consistent and compatible with bloom. By default, `catkin_prepare_release` will bump the patch version of the packages, e.g. 0.1.1 -> 0.1.2. However, it can also bump the minor or major number, or even have an exact version set. See the help output from `catkin_prepare_release` for more information.

Assuming the above was successful, a source release has been made.
### Binary release

The next step is to use the `bloom-release` command to create a binary release. For full instructions on how to use bloom, please see [http://wiki.ros.org/bloom](http://wiki.ros.org/bloom). To do a binary release of a package, run:

```
$ bloom-release --track <rosdistro> --rosdistro <rosdistro> <repository_name>
```

For instance, to release the `rclcpp` repository to the Iron distribution, the command would be:

```
$ bloom-release --track iron --rosdistro iron rclcpp
```

This command will fetch the release repository, make the necessary changes to make the release, push the changes to the release repository, and finally open a pull request to [https://github.com/ros/rosdistro](https://github.com/ros/rosdistro).

### Backporting to released distributions

All incoming changes should first land on the development branch. Once a change has been merged onto the development branch, it can be considered for backporting to released distributions. However, any backported code must not break API or ABI in a released distribution. If a change can be backported without breaking API or ABI, then a new pull request targeting the appropriate branch should be created. The new pull request should be added to the appropriate distributions project board at [https://github.com/orgs/ros2/projects](https://github.com/orgs/ros2/projects). The new pull request should have all of the steps run as before, but making sure to target the distribution in question for CI, etc.

### Responding to issues

Package maintainers should also look at incoming issues on the repository and triage the problems that users are having. For issues that look like questions, the issue should be closed and the user redirected to [https://answers.ros.org](https://answers.ros.org).

If an issue looks like a problem, but is not relevant to this particular repository, it should be moved to the appropriate repository with the GitHub “Transfer issue” button.

If the reporter has not provided enough information to determine the cause of the problem, more information should be requested from the reporter.

If this is a new feature, tag the issue with “help-wanted”.

Any remaining issues should be reproduced, and determined if they are truly a bug. If it is a bug, fixes are highly appreciated.

### Getting help

While doing maintenance on a package, questions about general procedures or individual issues may come up. For general questions, please follow the [contributing guidelines](https://github.com/ros-website/website/blob/master/PUBLISHING.md).

For questions on individual issues, please tag the ROS 2 GitHub team (@ros/team), and someone on the team will take a look.
Building a custom Debian package

Many Ubuntu users install ROS 2 on their system by installing *debian packages*. This guide gives a short set of instructions to build local, custom Debian packages.

### Table of Contents

- Prerequisites
- Install dependencies
- Initialize rosdep
- Build the debian from the package

### Prerequisites

To successfully build a custom package, all of the dependencies of the package to be built must be available locally or in rosdep. Additionally, all of the dependencies of the package should be properly declared in the `package.xml` file of the package.

### Install dependencies

Run the following command to install utilities needed for the build:

```bash
$ sudo apt install python3-bloom python3-rosdep fakeroot debhelper dh-python
```

### Initialize rosdep

Initialize the rosdep database by calling:

```bash
$ sudo rosdep init
$ rosdep update
```

Note that the `rosdep init` command may fail if it has already been initialized in the past; this can safely be ignored.

### Build the debian from the package

Run the following commands to build the debian:

```bash
$ cd /path/to/pkg_source  # this should be the directory that contains the package.xml
$ bloom-generate rosdebian
$ fakeroot debian/rules binary
```

Assuming that all required dependencies are available and that compilation succeeds, the new package will be available in the parent directory of this directory.
Building ROS 2 with tracing

Tracing instrumentation is included in the ROS 2 source code, and Linux installations of ROS 2 include the LTTng tracer as a dependency. Therefore, ROS 2 can be traced out-of-the-box on Linux.

However, ROS 2 can be built from source to remove the tracepoints or completely remove the instrumentation. This guide shows how to do that. For more information, see the repository.

**Note:** This guide only applies to Linux systems.

### Prerequisites

Set up your system to build ROS 2 from source. See the source installation page for more information.

### Build configurations

The ROS 2 tracing instrumentation is split into two components: function instrumentation and tracepoints. First, a ROS 2 core package (e.g., rclcpp) calls a function provided by the tracetools package. Then, that function triggers a tracepoint, which records data if the tracepoint is enabled at runtime.

By default, if the tracer is not configured to trace or if the tracepoints are not enabled, they will have virtually no impact on the execution. However, the tracepoints can still be removed through a CMake option. Furthermore, the functions can be completely removed through a CMake option, which implies that tracepoints are also removed.

### Building without tracepoints

This step depends on whether you are building ROS 2 from source or using ROS 2 binaries (Debian packages or “fat” archive). To remove the tracepoints, (re)build tracetools and set the TRACETOOLS_TRACEPOINTS_EXCLUDED CMake option to ON:

**Source installation**

```
cd ~/ros2_iron
colcon build --packages-select tracetools --cmake-clean-cache --cmake-args -DTRACETOOLS_TRACEPOINTS_EXCLUDED=ON
```

**Binary installation**

Clone the ros2_tracing repository into your workspace and build:
cd ~/ws
git clone https://github.com/ros2/ros2_tracing.git -b iron src/ros2_tracing
colcon build --packages-select tracetools --cmake-args -DTRACETOOLS_TRACEPOINTS_EXCLUDED=ON

Building without instrumentation

To completely remove both tracepoints and function calls, build ROS2 from source and set the TRACETOOLS_DISABLED CMake option to ON:

```sh
cd ~/ros2_iron
colcon build --cmake-args -DTRACETOOLS_DISABLED=ON --no-warn-unused-cli
```

Validating

Validate that tracing is disabled:

```sh
cd ~/ws
source install/setup.bash
ros2 run tracetools status
```

It should print out:

Without tracepoints

```
Tracing disabled
```

Without instrumentation

```
Tracing disabled through configuration
```

If something else is printed, then something went wrong.

Topics vs Services vs Actions

When designing a system there are three primary styles of interfaces. The specifications for the content is in the Interfaces Overview. This is written to provide the reader with guidelines about when to use each type of interface.
Topics

- Should be used for continuous data streams (sensor data, robot state, ...).
- Are for continuous data flow. Data might be published and subscribed at any time independent of any senders/receivers. Many to many connection. Callbacks receive data once it is available. The publisher decides when data is sent.

Services

- Should be used for remote procedure calls that terminate quickly, e.g. for querying the state of a node or doing a quick calculation such as IK. They should never be used for longer running processes, in particular processes that might be required to preempt if exceptional situations occur and they should never change or depend on state to avoid unwanted side effects for other nodes.
- Simple blocking call. Mostly used for comparably fast tasks as requesting specific data. Semantically for processing requests.

Actions

- Should be used for any discrete behavior that moves a robot or that runs for a longer time but provides feedback during execution.
- The most important property of actions is that they can be preempted and preemption should always be implemented cleanly by action servers.
- Actions can keep state for the lifetime of a goal, i.e. if executing two action goals in parallel on the same server, for each client a separate state instance can be kept since the goal is uniquely identified by its id.
- Slow perception routines which take several seconds to terminate or initiating a lower-level control mode are good use cases for actions.

Using variants

Metapackages do not provide software directly but depend on a group of other related packages to provide a convenient installation mechanism for the complete group of packages. Variants are a list of official metapackages for commonly useful groups of ROS packages.

The different variants in ROS 2 are specified in REP-2001.

In addition to the official variants, there may be metapackages for specific institutions or robots as described in REP-108.

---

1 https://wiki.debian.org/metapackage
2 https://help.ubuntu.com/community/MetaPackages
**Adding variants**

Additional variants that are of general use to the ROS community can be proposed by contributing an update to REP-2001 via pull request describing the packages included in the new variant. Institution and robot specific variants can be published directly by their respective maintainers and no update to REP-2001 is required.

**Creating project-specific variants**

If you are creating ROS packages to use privately in your own projects, you can create variants specific to your projects using the official variants as examples. To do so you need only create two files:

1. A minimal variant package is created as a package with the `ament_cmake` build type, a `buildtool_depend` on `ament_cmake` and `exec_depend` entries for each package you want to include in the variant.

```xml
<?xml version="1.0"?>
<package format="2">
  <name>my_project_variant</name>
  <version>1.0.0</version>
  <description>A package to aggregate all packages in my_project.</description>
  <maintainer email="maintainer-email">Maintainer Name</maintainer>
  <license>Apache License 2.0</license>
  <!-- packages in my_project -->
  <exec_depend>my_project_msgs</exec_depend>
  <exec_depend>my_project_services</exec_depend>
  <exec_depend>my_project_examples</exec_depend>

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

1. A minimal `ament_cmake` package includes a `CMakeLists.txt` which registers the package.xml as an ament package for use in ROS 2.

```cmake
cmake_minimum_required(VERSION 3.5)
project(my_project_variant NONE)
find_package(ament_cmake REQUIRED)
ament_package()
```

1. You can then build and install your variant package alongside your other private packages.
Creating custom variants with platform-specific tools

Some platforms have tools for creating basic packages that do not require a full ROS build farm environment or equivalent infrastructure. It is possible to use these tools to create platform-dependent variants. This approach does not include support for ROS packaging tools and is platform dependent but requires much less infrastructure to produce if you are creating collections of existing packages rather than a mix of public and private ROS packages. For example, on Debian or Ubuntu systems you can use the `equivs` utilities. The Debian Administrator's handbook has a section on meta-packages.

Using the `ros2 param` command-line tool

<table>
<thead>
<tr>
<th>Table of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>• <code>ros2 param list</code></td>
</tr>
<tr>
<td>• <code>ros2 param get</code></td>
</tr>
<tr>
<td>• <code>ros2 param set</code></td>
</tr>
<tr>
<td>• <code>ros2 param delete</code></td>
</tr>
<tr>
<td>• <code>ros2 param describe</code></td>
</tr>
<tr>
<td>• <code>ros2 param dump</code></td>
</tr>
<tr>
<td>• <code>ros2 param load</code></td>
</tr>
</tbody>
</table>

Parameters in ROS 2 can be get, set, listed, and described through a set of services as described in the concept document. The `ros2 param` command-line tool is a wrapper around these service calls that makes it easy to manipulate parameters from the command-line.

**ros2 param list**

This command will list all of the available parameters on a given node, or on all discoverable nodes if no node is given. To get all of the parameters on a given node:

```
ros2 param list /my_node
```

To get all of the parameters on all nodes in the system (this can take a long time on a complicated network):

```
ros2 param list
```

**ros2 param get**

This command will get the value of a particular parameter on a particular node. To get the value of a parameter on a node:

```
ros2 param get /my_node use_sim_time
```
**ros2 param set**

This command will set the value of a particular parameter on a particular node. For most parameters, the type of the new value must be the same as the existing type.

To set the value of a parameter on a node:

```
ros2 param set /my_node use_sim_time false
```

The value that is passed on the command-line is in YAML, which allows arbitrary YAML expressions to be used. However, it also means that certain expressions will be interpreted differently than might be expected. For instance, if the parameter `my_string` on node `my_node` is of type string, the following will not work:

```
ros2 param set /my_node my_string off
```

That’s because YAML is interpreting “off” as a boolean, and `my_string` is a string type. This can be worked around by using the YAML syntax for explicitly setting strings, e.g.:

```
ros param set /my_node my_string '!!str off'
```

Additionally, YAML supports heterogeneous lists, containing (say) a string, a boolean, and an integer. However, ROS 2 parameters do not support heterogeneous lists, so any YAML list that has multiple types will be interpreted as a string. Assuming that the parameter `my_int_array` on node `my_node` is of type integer array, the following will not work:

```
ros param set /my_node my_int_array '[foo,off,1]'
```

The following string typed parameter would work:

```
ros param set /my_node my_string '[foo,off,1]'
```

**ros2 param delete**

This command will remove a parameter from a particular node. However, note that this can only remove dynamic parameters (not declared parameters). See the concept document for more information.

```
ros2 param delete /my_node my_string
```

**ros2 param describe**

This command will provide a textual description of a particular parameter on a particular node:

```
ros2 param describe /my_node use_sim_time
```
**Vulcanexux Documentation, Release 1.0.0**

**ros2 param dump**

This command will print out all of the parameters on a particular node in a YAML file format. The output of this command can then be used to re-run the node with the same parameters later:

```bash
ros2 param dump /my_node
```

**ros2 param load**

This command will load the values of the parameters from a YAML file into a particular node. That is, this command can reload values at runtime that were dumped out by `ros2 param dump`:

```bash
ros2 param load /my_node my_node.yaml
```

**Using ros1_bridge with upstream ROS on Ubuntu 22.04**

The release of ROS 2 Humble (and Rolling) on Ubuntu 22.04 Jammy Jellyfish marks the first ROS 2 release on a platform with no official ROS 1 release. While ROS 1 Noetic will continue to be supported through the duration of its long term support window, it will only target Ubuntu 20.04. Alternatively, there are upstream variants of ROS 1 packages in Debian and Ubuntu that are not maintained as an official distribution by the ROS maintainers.

This guide outlines the current mechanism for bridging ROS 2 releases with these upstream packages on Ubuntu 22.04 Jammy Jellyfish. This provides a migration path for users who still depend on ROS 1, but desire moving to newer ROS 2 and Ubuntu releases.

**ROS 2 via Debian packages**

Installing *ROS 2 from Debian packages* currently does not work for ROS 2 on Ubuntu Jammy. The version of `catkin-pkg-modules` available in the Ubuntu repository conflicts with that in the ROS 2 package repository.

If the ROS 2 apt repository is in the available apt repositories (`/etc/apt/sources.list.d`), no ROS 1 packages will be installable. The error will be:

```bash
$ apt install ros-core-dev
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
Some packages could not be installed. This may mean that you have requested an impossible situation or if you are using the unstable distribution that some required packages have not yet been created or been moved out of Incoming.
The following information may help to resolve the situation:

The following packages have unmet dependencies:
```

(continues on next page)
ros-core-dev : Depends: catkin but it is not installable
E: Unable to correct problems, you have held broken packages.

To correct this, remove packages.ros.org from your sources.list. If you were following the ROS 2 installation guide, simply remove /etc/apt/sources.list.d/ros2.list

For now, to support ros1_bridge, follow the instructions below for building ROS 2 from source.

**ROS 2 from source**

Installing *ROS 2 from Source* is the only configuration that works for ROS 2 on Ubuntu Jammy.

Below is a summary of the necessary instructions from the source build instructions. The substantial deviation is that we skip using the ROS 2 apt repositories because of conflicting packages.

**Install development tools and ROS tools**

Since we aren’t using the ROS 2 apt repositories, colcon must be installed via pip.

```
sudo apt update && sudo apt install -y \
  build-essential \
  cmake \
  git \
  python3-flake8 \
  python3-flake8-blind-except \
  python3-flake8-bultins \
  python3-flake8-class-newline \
  python3-flake8-comprehensions \
  python3-flake8-deprecated \
  python3-flake8-docstrings \
  python3-flake8-import-order \
  python3-flake8-quotes \
  python3-pip \
  python3-pytest \
  python3-pytest-cov \
  python3-pytest-repeat \
  python3-pytest-rerunfailures \
  python3-rosdep \
  python3-setuptools \
  wget

# Install colcon from PyPI, rather than apt packages
python3 -m pip install -U colcon-common-extensions vcstool
```

From here, continue with the *source install guide* to build ROS 2.
Install ROS 1 from Ubuntu packages

```
sudo apt update && sudo apt install -y ros-core-dev
```

Build ros1_bridge

```
# Create a workspace for the ros1_bridge
mkdir -p ~/ros1_bridge/src
cd ~/ros1_bridge/src
git clone https://github.com/ros2/ros1_bridge

cd ~/ros1_bridge

# Source the ROS 2 workspace
. ~/ros2_humble/install/local_setup.bash

# Build
colcon build
```

After building all of `ros1_bridge`, the remainder of the `ros1_bridge` examples should work with your new installation.

Disabling Zero Copy Loaned Messages

Contents

- How to disable Loaned Messages

See the Loaned Messages article for details on how loaned messages work.

How to disable Loaned Messages

By default, Loaned Messages will try to borrow the memory from underlying middleware if it supports Loaned Messages. The `ROS_DISABLE_LOANED_MESSAGES` environment variable can be used to disable Loaned Messages, and fallback to normal publisher and subscription behavior, without any code changes or middleware configuration. You can set the environment variable with the following command:

Linux

```bash
export ROS_DISABLE_LOANED_MESSAGES=1
```

To maintain this setting between shell sessions, you can add the command to your shell startup script:

```bash
echo "export ROS_DISABLE_LOANED_MESSAGES=1" >> ~/.bashrc
```

macOS

```bash
export ROS_DISABLE_LOANED_MESSAGES=1
```

To maintain this setting between shell sessions, you can add the command to your shell startup script:
echo "export ROS_DISABLE_LOANED_MESSAGES=1" >> ~/.bash_profile

Windows

set ROS_DISABLE_LOANED_MESSAGES=1

If you want to make this permanent between shell sessions, also run:

setx ROS_DISABLE_LOANED_MESSAGES 1

**ROS 2 on Raspberry Pi**

ROS 2 is supported on both 32 bit (arm32) and 64 bit (arm64) ARM processors. However, you can see here that arm64 receives Tier 1 support, while arm32 is Tier 3. Tier 1 support means distribution specific packages and binary archives are available, while Tier 3 requires the user to compile ROS 2 from source.

The fastest and simplest way to use ROS 2 is to use a Tier 1 supported configuration.

This would mean either installing 64 bit Ubuntu on to the Raspberry Pi, or using the 64 bit version of Raspberry Pi OS and running ROS 2 in Docker.

**Ubuntu Linux on Raspberry Pi with binary ROS 2 install**

Ubuntu for Raspberry Pi is available [here](#).

Make sure to confirm that you have selected the correct version as described in REP-2000.

You can now install ROS 2 using the normal binary installation instructions for Ubuntu Linux.

**Raspberry Pi OS with ROS 2 in docker**

Raspberry Pi OS 64 bit version is available [here](#).

Raspberry Pi OS is based on Debian which receives Tier 3 support, but it can run Ubuntu docker containers for Tier 1 support.

After flashing the OS, install Docker.

The official OSRF ROS 2 Docker container definitions can be found [here](#).

You may choose from ros-core, ros-base, or ros-desktop. See [here](#) for more information on these variants.

Clone the docker_images git repo onto the Raspberry Pi, change in to the directory linked above, then to the directory with your preferred variant.

Inside of the directory, build the container with:

docker build -t ros_docker .

On a supported system it will only take a minute or two to build the docker containers, as the source code is already built in to binaries.
Pre-built Docker container

A pre-built container for the desktop variant is available as well, which only requires a docker pull command. See this page for more information.

Using Callback Groups

When running a node in a Multi-Threaded Executor, ROS 2 offers callback groups as a tool for controlling the execution of different callbacks. This page is meant as a guide on how to use callback groups efficiently. It is assumed that the reader has a basic understanding about the concept of executors.

Basics of callback groups

When running a node in a Multi-Threaded Executor, ROS 2 offers two different types of callback groups for controlling execution of callbacks:

- Mutually Exclusive Callback Group
- Reentrant Callback Group

These callback groups restrict the execution of their callbacks in different ways. In short:

- Mutually Exclusive Callback Group prevents its callbacks from being executed in parallel - essentially making it as if the callbacks in the group were executed by a SingleThreadedExecutor.

- Reentrant Callback Group allows the executor to schedule and execute the group’s callbacks in any way it sees fit, without restrictions. This means that, in addition to different callbacks being run parallel to each other, different instances of the same callback may also be executed concurrently.

- Callbacks belonging to different callback groups (of any type) can always be executed parallel to each other.

It is also important to keep in mind that different ROS 2 entities relay their callback group to all callbacks they spawn. For example, if one assigns a callback group to an action client, all callbacks created by the client will be assigned to that callback group.

Callback groups can be created by a node’s create_callback_group function in rclcpp and by calling the constructor of the group in rclpy. The callback group can then be passed as argument/option when creating a subscription, timer, etc.

C++
my_callback_group = create_callback_group(rclcpp::CallbackGroupType::MutuallyExclusive);

rclcpp::SubscriptionOptions options;
options.callback_group = my_callback_group;

my_subscription = create_subscription<Int32>("/topic", rclcpp::SensorDataQoS(),
callback, options);

Python

my_callback_group = MutuallyExclusiveCallbackGroup()
my_subscription = self.create_subscription(Int32, "/topic", self.callback, qos_profile=1,
callback_group=my_callback_group)

If the user does not specify any callback group when creating a subscription, timer, etc., this entity will be assigned to the node’s default callback group. The default callback group is a Mutually Exclusive Callback Group and it can be queried via NodeBaseInterface::get_default_callback_group() in rclcpp and via Node.
default_callback_group in rclpy.

About callbacks

In the context of ROS 2 and executors, a callback means a function whose scheduling and execution is handled by an executor. Examples of callbacks in this context are

• subscription callbacks (receiving and handling data from a topic),
• timer callbacks,
• service callbacks (for executing service requests in a server),
• different callbacks in action servers and clients,
• done-callbacks of Futures.

Below are a couple important points about callbacks that should be kept in mind when working with callback groups.

• Almost everything in ROS 2 is a callback! Every function that is run by an executor is, by definition, a callback. The non-callback functions in a ROS 2 system are found mainly at the edge of the system (user and sensor inputs etc).

• Sometimes the callbacks are hidden and their presence may not be obvious from the user/developer API. This is the case especially with any kind of “synchronous” call to a service or an action (in rclpy). For example, the synchronous call Client.call(request) to a service adds a Future’s done-callback that needs to be executed during the execution of the function call, but this callback is not directly visible to the user.

Controlling execution

In order to control execution with callback groups, one can consider the following guidelines.

For the interaction of an individual callback with itself:

• Register it to a Reentrant Callback Group if it should be executed in parallel to itself. An example case could be an action/service server that needs to be able to process several action calls in parallel to each other.

• Register it to a Mutually Exclusive Callback Group if it should never be executed in parallel to itself. An example case could be a timer callback that runs a control loop that publishes control commands.

For the interaction of different callbacks with each other:
- Register them to the same Mutually Exclusive Callback Group if they should never be executed in parallel. An example case could be that the callbacks are accessing shared critical and non-thread-safe resources.

If they should be executed in parallel, you have two options, depending on whether the individual callbacks should be able to overlap themselves or not:

- Register them to different Mutually Exclusive Callback Groups (no overlap of the individual callbacks)
- Register them to a Reentrant Callback Group (overlap of the individual callbacks)

An example case of running different callbacks in parallel is a Node that has a synchronous service client and a timer calling this service. See the detailed example below.

**Avoiding deadlocks**

Setting up callback groups of a node incorrectly can lead to deadlocks (or other unwanted behavior), especially if one desires to use synchronous calls to services or actions. Indeed, even the API documentation of ROS 2 mentions that synchronous calls to actions or services should not be done in callbacks, because it can lead to deadlocks. While using asynchronous calls is indeed safer in this regard, synchronous calls can also be made to work. On the other hand, synchronous calls also have their advantages, such as making the code simpler and easier to understand. Hence, this section provides some guidelines on how to set up a node’s callback groups correctly in order to avoid deadlocks.

First thing to note here is that every node’s default callback group is a Mutually Exclusive Callback Group. If the user does not specify any other callback group when creating a timer, subscription, client etc., any callbacks created then or later by these entities will use the node’s default callback group. Furthermore, if everything in a node uses the same Mutually Exclusive Callback Group, that node essentially acts as if it was handled by a Single-Threaded Executor, even if a multi-threaded one is specified! Thus, whenever one decides to use a Multi-Threaded Executor, some callback group(s) should always be specified in order for the executor choice to make sense.

With the above in mind, here are a couple guidelines to help avoid deadlocks:

- If you make a synchronous call in any type of a callback, this callback and the client making the call need to belong to
  - different callback groups (of any type), or
  - a Reentrant Callback Group.

- If the above configuration is not possible due to other requirements - such as thread-safety and/or blocking of other callbacks while waiting for the result (or if you want to make absolutely sure that there is never a possibility of a deadlock), use asynchronous calls.

Failing the first point will always cause a deadlock. An example of such a case would be making a synchronous service call in a timer callback (see the next section for an example).

**Examples**

Let us look at some simple examples of different callback group setups. The following demo code considers calling a service synchronously in a timer callback.
Demo code

We have two nodes - one providing a simple service:

C++

```cpp
#include <memory>
#include "rclcpp/rclcpp.hpp"
#include "std_srvs/srv/empty.hpp"

using namespace std::placeholders;

namespace cb_group_demo {
    class ServiceNode : public rclcpp::Node {
    public:
        ServiceNode() : Node("service_node") {
            service_ptr_ = this->create_service<std_srvs::srv::Empty>(
                "test_service",
                std::bind(&ServiceNode::service_callback, this, _1, _2, _3)
            );
        }
    private:
        rclcpp::Service<std_srvs::srv::Empty>::SharedPtr service_ptr_;
        void service_callback(
            const std::shared_ptr<rmw_request_id_t> request_header,
            const std::shared_ptr<std_srvs::srv::Empty::Request> request,
            const std::shared_ptr<std_srvs::srv::Empty::Response> response)
            {
                (void)request_header;
                (void)request;
                (void)response;
                RCLCPP_INFO(this->get_logger(), "Received request, responding...");
            }
    };
}

int main(int argc, char* argv[]) {
    rclcpp::init(argc, argv);
    auto service_node = std::make_shared<cb_group_demo::ServiceNode>();
    RCLCPP_INFO(service_node->get_logger(), "Starting server node, shut down with CTRL-C \n");
    rclcpp::spin(service_node);
    RCLCPP_INFO(service_node->get_logger(), "Keyboard interrupt, shutting down.\n");
    rclcpp::shutdown();
    return 0;
}
Python

```python
import rclpy
from rclpy.node import Node
from std_srvs.srv import Empty

class ServiceNode(Node):
    def __init__(self):
        super().__init__('service_node')
        self.srv = self.create_service(Empty, 'test_service', callback=self.service_callback)

    def service_callback(self, request, result):
        self.get_logger().info('Received request, responding...')
        return result

if __name__ == '__main__':
    rclpy.init()
    node = ServiceNode()
    try:
        node.get_logger().info("Starting server node, shut down with CTRL-C")
        rclpy.spin(node)
    except KeyboardInterrupt:
        node.get_logger().info('Keyboard interrupt, shutting down.
        node.destroy_node()
        rclpy.shutdown()
```

and another containing a client to the service along with a timer for making service calls:

C++

Note: The API of service client in rclcpp does not offer a synchronous call method similar to the one in rclpy, so we wait on the future object to simulate the effect of a synchronous call.

```cpp
#include <chrono>
#include <memory>
#include "rclcpp/rclcpp.hpp"
#include "std_srvs/srv/empty.hpp"

using namespace std::chrono_literals;

namespace cb_group_demo {
    class DemoNode : public rclcpp::Node {
    public:
        DemoNode() : Node("client_node")
        {
            client_cb_group_ = nullptr;
            timer_cb_group_ = nullptr;
            client_ptr_ = this->create_client<std_srvs::srv::Empty>("test_service", rmw_qos_profile_services_default, client_cb_group_);
            timer_ptr_ = this->create_wall_timer(1s, std::bind(&DemoNode::timer_callback, this),
```
```cpp
private:
    rclcpp::CallbackGroup::SharedPtr client_cb_group_;  
    rclcpp::CallbackGroup::SharedPtr timer_cb_group_;  
    rclcpp::Client<std_srvs::srv::Empty>::SharedPtr client_ptr_;  
    rclcpp::TimerBase::SharedPtr timer_ptr_;  

    void timer_callback()  
    {  
        RCLCPP_INFO(this->get_logger(), "Sending request");  
        auto request = std::make_shared<std_srvs::srv::Empty::Request>();  
        auto result_future = client_ptr_->async_send_request(request);  
        std::future_status status = result_future.wait_for(10s); // timeout to guarantee a graceful finish  
        if (status == std::future_status::ready) {  
            RCLCPP_INFO(this->get_logger(), "Received response");  
        }  
    }  
}; // class DemoNode  
} // namespace cb_group_demo
```

```python
import rclpy
from rclpy.executors import MultiThreadedExecutor  
from rclpy.callback_groups import MutuallyExclusiveCallbackGroup, ReentrantCallbackGroup  
from rclpy.node import Node  
from std_srvs.srv import Empty

class CallbackGroupDemo(Node):
    def __init__(self):
        super().__init__('client_node')
```

(continues on next page)
The client node's constructor contains options for setting the callback groups of the service client and the timer. With the default setting above (both being `nullptr / None`), both the timer and the client will use the node's default Mutually Exclusive Callback Group.

The problem

Since we are making service calls with a 1 second timer, the expected outcome is that the service gets called once a second, the client always gets a response and prints Received response. If we try running the server and client nodes in terminals, we get the following outputs.

Client

```
[INFO] [1653034371.758739131] [client_node]: Starting client node, shut down with CTRL-C
[INFO] [1653034372.755865649] [client_node]: Sending request
^C[INFO] [1653034398.161674869] [client_node]: Keyboard interrupt, shutting down.
```

Server

```
[INFO] [1653034355.308958238] [service_node]: Starting server node, shut down with CTRL-C
[INFO] [1653034372.758197320] [service_node]: Received request, responding...
^C[INFO] [1653034416.021962246] [service_node]: Keyboard interrupt, shutting down.
```

So, it turns out that instead of the service being called repeatedly, the response of the first call is never received, after which the client node seemingly gets stuck and does not make further calls. That is, the execution stopped at a deadlock!

The reason for this is that the timer callback and the client are using the same Mutually Exclusive Callback Group (the
When the service call is made, the client then passes its callback group to the Future object (hidden inside the call-method in the Python version) whose done-callback needs to execute for the result of the service call to be available. But because this done-callback and the timer callback are in the same Mutually Exclusive group and the timer callback is still executing (waiting for the result of the service call), the done-callback never gets to execute. The stuck timer callback also blocks any other executions of itself, so the timer does not fire for a second time.

**Solution**

We can fix this easily - for example - by assigning the timer and client to different callback groups. Thus, let us change the first two lines of the client node’s constructor to be as follows (everything else shall stay the same):

C++

```cpp
client_cb_group_ = this->create_callback_group(rclcpp::CallbackGroupType::MutuallyExclusive);
timer_cb_group_ = this->create_callback_group(rclcpp::CallbackGroupType::MutuallyExclusive);
```

Python

```python
client_cb_group = MutuallyExclusiveCallbackGroup()
timer_cb_group = MutuallyExclusiveCallbackGroup()
```

Now we get the expected result, i.e. the timer fires repeatedly and each service call gets the result as it should:

**Client**

```
[INFO] [1653067523.431731177] [client_node]: Starting client node, shut down with CTRL-C
[INFO] [1653067524.431912821] [client_node]: Sending request
[INFO] [1653067524.433230445] [client_node]: Received response
[INFO] [1653067525.431869330] [client_node]: Sending request
[INFO] [1653067525.432912803] [client_node]: Received response
[INFO] [1653067526.431844726] [client_node]: Sending request
[INFO] [1653067526.432893954] [client_node]: Received response
[INFO] [1653067527.431828287] [client_node]: Sending request
[INFO] [1653067527.432848369] [client_node]: Received response
^C[INFO] [1653067528.400052749] [client_node]: Keyboard interrupt, shutting down.
```

**Server**

```
[INFO] [1653067522.052866001] [service_node]: Starting server node, shut down with CTRL-C
[INFO] [1653067524.432577720] [service_node]: Received request, responding...
[INFO] [1653067525.43265009] [service_node]: Received request, responding...
[INFO] [1653067526.432300261] [service_node]: Received request, responding...
[INFO] [1653067527.432272441] [service_node]: Received request, responding...
^C[INFO] [1653034416.021962246] [service_node]: KeyboardInterrupt, shutting down.
```

One might consider if just avoiding the node’s default callback group is enough. This is not the case: replacing the default group by a different Mutually Exclusive group changes nothing. Thus, the following configuration also leads to the previously discovered deadlock.

C++
client_cb_group_ = this->create_callback_group(rclcpp::CallbackGroupType::MutuallyExclusive);
timer_cb_group_ = client_cb_group_;

Python

client_cb_group = MutuallyExclusiveCallbackGroup()
timer_cb_group = client_cb_group

In fact, the exact condition with which everything works in this case is that the timer and client must not belong to
the same Mutually Exclusive group. Hence, all of the following configurations (and some others as well) produce the
desired outcome where the timer fires repeatedly and service calls are completed.

C++

client_cb_group_ = this->create_callback_group(rclcpp::CallbackGroupType::Reentrant);
timer_cb_group_ = client_cb_group_;

or

client_cb_group_ = this->create_callback_group(rclcpp::CallbackGroupType::MutuallyExclusive);
timer_cb_group_ = nullptr;

or

client_cb_group_ = nullptr;
timer_cb_group_ = this->create_callback_group(rclcpp::CallbackGroupType::MutuallyExclusive);

or

client_cb_group_ = this->create_callback_group(rclcpp::CallbackGroupType::Reentrant);
timer_cb_group_ = nullptr;

Python

client_cb_group = ReentrantCallbackGroup()
timer_cb_group = client_cb_group

or

client_cb_group = MutuallyExclusiveCallbackGroup()
timer_cb_group = None

or

client_cb_group = None
timer_cb_group = MutuallyExclusiveCallbackGroup()

or

client_cb_group = ReentrantCallbackGroup()
timer_cb_group = None
Setup ROS 2 with VSCode and Docker [community-contributed]

Contents

- Install VS Code and Docker
  - Install Docker
  - Install VS Code
    - Install Remote Development Extension
- Configure workspace in Docker and VS Code
  - Add your ROS 2 workspace
  - Edit devcontainer.json for your environment
    - Edit Dockerfile
- Open and Build Development Container
  - Test Container

Install VS Code and Docker

Using Visual Studio Code and Docker Containers will enable you to run your favorite ROS 2 Distribution without the necessity to change your operating system or use a virtual machine. With this tutorial you can set up a docker container, which can be used for your future ROS 2 projects.

Install Docker

To install docker and set the correct user rights please use the following commands.

```bash
sudo apt install docker.io git python3-pip
pip3 install vcstool
echo export PATH=$HOME/.local/bin:$PATH >> ~/.bashrc
source ~/.bashrc
sudo groupadd docker
sudo usermod -aG docker $USER
newgrp docker
```

Now you can check if the installation was successful by running the following command:

```bash
docker run hello-world
```

You might need to start the Docker Daemon first, if you cannot run hello-world out of the box:

```bash
sudo systemctl start docker
```
Install VS Code

To install VS Code please use the following commands:

```
sudo apt update
sudo apt install software-properties-common apt-transport-https
wget -q https://packages.microsoft.com/keys/microsoft.asc -O- | sudo apt-key add -
sudo add-apt-repository "deb [arch=amd64] https://packages.microsoft.com/repos/vscode＿
˓stable main"
sudo apt install code
```

You can run VS Code by typing `code` in a terminal.

Install Remote Development Extension

Within VS Code search in Extensions (CTRL+SHIFT+X) for the “Remote Development” Extension and install it.

Configure workspace in Docker and VS Code

Add your ROS 2 workspace

Add a workspace in order to build and open them in a container, e.g.: 

```
cd ~/
mkdir ws_[project]
cd ws_[project]
mkdir src
```

Now create a .devcontainer folder in the root of your workspace and add a devcontainer.json and Dockerfile to this .devcontainer folder. Additionally, you need to create a cache folder in which you can cache the build and install folders for different ROS 2 distros. The workspace structure should look like this:

```
ws_[project]
  ├── cache
  │    └── [ROS2_DISTRO]
  │         ├── build
  │         └── install
  │             └── log
  │                 └── ...
  └── src
  │     ├── .devcontainer
  │     │    └── devcontainer.json
  │     └── Dockerfile
  │         └── package1
  │             └── package2
```

With File→Open Folder... or Ctrl+K Ctrl+O, open the src folder of your workspace in VS Code.
Edit devcontainer.json for your environment

For the Dev Container to function properly, we have to build it with the correct user. Therefore add the following to .devcontainer/devcontainer.json:

```json
{
    "name": "ROS 2 Development Container",
    "privileged": true,
    "remoteUser": "USERNAME",
    "build": {
        "dockerfile": "Dockerfile",
        "args": {
            "USERNAME": "USERNAME"
        }
    },
    "workspaceFolder": "/home/ws",
    "workspaceMount": "source=${localWorkspaceFolder},target=/home/ws/src,type=bind",
    "customizations": {
        "vscode": {
            "extensions": [
                "ms-vscode.cpptools",
                "ms-vscode.cpptools-themes",
                "twxs.cmake",
                "donjayamanne.python-extension-pack",
                "eamodio.gitlens",
                "ms-iot.vscode-ros"
            ]
        }
    },
    "containerEnv": {
        "DISPLAY": "unix:0",
        "ROS_AUTOMATIC_DISCOVERY_RANGE": "LOCALHOST",
        "ROS_DOMAIN_ID": "42"
    },
    "runArgs": [
        "--net=host",
        "-e", "DISPLAY=${env:DISPLAY}"],
    "mounts": [
        "source=/tmp/.X11-unix,target=/tmp/.X11-unix,type=bind,consistency=cached",
        "source=/dev/dri,target=/dev/dri,type=bind,consistency=cached",
        "source=${localWorkspaceFolder}/..cache/ROS_DISTRO/build,target=/home/ws/build,type=bind",
        "source=${localWorkspaceFolder}/..cache/ROS_DISTRO/install,target=/home/ws/install,type=bind",
        "source=${localWorkspaceFolder}/..cache/ROS_DISTRO/log,target=/home/ws/log,type=bind"
    ],
    "postCreateCommand": "sudo rosdep update && sudo rosdep install --from-paths src --ignore-src -y && sudo chown -R USERNAME /home/ws/"
}
```

Use Ctrl+F to open the search and replace menu. Search for `USERNAME` and replace it with your Linux username. If you do not know your username, you can find it by running `echo $USERNAME` in the terminal. Also replace
**Edit Dockerfile**

Open the Dockerfile and add the following contents:

```dockerfile
FROM ros:ROS_DISTRO
ARG USERNAME=USERNAME
ARG USER_UID=1000
ARG USER_GID=$USER_UID

# Create the user
RUN groupadd --gid $USER_GID $USERNAME \
    && useradd --uid $USER_UID --gid $USER_GID -m $USERNAME \
    #
    # [Optional] Add sudo support. Omit if you don't need to install software after connecting.
    && apt-get update \
    && apt-get install -y sudo \
    && echo $USERNAME ALL=(root) NOPASSWD:ALL > /etc/sudoers.d/$USERNAME \
    && chmod 0440 /etc/sudoers.d/$USERNAME
RUN apt-get update && apt-get upgrade -y
RUN apt-get install -y python3-pip
ENV SHELL /bin/bash

# ********************************************************
# * Anything else you want to do like clean up goes here *
# ********************************************************

# [Optional] Set the default user. Omit if you want to keep the default as root.
USER $USERNAME
CMD ["/bin/bash"]
```

Search here also for the `USERNAME` and replace it with your Linux username and the `ROS_DISTRO` with the ROS 2 distribution you wish to use and added to the cache previously.
Open and Build Development Container

Use View->Command Palette... or Ctrl+Shift+P to open the command palette. Search for the command Dev Containers: (Re-)build and Reopen in Container and execute it. This will build your development docker container for your. It will take a while - sit back or go for a coffee.

Test Container

To test if everything worked correctly, open a terminal in the container using View->Terminal or Ctrl+Shift+` and New Terminal in VS Code. Inside the terminal do the following:

```bash
sudo apt install ros-$ROS_DISTRO-rviz2 -y
source /opt/ros/$ROS_DISTRO/setup.bash
rviz2
```

Note: There might be a problem with displaying RVIZ. If no window pops up, then check the value of `echo $DISPLAY`
- if the output is 1, you can fix this problem with `echo "export DISPLAY=unix:1" >> /etc/bash.bashrc` and then test it again. You can also change the DISPLAY value in the devcontainer.json and rebuild it.

Building RQt from source

We’ve provided our development setup here to aid future users in easily extending RQt by creating their own plugins. We encourage you to contribute those plugins back to the ros-visualization GitHub repository!

System Requirements

These instructions are written for the target platforms for Crystal Clemmys (see REP).

- Ubuntu Bionic Beaver 18.04 64-bit
- Mac OSX Sierra 10.12.x
- Windows 10 with Visual Studio 2017

Other Requirements

- In ROS 2 Crystal the minimum Qt version is Qt5

Building From Source

In order to build RQt from source, first create a ROS 2 workspace at `~/ros2_ws/`. This step is already covered in building ROS 2 from source instructions, so we skip it here.
Download RQt Repositories

```
cd ~/ros2_ws
vcs import --force --input https://raw.githubusercontent.com/PickNikRobotics/rqt2_setup/
˓→master/rqt2.repos src
```

As an alternative to the hosted .repos file you can use rosininstall_generator to generate a custom one:

```
rosinstall_generator --rosdistro crystal --upstream-development --repos python_qt_
˓→-binding qt_gui_core rqt <more-repos-with-rqt-plugins> > rqt2.repos
vcs import --force src < rqt2.repos
```

Install Dependencies

Building RQt from source on macOS

This page provides specific information to building RQt from source on macOS. Follow these instructions before proceeding with RQt Source Install page.

System Requirements

RQt is supported on macOS 10.12, but 10.13 also seems to work.

Dependencies

The primary dependencies of the RQt package are sip and PyQt5. PySide2 may be supported in the future.

Install dependencies

```
$ brew install sip pyqt5
$ brew install graphviz
$ python3 -m pip install pygraphviz pydot
$ brew link --force qt
```

This is the quickest solution but may cause issues when upgrading Qt or if other packages are expecting Qt 4. Another option is to update your PATH and CMAKE_PREFIX_PATH to include the Qt install location:

```
$ export PATH="$(brew --prefix qt)/bin:$PATH"
$ export CMAKE_PREFIX_PATH="$(brew --prefix qt):$CMAKE_PREFIX_PATH"
```
Install RQt by source

Continue with the RQt source install page.

Building RQt from source on Windows 10

This page provides specific information to building RQt from source on Windows. Follow these instructions before proceeding with the RQt Source Install page.

If you have not done so, follow the ROS 2 Windows Development Setup guide before continuing.

System Requirements

- Windows 10
- Visual Studio 15.7.6

Currently Visual Studio 15.8 fails to build ROS 2 (see issue). Older versions of VS can be found here.

Dependencies

The primary dependencies of the RQt package are sip and PyQt5. PySide2 may be supported in the future. Even though they are provided through PyPI and chocolatey, you must install them by source to get compatible versions.

Install sip by source

Download from https://www.riverbankcomputing.com/software/sip/download

Run the x64 Native Tools Command Prompt as Administrator, and cd to the uncompressed source directory.

Run:

```bash
python3 configure.py
nmake
nmake install
```

If python3 is installed on your system as python, be sure to use that program name instead.

Install PyQt5 by source

Download from https://www.riverbankcomputing.com/software/pyqt/download5

Run the x64 Native Tools Command Prompt as Administrator, and cd to the uncompressed source directory. I ran into trouble with Qt 5.11.3 and PyQt5 compiling QtNfc, but it can be easily disabled.

```bash
python3 configure.py --disable QtNfc
nmake
nmake install
```
Test that it works

If install occurred without failure, try the commands below. They should run without issue and you should see 4.19.13 as your sip.exe version.

```
sip -V
python3 -c "from PyQt5 import QtCore"
```

Other dependencies

Install GraphViz from https://graphviz.gitlab.io/_pages/Download/Download_windows.html.

Install pydot and pyparsing:

```
pip3 install pydot pyparsing
```

PyGraphViz is a test dependency of qt_dotgraph, but it is currently unsupported on Windows and building by source is not straight forward. Manually merging this patch is the currently recommended solution (not verified): pygraphviz patch

Install RQt by source

Continue with the RQt source install page.

For non-Linux platforms, see the macOS RQt source install page or the Windows 10 RQt source install page before continuing here.

```
rosdep install --from-paths src --ignore-src --rosdistro bouncy -y --skip-keys "console_bridge fastcdr fastrtps rti-connex-ds-6.0.1 urdfdom_headers"
```

Build The Workspace

Generally building a workspace is as simple as:

```
colcon build
```

For Windows, it is recommended to use the --merge-install option.

```
colcon build --merge-install
```

Advanced Colcon usages:

- Show verbose output on the console:

```
colcon build -event-handlers console_direct+
```

- Only build one package and its dependencies:

```
colcon build -packages-up-to rqt_shell
```
Source your environment

Linux or macOS

```bash
. install/local_setup.bash
```

Windows

```bash
call install/local_setup.bat
```

Using RQt

See Overview of RQt.

4.8.5 Concepts

Conceptual overviews provide relatively high-level, general background information about key aspects of ROS 2.

Basic Concepts

ROS 2 is a middleware based on a strongly-typed, anonymous publish/subscribe mechanism that allows for message passing between different processes.

At the heart of any ROS 2 system is the ROS graph. The ROS graph refers to the network of nodes in a ROS system and the connections between them by which they communicate.

These are the concepts that will help you get started understanding the basics of ROS 2.

Nodes

A node is a participant in the ROS 2 graph, which uses a client library to communicate with other nodes. Nodes can communicate with other nodes within the same process, in a different process, or on a different machine. Nodes are typically the unit of computation in a ROS graph; each node should do one logical thing.

Nodes can publish to named topics to deliver data to other nodes, or subscribe to named topics to get data from other nodes. They can also act as a service client to have another node perform a computation on their behalf, or as a service server to provide functionality to other nodes. For long-running computations, a node can act as an action client to perform it, or as an action server to have another node perform it. Nodes can provide configurable parameters to change behavior during run-time.

Connections between nodes are established through a distributed discovery process.
Discovery

Discovery of nodes happens automatically through the underlying middleware of ROS 2. It can be summarized as follows:

1. When a node is started, it advertises its presence to other nodes on the network with the same ROS domain (set with the ROS_DOMAIN_ID environment variable). Nodes respond to this advertisement with information about themselves so that the appropriate connections can be made and the nodes can communicate.

2. Nodes periodically advertise their presence so that connections can be made with new-found entities, even after the initial discovery period.

3. Nodes advertise to other nodes when they go offline.

Nodes will only establish connections with other nodes if they have compatible Quality of Service settings.

Take the talker-listener demo for example. Running the C++ talker node in one terminal will publish messages on a topic, and the Python listener node running in another terminal will subscribe to messages on the same topic.

You should see that these nodes discover each other automatically, and begin to exchange messages.

Interfaces

Table of Contents

- Background
- Messages
  - Fields
    * Field types
    * Field names
    * Field default value
  - Constants
- Services
- Actions

Background

ROS applications typically communicate through interfaces of one of three types: topics, services, or actions. ROS 2 uses a simplified description language, the interface definition language (IDL), to describe these interfaces. This description makes it easy for ROS tools to automatically generate source code for the interface type in several target languages.

In this document we will describe the supported types:

- msg: .msg files are simple text files that describe the fields of a ROS message. They are used to generate source code for messages in different languages.

- srv: .srv files describe a service. They are composed of two parts: a request and a response. The request and response are message declarations.
• action: .action files describe actions. They are composed of three parts: a goal, a result, and feedback. Each part is a message declaration itself.

Messages

Messages are a way for a ROS 2 node to send data on the network to other ROS nodes, with no response expected. For instance, if a ROS 2 node reads temperature data from a sensor, it can then publish that data on the ROS 2 network using a Temperature message. Other nodes on the ROS 2 network can subscribe to that data and receive the Temperature message.

Messages are described and defined in .msg files in the msg/ directory of a ROS package. .msg files are composed of two parts: fields and constants.

Fields

Each field consists of a type and a name, separated by a space, i.e:

```
fieldtype1 fieldname1
fieldtype2 fieldname2
fieldtype3 fieldname3
```

For example:

```
int32 my_int
string my_string
```

Field types

Field types can be:

• a built-in-type

• names of Message descriptions defined on their own, such as “geometry_msgs/PoseStamped”

Built-in-types currently supported:

<table>
<thead>
<tr>
<th>Type name</th>
<th>C++</th>
<th>Python</th>
<th>DDS type</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>bool</td>
<td>builtns.bool</td>
<td>boolean</td>
</tr>
<tr>
<td>byte</td>
<td>uint8_t</td>
<td>builtns.bytes*</td>
<td>octet</td>
</tr>
<tr>
<td>char</td>
<td>char</td>
<td>builtns.str*</td>
<td>char</td>
</tr>
<tr>
<td>float32</td>
<td>float</td>
<td>builtns.float*</td>
<td>float</td>
</tr>
<tr>
<td>float64</td>
<td>double</td>
<td>builtns.float*</td>
<td>double</td>
</tr>
<tr>
<td>int8</td>
<td>int8_t</td>
<td>builtns.int*</td>
<td>octet</td>
</tr>
<tr>
<td>uint8</td>
<td>uint8_t</td>
<td>builtns.int*</td>
<td>octet</td>
</tr>
<tr>
<td>int16</td>
<td>int16_t</td>
<td>builtns.int*</td>
<td>short</td>
</tr>
<tr>
<td>uint16</td>
<td>uint16_t</td>
<td>builtns.int*</td>
<td>unsigned short</td>
</tr>
<tr>
<td>int32</td>
<td>int32_t</td>
<td>builtns.int*</td>
<td>long</td>
</tr>
<tr>
<td>uint32</td>
<td>uint32_t</td>
<td>builtns.int*</td>
<td>unsigned long</td>
</tr>
<tr>
<td>int64</td>
<td>int64_t</td>
<td>builtns.int*</td>
<td>long long</td>
</tr>
<tr>
<td>uint64</td>
<td>uint64_t</td>
<td>builtns.int*</td>
<td>unsigned long long</td>
</tr>
<tr>
<td>string</td>
<td>std::string</td>
<td>builtns.str</td>
<td>string</td>
</tr>
<tr>
<td>wstring</td>
<td>std::u16string</td>
<td>builtns.str</td>
<td>wstring</td>
</tr>
</tbody>
</table>
Every built-in-type can be used to define arrays:

<table>
<thead>
<tr>
<th>Type name</th>
<th>C++</th>
<th>Python</th>
<th>DDS type</th>
</tr>
</thead>
<tbody>
<tr>
<td>static array</td>
<td>std::array&lt;T, N&gt;</td>
<td>builtins.list*</td>
<td>T[N]</td>
</tr>
<tr>
<td>unbounded dynamic array</td>
<td>std::vector</td>
<td>builtins.list</td>
<td>sequence</td>
</tr>
<tr>
<td>bounded dynamic array</td>
<td>custom_class&lt;T, N&gt;</td>
<td>builtins.list*</td>
<td>sequence&lt;T, N&gt;</td>
</tr>
<tr>
<td>bounded string</td>
<td>std::string</td>
<td>builtins.str*</td>
<td>string</td>
</tr>
</tbody>
</table>

All types that are more permissive than their ROS definition enforce the ROS constraints in range and length by software.

Example of message definition using arrays and bounded types:

```cpp
int32[] unbounded_integer_array
int32[5] five_integers_array
int32[<=5] up_to_five_integers_array

string string_of_unbounded_size
string<10 up_to_ten_characters_string

string[<=5] up_to_five_unbounded_strings
string<10[] unbounded_array_of_strings_up_to_ten_characters_each
string<10[<=5] up_to_five_strings_up_to_ten_characters_each
```

Field names

Field names must be lowercase alphanumeric characters with underscores for separating words. They must start with an alphabetic character, and they must not end with an underscore or have two consecutive underscores.

Field default value

Default values can be set to any field in the message type. Currently default values are not supported for string arrays and complex types (i.e. types not present in the built-in-types table above; that applies to all nested messages).

Defining a default value is done by adding a third element to the field definition line, i.e:

```cpp
fieldtype fieldname fielddefaultvalue
```

For example:

```cpp
uint8 x 42
int16 y -2000
string full_name "John Doe"
int32[] samples [-200, -100, 0, 100, 200]
```

Note:

- string values must be defined in single ‘ or double " quotes
- currently string values are not escaped
**Constants**

Each constant definition is like a field description with a default value, except that this value can never be changed programatically. This value assignment is indicated by use of an equal ‘=’ sign, e.g.

```
constanttype CONSTANTNAME=constantvalue
```

For example:

```
int32 X=123
int32 Y=-123
string FOO="foo"
string EXAMPLE='bar'
```

**Note:** Constants names have to be UPPERCASE

**Services**

Services are a request/response communication, where the client (requester) is waiting for the server (responder) to make a short computation and return a result.

Services are described and defined in .srv files in the srv/ directory of a ROS package.

A service description file consists of a request and a response msg type, separated by `---`. Any two .msg files concatenated with a `---` are a legal service description.

Here is a very simple example of a service that takes in a string and returns a string:

```
string str
---
string str
```

We can of course get much more complicated (if you want to refer to a message from the same package you must not mention the package name):

```
# request constants
int8 FOO=1
int8 BAR=2
# request fields
int8 foobar
another_pkg/AnotherMessage msg
---
# response constants
uint32 SECRET=123456
# response fields
another_pkg/YetAnotherMessage val
CustomMessageDefinedInThisPackage value
uint32 an_integer
```

You cannot embed another service inside of a service.
**Actions**

Actions are a long-running request/response communication, where the action client (requester) is waiting for the action server (responder) to take some action and return a result. In contrast to services, actions can be long-running (many seconds or minutes), provide feedback while they are happening, and can be interrupted.

Action definitions have the following form:

```
<request_type> <request_fieldname>
---
<response_type> <response_fieldname>
---
<feedback_type> <feedback_fieldname>
```

Like services, the request fields are before and the response fields are after the first triple-dash (---), respectively. There is also a third set of fields after the second triple-dash, which is the fields to be sent when sending feedback.

There can be arbitrary numbers of request fields (including zero), arbitrary numbers of response fields (including zero), and arbitrary numbers of feedback fields (including zero).

The `<request_type>`, `<response_type>`, and `<feedback_type>` follow all of the same rules as the `<type>` for a message. The `<request_fieldname>`, `<response_fieldname>`, and `<feedback_fieldname>` follow all of the same rules as the `<fieldname>` for a message.

For instance, the Fibonacci action definition contains the following:

```
int32 order
---
int32[] sequence
---
int32[] sequence
```

This is an action definition where the action client is sending a single `int32` field representing the number of Fibonacci steps to take, and expecting the action server to produce an array of `int32` containing the complete steps. Along the way, the action server may also provide an intermediate array of `int32` contains the steps accomplished up until a certain point.

**Topics**

**Table of Contents**

- *Publish/Subscribe*
- *Anonymous*
- *Strongly-typed*

Topics are one of the three primary styles of interfaces provided by ROS 2. Topics should be used for continuous data streams, like sensor data, robot state, etc.

As stated earlier, ROS 2 is a strongly-typed, anonymous publish/subscribe system. Let’s break down that sentence and explain it a bit more.
Publish/Subscribe

A publish/subscribe system is one in which there are producers of data (publishers) and consumers of data (subscribers). The publishers and subscribers know how to contact each other through the concept of a “topic”, which is a common name so that the entities can find each other. For instance, when you create a publisher, you must also give it a string that is the name of the topic; the same goes for the subscriber. Any publishers and subscribers that are on the same topic name can directly communicate with each other. There may be zero or more publishers and zero or more subscribers on any particular topic. When data is published to the topic by any of the publishers, all subscribers in the system will receive the data. This system is also known as a “bus”, since it somewhat resembles a device bus from electrical engineering. This concept of a bus is part of what makes ROS 2 a powerful and flexible system. Publishers and subscribers can come and go as needed, meaning that debugging and introspection are natural extensions to the system. For instance, if you want to record data, you can use the `ros2 bag record` command. Under the hood, `ros2 bag record` creates a new subscriber to whatever topic you tell it, without interrupting the flow of data to the other parts of the system.

Anonymous

Another fact mentioned in the introduction is that ROS 2 is “anonymous”. This means that when a subscriber gets a piece of data, it doesn’t generally know or care which publisher originally sent it (though it can find out if it wants). The benefit to this architecture is that publishers and subscribers can be swapped out at will without affecting the rest of the system.

Strongly-typed

Finally, the introduction also mentioned that the publish/subscribe system is “strongly-typed”. That has two meanings in this context:

1. The types of each field in a ROS message are typed, and that type is enforced at various levels. For instance, if the ROS message contains:

```
uint32 field1
string field2
```

Then the code will ensure that `field` is always an unsigned integer and that `field2` is always a string.

2. The semantics of each field are well-defined. There is no automated mechanism to ensure this, but all of the core ROS types have strong semantics associated with them. For instance, the IMU message contains a 3-dimensional vector for the measured angular velocity, and each of the dimensions is specified to be in radians/second. Other interpretations should not be placed into the message.

Services

Table of Contents

- Service server
- Service client

In ROS 2, a service refers to a remote procedure call. In other words, a node can make a remote procedure call to another node which will do a computation and return a result.

This structure is reflected in how a service message definition looks:
In ROS 2, services are expected to return quickly, as the client is generally waiting on the result. Services should never be used for longer running processes, in particular processes that might need to be preempted for exceptional situations. If you have a service that will be doing a long-running computation, consider using an action instead.

Services are identified by a service name, which looks much like a topic name (but is in a different namespace). A service consists of two parts: the service server and the service client.

**Service server**

A service server is the entity that will accept a remote procedure request, and perform some computation on it. For instance, suppose the ROS 2 message contains the following:

```plaintext
uint32 a
uint32 b
---
uint32 sum
```

The service server would be the entity that receives this message, adds \( a \) and \( b \) together, and returns the \( \text{sum} \).  

**Note:** There should only ever be one service server per service name. It is undefined which service server will receive client requests in the case of multiple service servers on the same service name.

**Service client**

A service client is an entity that will request a remote service server to perform a computation on its behalf. Following from the example above, the service client is the entity that creates the initial message containing \( a \) and \( b \), and waits for the service server to compute the sum and return the result.

Unlike the service server, there can be arbitrary numbers of service clients using the same service name.

**Actions**

In ROS 2, an action refers to a long-running remote procedure call with feedback and the ability to cancel or preempt the goal. For instance, the high-level state machine running a robot may call an action to tell the navigation subsystem to travel to a waypoint, which may take several seconds (or minutes) to do. Along the way, the navigation subsystem can provide feedback on how far along it is, and the high-level state machine has the option to cancel or preempt the travel to that waypoint.

This structure is reflected in how an action message definition looks:

```plaintext
```
In ROS 2, actions are expected to be long running procedures, as there is overhead in setting up and monitoring the connection. If you need a short running remote procedure call, consider using a *service* instead.

Actions are identified by an action name, which looks much like a topic name (but is in a different namespace). An action consists of two parts: the action server and the action client.

**Action server**

The action server is the entity that will accept the remote procedure request and perform some procedure on it. It is also responsible for sending out feedback as the action progresses, and should react to cancellation/preemption requests. For instance, suppose the action is supposed to calculate the Fibonacci sequence, and the message contains the following:

```
int32 order
---
int32[] sequence
---
int32[] sequence
```

The action server is the entity that receives this message, starts calculating the sequence up to `order` (providing feedback along the way), and finally returning a full result in `sequence`.

**Note:** There should only ever be one action server per action name. It is undefined which action server will receive client requests in the case of multiple action servers on the same action name.

**Action client**

An action client is an entity that will request a remote action server to perform a procedure on its behalf. Following from the example above, the action client is the entity that creates the initial message containing the `order`, and waits for the action server to compute the sequence and return it (with feedback along the way).

Unlike the action server, there can be arbitrary numbers of actions clients using the same action name.

**Parameters**

**Table of Contents**

- Overview
- Parameters background
  - Declaring parameters
  - Parameter types
Overview

Parameters in ROS 2 are associated with individual nodes. Parameters are used to configure nodes at startup (and during runtime), without changing the code. The lifetime of a parameter is tied to the lifetime of the node (though the node could implement some sort of persistence to reload values after restart).

Parameters are addressed by node name, node namespace, parameter name, and parameter namespace. Providing a parameter namespace is optional.

Each parameter consists of a key, a value, and a descriptor. The key is a string and the value is one of the following types: bool, int64, float64, string, byte[], bool[], int64[], float64[] or string[]. By default all descriptors are empty, but can contain parameter descriptions, value ranges, type information, and additional constraints.

For an hands-on tutorial with ROS parameters see Understanding parameters.

Parameters background

Declaring parameters

By default, a node needs to declare all of the parameters that it will accept during its lifetime. This is so that the type and name of the parameters are well-defined at node startup time, which reduces the chances of misconfiguration later on. See Using parameters in a class (C++) or Using parameters in a class (Python) for tutorials on declaring and using parameters from a node.

For some types of nodes, not all of the parameters will be known ahead of time. In these cases, the node can be instantiated with allow_undeclared_parameters set to true, which will allow parameters to be get and set on the node even if they haven’t been declared.

Parameter types

Each parameter on a ROS 2 node has one of the pre-defined parameter types as mentioned in the Overview. By default, attempts to change the type of a declared parameter at runtime will fail. This prevents common mistakes, such as putting a boolean value into an integer parameter.

If a parameter needs to be multiple different types, and the code using the parameter can handle it, this default behavior can be changed. When the parameter is declared, it should be declared using a ParameterDescriptor with the dynamic_typing member variable set to true.
Parameter callbacks

A ROS 2 node can register three different types of callbacks to be informed when changes are happening to parameters. All three of the callbacks are optional.

The first is known as a "preset parameter" callback, and can be set by calling `add_pre_set_parameters_callback` from the node API. This callback is passed a list of the `Parameter` objects that are being changed, and returns nothing. When it is called, it can modify the `Parameter` list to change, add, or remove entries. As an example, if `parameter2` should change anytime that `parameter1` changes, that can be implemented with this callback.

The second is known as a "set parameter" callback, and can be set by calling `add_on_set_parameters_callback` from the node API. The callback is passed a list of immutable `Parameter` objects, and returns an `rcl_interfaces/msg/SetParametersResult`. The main purpose of this callback is to give the user the ability to inspect the upcoming change to the parameter and explicitly reject the change.

**Note:** It is important that "set parameter" callbacks have no side-effects. Since multiple "set parameter" callbacks can be chained, there is no way for an individual callback to know if a later callback will reject the update. If the individual callback were to make changes to the class it is in, for instance, it may get out-of-sync with the actual parameter. To get a callback after a parameter has been successfully changed, see the next type of callback below.

The third type of callback is known as an "post set parameter" callback, and can be set by calling `add_post_set_parameters_callback` from the node API. The callback is passed a list of immutable `Parameter` objects, and returns nothing. The main purpose of this callback is to give the user the ability to react to changes from parameters that have successfully been accepted.

The ROS 2 demos have an example of all of these callbacks in use.

Interacting with parameters

ROS 2 nodes can perform parameter operations through node APIs as described in Using parameters in a class (C++) or Using parameters in a class (Python). External processes can perform parameter operations via parameter services that are created by default when a node is instantiated. The services that are created by default are:

- `/node_name/describe_parameters`: Uses a service type of `rcl_interfaces/srv/DescribeParameters`. Given a list of parameter names, returns a list of descriptors associated with the parameters.
- `/node_name/get_parameter_types`: Uses a service type of `rcl_interfaces/srv/GetParameterTypes`. Given a list of parameter names, returns a list of parameter types associated with the parameters.
- `/node_name/get_parameters`: Uses a service type of `rcl_interfaces/srv/GetParameters`. Given a list of parameter names, returns a list of parameter values associated with the parameters.
- `/node_name/list_parameters`: Uses a service type of `rcl_interfaces/srv/ListParameters`. Given an optional list of parameter prefixes, returns a list of the available parameters with that prefix. If the prefixes are empty, returns all parameters.
- `/node_name/set_parameters`: Uses a service type of `rcl_interfaces/srv/SetParameters`. Given a list of parameter names and values, attempts to set the parameters on the node. Returns a list of results from trying to set each parameter; some of them may have succeeded and some may have failed.
- `/node_name/set_parameters_atomically`: Uses a service type of `rcl_interfaces/srv/SetParametersAtomically`. Given a list of parameter names and values, attempts to set the parameters on the node. Returns a single result from trying to set all parameters, so if one failed, all of them failed.
Setting initial parameter values when running a node

Initial parameter values can be set when running the node either through individual command-line arguments, or through YAML files. See Setting parameters directly from the command line for examples on how to set initial parameter values.

Setting initial parameter values when launching nodes

Initial parameter values can also be set when running the node through the ROS 2 launch facility. See this document for information on how to specify parameters via launch.

Manipulating parameter values at runtime

The ros2 param command is the general way to interact with parameters for nodes that are already running. ros2 param uses the parameter service API as described above to perform the various operations. See this how-to guide for details on how to use ros2 param.

Migrating from ROS 1

The Launch file migration guide explains how to migrate param and rosparam launch tags from ROS 1 to ROS 2. The YAML parameter file migration guide explains how to migrate parameter files from ROS 1 to ROS 2.

In ROS 1, the roscore acted like a global parameter blackboard where all nodes could get and set parameters. Since there is no central roscore in ROS 2, that functionality no longer exists. The recommended approach in ROS 2 is to use per-node parameters that are closely tied to the nodes that use them. If a global blackboard is still needed, it is possible to create a dedicated node for this purpose. ROS 2 ships with one in the ros-iron-demo-nodes-cpp package called parameter_blackboard; it can be run with:

```bash
ros2 run demo_nodes_cpp parameter_blackboard
```

The code for the parameter_blackboard is here.

Introspection with command line tools

ROS 2 includes a suite of command-line tools for introspecting a ROS 2 system.
Usage

The main entry point for the tools is the command **ros2**, which itself has various sub-commands for introspecting and working with nodes, topics, services, and more.

To see all available sub-commands run:

```
ros2 --help
```

Examples of sub-commands that are available include:

- **action**: Introspect/interact with ROS actions
- **bag**: Record/play a rosbag
- **component**: Manage component containers
- **daemon**: Introspect/configure the ROS 2 daemon
- **doctor**: Check ROS setup for potential issues
- **interface**: Show information about ROS interfaces
- **launch**: Run/introspect a launch file
- **lifecycle**: Introspect/manage nodes with managed lifecycles
- **multicast**: Multicast debugging commands
- **node**: Introspect ROS nodes
- **param**: Introspect/configure parameters on a node
- **pkg**: Introspect ROS packages
- **run**: Run ROS nodes
- **security**: Configure security settings
- **service**: Introspect/call ROS services
- **test**: Run a ROS launch test
- **topic**: Introspect/publish ROS topics
- **trace**: Tracing tools to get information on ROS nodes execution (only available on Linux)
- **wtf**: An alias for **doctor**

Example

To produce the typical talker-listener example using command-line tools, the **topic** sub-command can be used to publish and echo messages on a topic.

Publish messages in one terminal with:

```
$ ros2 topic pub /chatter std_msgs/msg/String "data: Hello world"
publisher: beginning loop
publishing  #1: std_msgs.msg.String(data=Hello world)
publishing  #2: std_msgs.msg.String(data=Hello world)
```

Echo messages received in another terminal with:
$ ros2 topic echo /chatter
data: Hello world

data: Hello world

**Behind the scenes**

ROS 2 uses a distributed discovery process for nodes to connect to each other. As this process purposefully does not use a centralized discovery mechanism, it can take time for ROS nodes to discover all other participants in the ROS graph. Because of this, there is a long-running daemon in the background that stores information about the ROS graph to provide faster responses to queries, e.g. the list of node names.

The daemon is automatically started when the relevant command-line tools are used for the first time. You can run `ros2 daemon --help` for more options for interacting with the daemon.

**Implementation**

The source code for the `ros2` command is available at https://github.com/ros2/ros2cli.

The `ros2` tool has been implemented as a framework that can be extended via plugins. For example, the `sros2` package provides a `security` sub-command that is automatically detected by the `ros2` tool if the `sros2` package is installed.

**Launch**

A ROS 2 system typically consists of many nodes running across many different processes (and even different machines). While it is possible to run each of these nodes separately, it gets cumbersome quite quickly.

The launch system in ROS 2 is meant to automate the running of many nodes with a single command. It helps the user describe the configuration of their system and then executes it as described. The configuration of the system includes what programs to run, where to run them, what arguments to pass them, and ROS-specific conventions which make it easy to reuse components throughout the system by giving them each a different configuration. It is also responsible for monitoring the state of the processes launched, and reporting and/or reacting to changes in the state of those processes.

All of the above is specified in a launch file, which can be written in Python, XML, or YAML. This launch file can then be run using the `ros2 launch` command, and all of the nodes specified will be run.

The design document details the goal of the design of ROS 2’s launch system (not all functionality is currently available).

**Client libraries**

---

### Table of Contents

- **Overview**
- **Supported client libraries**
  - The `rclcpp` package
  - The `rclpy` package
  - Community-maintained

---
Overview

Client libraries are the APIs that allow users to implement their ROS 2 code. Using client libraries, users gain access to ROS 2 concepts such as nodes, topics, services, etc. Client libraries come in a variety of programming languages so that users may write ROS 2 code in the language that is best-suited for their application. For example, you might prefer to write visualization tools in Python because it makes prototyping iterations faster, while for parts of your system that are concerned with efficiency, the nodes might be better implemented in C++.

Nodes written using different client libraries are able to share messages with each other because all client libraries implement code generators that provide users with the capability to interact with ROS 2 interface files in the respective language.

In addition to the language-specific communication tools, client libraries expose to users the core functionality that makes ROS “ROS”. For example, here is a list of functionality that can typically be accessed through a client library:

- Names and namespaces
- Time (real or simulated)
- Parameters
- Console logging
- Threading model
- Intra-process communication

Supported client libraries

The C++ client library (rclcpp) and the Python client library (rclpy) are both client libraries which utilize common functionality in rcl.

The rclcpp package

The ROS Client Library for C++ (rclcpp) is the user facing, C++ idiomatic interface which provides all of the ROS client functionality like creating nodes, publishers, and subscriptions. rclcpp builds on top of rcl and the rosidl API, and it is designed to be used with the C++ messages generated by rosidl_generator_cpp.

rclcpp makes use of all the features of C++ and C++17 to make the interface as easy to use as possible, but since it reuses the implementation in rcl it is able maintain a consistent behavior with the other client libraries that use the rcl API.

The rclcpp repository is located on GitHub at ros2/rclcpp and contains the package rclcpp. The generated API documentation is here:

api/rclcpp/index.html
The **rclpy** package

The ROS Client Library for Python (rclpy) is the Python counterpart to the C++ client library. Like the C++ client library, rclpy also builds on top of the rcl C API for its implementation. The interface provides an idiomatic Python experience that uses native Python types and patterns like lists and context objects. By using the rcl API in the implementation, it stays consistent with the other client libraries in terms of feature parity and behavior. In addition to providing Python idiomatic bindings around the rcl API and Python classes for each message, the Python client library takes care of the execution model, using threading.Thread or similar to run the functions in the rcl API.

Like C++ it generates custom Python code for each ROS message that the user interacts with, but unlike C++ it eventually converts the native Python message object into the C version of the message. All operations happen on the Python version of the messages until they need to be passed into the rcl layer, at which point they are converted into the plain C version of the message so it can be passed into the rcl C API. This is avoided if possible when communicating between publishers and subscriptions in the same process to cut down on the conversion into and out of Python.

The rclpy repository is located on GitHub at ros2/rclpy and contains the package rclpy. The generated API documentation is here:

api/rclpy/index.html

**Community-maintained**

While the C++ and Python client libraries are maintained by the core ROS 2 team, members of the ROS 2 community maintain additional client libraries:

- **C rclc** does not put a layer on top of rcl but complements rcl to make rcl+rclc a feature-complete client library in C. See micro.ros.org for tutorials.

- **JVM and Android** Java and Android bindings for ROS 2.

- **.NET Core, UWP and C#** This is a collection of projects (bindings, code generator, examples and more) for writing ROS 2 applications for .NET Core and .NET Standard.

- **Node.js** rclnodejs is a Node.js client for ROS 2. It provides a simple and easy JavaScript API for ROS 2 programming.

- **Rust** This is a set of projects (the rcrs client library, code generator, examples and more) that enables developers to write ROS 2 applications in Rust.

Older, unmaintained client libraries are:

- **Ada**
- **C#**
- **Objective C and iOS**
- **Zig**
Common functionality: \texttt{rcl}

Most of the functionality found in a client library is not specific to the programming language of the client library. For example, the behavior of parameters and the logic of namespaces should ideally be the same across all programming languages. Because of this, rather than implementing the common functionality from scratch, client libraries make use of a common core ROS Client Library (RCL) interface that implements logic and behavior of ROS concepts that is not language-specific. As a result, client libraries only need to wrap the common functionality in the RCL with foreign function interfaces. This keeps client libraries thinner and easier to develop. For this reason the common RCL functionality is exposed with C interfaces as the C language is typically the easiest language for client libraries to wrap.

In addition to making the client libraries light-weight, an advantage of having the common core is that the behavior between languages is more consistent. If any changes are made to the logic/behavior of the functionality in the core RCL – namespaces, for example – all client libraries that use the RCL will have these changes reflected. Furthermore, having the common core means that maintaining multiple client libraries becomes less work when it comes to bug fixes.

The API documentation for \texttt{rcl} can be found here.

Language-specific functionality

Client library concepts that require language-specific features/properties are not implemented in the RCL but instead are implemented in each client library. For example, threading models used by “spin” functions will have implementations that are specific to the language of the client library.

Demo

For a walkthrough of the message exchange between a publisher using \texttt{rclpy} and a subscription using \texttt{rclcpp}, we encourage you to watch this ROSCon talk starting at 17:25 (see the slides here).

Comparison to ROS 1

In ROS 1, all client libraries are developed “from the ground up”. This allows for the ROS 1 Python client library to be implemented purely in Python, for example, which brings benefits of such as not needing to compile code. However, naming conventions and behaviors are not always consistent between client libraries, bug fixes have to be done in multiple places, and there is a lot of functionality that has only ever been implemented in one client library (e.g. UDPROS).

Summary

By utilizing the common core ROS client library, client libraries written in a variety of programming languages are easier to write and have more consistent behavior.
Intermediate Concepts

These are the concepts that further your understanding of a basic ROS 2 system.

The ROS_DOMAIN_ID

Table of Contents

- Overview
- Choosing a domain ID (short version)
- Choosing a domain ID (long version)
  - Platform-specific constraints
  - Participant constraints
  - Domain ID to UDP Port Calculator

Overview

As explained elsewhere, the default middleware that ROS 2 uses for communication is DDS. In DDS, the primary mechanism for having different logical networks share a physical network is known as the Domain ID. ROS 2 nodes on the same domain can freely discover and send messages to each other, while ROS 2 nodes on different domains cannot. All ROS 2 nodes use domain ID 0 by default. To avoid interference between different groups of computers running ROS 2 on the same network, a different domain ID should be set for each group.

Choosing a domain ID (short version)

The text below explains the derivation of the range of domain IDs that should be used in ROS 2. To skip that background and just choose a safe number, simply choose a domain ID between 0 and 101, inclusive.

Choosing a domain ID (long version)

The domain ID is used by DDS to compute the UDP ports that will be used for discovery and communication. See this article for details on how the ports are computed. Remembering our basic networking, the UDP port is an unsigned 16-bit integer. Thus, the highest port number that can be allocated is 65535. Doing some math with the formula in the article above, this means that the highest domain ID that can possibly be assigned is 232, while the lowest that can be assigned is 0.
Platform-specific constraints

For maximum compatibility, some additional platform-specific constraints should be followed when choosing a domain ID. In particular, it is best to avoid allocating domain IDs in the operating system’s ephemeral port range. This avoids possible conflicts between the ports used by the ROS 2 nodes and other networking services on the computers.

Here are some platform-specific notes about ephemeral ports.

Linux

By default, the Linux kernel uses ports 32768-60999 for ephemeral ports. This means that domain IDs 0-101 and 215-232 can be safely used without colliding with ephemeral ports. The ephemeral port range is configurable in Linux by setting custom values in `/proc/sys/net/ipv4/ip_local_port_range`. If a custom ephemeral port range is used, the above numbers may have to be adjusted accordingly.

macOS

By default, the ephemeral port range on macOS is 49152-65535. This means that domain IDs 0-166 can be safely used without colliding with ephemeral ports. The ephemeral port range is configurable in macOS by setting custom sysctl values for `net.inet.ip.portrange.first` and `net.inet.ip.portrange.last`. If a custom ephemeral port range is used, the above numbers may have to be adjusted accordingly.

Windows

By default, the ephemeral port range on Windows is 49152-65535. This means that domain IDs 0-166 can be safely used without colliding with ephemeral ports. The ephemeral port range is configurable in Windows by using `netsh`. If a custom ephemeral port range is used, the above numbers may have to be adjusted accordingly.

Participant constraints

For each ROS 2 process running on a computer, one DDS “participant” is created. Since each DDS participant takes up two ports on the computer, running more than 120 ROS 2 processes on one computer may spill over into other domain IDs or the ephemeral ports.

To see why, consider the domain IDs 1 and 2.

- Domain ID 1 uses port 7650 and 7651 for multicast.
- Domain ID 2 uses port 7900 and 7901 for multicast.
- When creating the 1st process (zeroth participant) in domain ID 1, the ports 7660 and 7661 are used for unicast.
- When creating the 120th process (119th participant) in domain ID 1, the ports 7898 and 7899 are used for unicast.
- When creating the 121st process (120th participant) in domain ID 1, the ports 7900 and 7901 are used for unicast and overlap with domain ID 2.

If it is known that the computer will only ever be on a single domain ID at a time, and the domain ID is low enough, it is safe to create more ROS 2 processes than this.

When choosing a domain ID that is near the top of the range of platform-specific domain IDs, one additional constraint should be considered.

For instance, assume a Linux computer with a domain ID of 101:

- The zero’th ROS 2 process on the computer will connect to ports 32650, 32651, 32660, and 32661.
- The first ROS 2 process on the computer will connect to ports 32650, 32651, 32662, and 32663.
- The 53rd ROS 2 process on the computer will connect to ports 32650, 32651, 32766, and 32767.
- The 54th ROS 2 process on the computer will connect to ports 32650, 32651, 32768, and 32769, running into the ephemeral port range.
Thus the maximum number of processes that should be created when using domain ID 101 on Linux is 54. Similarly, the maximum number of processes that should be created when using domain ID 232 on Linux is 63, as the maximum port number is 65535.

The situation is similar on macOS and Windows, though the numbers are different. On macOS and Windows, when choosing a domain ID of 166 (the top of the range), the maximum number of ROS 2 processes that can be created on a computer before running into the ephemeral port range is 120.

**Domain ID to UDP Port Calculator**

**Different ROS 2 DDS/RTPS vendors**

ROS 2 is built on top of DDS/RTPS as its middleware, which provides discovery, serialization and transportation. This article explains the motivation behind using DDS implementations, and/or the RTPS wire protocol of DDS, in detail. In summary, DDS is an end-to-end middleware that provides features which are relevant to ROS systems, such as distributed discovery (not centralized like in ROS 1) and control over different “Quality of Service” options for the transportation.

DDS is an industry standard which is implemented by a range of vendors, such as RTI’s Connext DDS, eProsima’s Fast DDS, Eclipse’s Cyclone DDS, or GurumNetworks’s GurumDDS. RTPS (a.k.a. DDSI-RTPS) is the wire protocol used by DDS to communicate over the network.

ROS 2 supports multiple DDS/RTPS implementations because it is not necessarily “one size fits all” when it comes to choosing a vendor/implementation. There are many factors you might consider while choosing a middleware implementation: logistical considerations like the license, or technical considerations like platform availability, or computation footprint. Vendors may provide more than one DDS or RTPS implementation targeted at meeting different needs. For example, RTI has a few variations of their Connext implementation that vary in purpose, like one that specifically targets microcontrollers and another which targets applications requiring special safety certifications (we only support their standard desktop version at this time).

In order to use a DDS/RTPS implementation with ROS 2, a “ROS Middleware interface” (a.k.a. rmw interface or just rmw) package needs to be created that implements the abstract ROS middleware interface using the DDS or RTPS implementation’s API and tools. It’s a lot of work to implement and maintain RMW packages for supporting DDS implementations, but supporting at least a few implementations is important for ensuring that the ROS 2 codebase is not tied to any one particular implementation, as users may wish to switch out implementations depending on their project’s needs.

**Supported RMW implementations**

<table>
<thead>
<tr>
<th>Product name</th>
<th>License</th>
<th>RMW implementation</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>eProsima Fast DDS</td>
<td>Apache 2</td>
<td>rmw_fastrtps_cpp</td>
<td>Full support. Default RMW. Packaged with binary releases.</td>
</tr>
<tr>
<td>Eclipse Cyclone DDS</td>
<td>Eclipse Public License v2.0</td>
<td>rmw_cyclonedds_cpp</td>
<td>Full support. Packaged with binary releases.</td>
</tr>
<tr>
<td>RTI Connext DDS</td>
<td>commercial, research</td>
<td>rmw_connextdds_cpp</td>
<td>Full support. Support included in binaries, but Connext installed separately.</td>
</tr>
<tr>
<td>GurumNetworks GurumDDS</td>
<td>commercial</td>
<td>rmw_gurumdds_cpp</td>
<td>Community support. Support included in binaries, but GurumDDS installed separately.</td>
</tr>
</tbody>
</table>

For practical information on working with multiple RMW implementations, see the “Working with multiple RMW implementations” tutorial.
Multiple RMW implementations

The ROS 2 binary releases for currently active distros have built-in support for several RMW implementations out of the box (Fast DDS, RTI Connext Pro, Eclipse Cyclone DDS, GurumNetworks GurumDDS). The default is Fast DDS, which works without any additional installation steps because we distribute it with our binary packages.

Other RMWs like Cyclone DDS, Connext or GurumDDS can be enabled by installing additional packages, but without having to rebuild anything or replace any existing packages.

A ROS 2 workspace that has been built from source may build and install multiple RMW implementations simultaneously. While the core ROS 2 code is being compiled, any RMW implementation that is found will be built if the relevant DDS/RTPS implementation has been installed properly and the relevant environment variables have been configured. For example, if the code for the RMW package for RTI Connext DDS is in the workspace, it will be built if an installation of RTI’s Connext Pro can also be found.

For many cases you will find that nodes using different RMW implementations are able to communicate, however this is not true under all circumstances. Here is a list of inter-vendor communication configurations that are not supported:

- Fast DDS <-> Connext
  - WideString published by Fast DDS can’t be received correctly by Connext on macOS
- Connext <-> Cyclone DDS
  - does not support pub/sub communication for WideString

Default RMW implementation

If a ROS 2 workspace has multiple RMW implementations, Fast DDS is selected as the default RMW implementation if it is available. If the Fast DDS RMW implementation is not installed, the RMW implementation with the first RMW implementation identifier in alphabetical order will be used. The implementation identifier is the name of the ROS package that provides the RMW implementation, e.g. rmw_cyclonedds_cpp. For example, if both rmw_cyclonedds_cpp and rmw_connextdds ROS packages are installed, rmw_connextdds would be the default. If rmw_fastrtps_cpp is ever installed, it would be the default.

See the guide for how to specify which RMW implementation is to be used when running the ROS 2 examples.

Logging and logger configuration

Table of Contents

- Overview
- Severity level
- APIs
- Configuration
  - Environment variables
  - Node creation
- Logging subsystem design
  - rcutils
  - rcl_logging_spdlog

4.8. ROS 2 Documentation
Overview

The logging subsystem in ROS 2 aims to deliver logging messages to a variety of targets, including:

- To the console (if one is attached)
- To log files on disk (if local storage is available)
- To the `/rosout` topic on the ROS 2 network

By default, log messages in ROS 2 nodes will go out to the console (on stderr), to log files on disk, and to the `/rosout` topic on the ROS 2 network. All of the targets can be individually enabled or disabled on a per-node basis.

The rest of this document will go over some of the ideas behind the logging subsystem.

Severity level

Log messages have a severity level associated with them: `DEBUG`, `INFO`, `WARN`, `ERROR` or `FATAL`, in ascending order. A logger will only process log messages with severity at or higher than a specified level chosen for the logger.

Each node has a logger associated with it that automatically includes the node’s name and namespace. If the node’s name is externally remapped to something other than what is defined in the source code, it will be reflected in the logger name. Non-node loggers can also be created that use a specific name.

Logger names represent a hierarchy. If the level of a logger named “abc.def” is unset, it will defer to the level of its parent named “abc”, and if that level is also unset, the default logger level will be used. When the level of logger “abc” is changed, all of its descendants (e.g. “abc.def”, “abc.ghi.jkl”) will have their level impacted unless their level has been explicitly set.

APIs

These are the APIs that end users of the ROS 2 logging infrastructure should use, split up by client library.

C++

- `RCLCPP_DEBUG`, `RCLCPP_INFO`, `RCLCPP_WARN`, `RCLCPP_ERROR`, `RCLCPP_FATAL` - output the given printf-style message every time this line is hit
- `RCLCPP_DEBUG_ONCE`, `RCLCPP_INFO_ONCE`, `RCLCPP_WARN_ONCE`, `RCLCPP_ERROR_ONCE`, `RCLCPP_FATAL_ONCE` - output the given printf-style message only the first time this line is hit
- `RCLCPP_DEBUG_EXPRESSION`, `RCLCPP_INFO_EXPRESSION`, `RCLCPP_WARN_EXPRESSION`, `RCLCPP_ERROR_EXPRESSION`, `RCLCPP_FATAL_EXPRESSION` - output the given printf-style message only if the given expression is true
- `RCLCPP_DEBUG_FUNCTION`, `RCLCPP_INFO_FUNCTION`, `RCLCPP_WARN_FUNCTION`, `RCLCPP_ERROR_FUNCTION`, `RCLCPP_FATAL_FUNCTION` - output the given printf-style message only if the given function returns true
- `RCLCPP_DEBUG_SKIPFIRST`, `RCLCPP_INFO_SKIPFIRST`, `RCLCPP_WARN_SKIPFIRST`, `RCLCPP_ERROR_SKIPFIRST`, `RCLCPP_FATAL_SKIPFIRST` - output the given printf-style message all but the first time this line is hit
• RCLCPP_{DEBUG, INFO, WARN, ERROR, FATAL}_THROTTLE - output the given printf-style message no more than the given rate in integer milliseconds

• RCLCPP_{DEBUG, INFO, WARN, ERROR, FATAL}_SKIPFIRST_THROTTLE - output the given printf-style message no more than the given rate in integer milliseconds, but skip the first

• RCLCPP_{DEBUG, INFO, WARN, ERROR, FATAL}_STREAM - output the given C++ stream-style message every time this line is hit

• RCLCPP_{DEBUG, INFO, WARN, ERROR, FATAL}_STREAM_ONCE - output the given C++ stream-style message only the first time this line is hit

• RCLCPP_{DEBUG, INFO, WARN, ERROR, FATAL}_STREAM_EXPRESSION - output the given C++ stream-style message only if the given expression is true

• RCLCPP_{DEBUG, INFO, WARN, ERROR, FATAL}_STREAM_FUNCTION - output the given C++ stream-style message only if the given function returns true

• RCLCPP_{DEBUG, INFO, WARN, ERROR, FATAL}_STREAM_SKIPFIRST - output the given C++ stream-style message all but the first time this line is hit

• RCLCPP_{DEBUG, INFO, WARN, ERROR, FATAL}_STREAM_THROTTLE - output the given C++ stream-style message no more than the given rate in integer milliseconds

• RCLCPP_{DEBUG, INFO, WARN, ERROR, FATAL}_STREAM_SKIPFIRST_THROTTLE - output the given C++ stream-style message no more than the given rate in integer milliseconds, but skip the first

Each of the above APIs takes an rclcpp::Logger object as the first argument. This can be pulled from the node API by calling node->get_logger() (recommended), or by constructing a stand-alone rclcpp::Logger object.

• rcutils_logging_set_logger_level - Set the logging level for a particular logger name to the given severity level

• rcutils_logging_get_logger_effective_level - Given a logger name, return the logger level (which may be unset)

Python

• logger.{debug, info, warning, error, fatal} - output the given Python string to the logging infrastructure. The calls accept the following keyword args to control behavior:
  – throttle_duration_sec - if not None, the duration of the throttle interval in floating-point seconds
  – skip_first - if True, output the message all but the first time this line is hit
  – once - if True, only output the message the first time this line is hit

• rclpy.logging.set_logger_level - Set the logging level for a particular logger name to the given severity level

• rclpy.logging.get_logger_effective_level - Given a logger name, return the logger level (which may be unset)
Configuration

Since rclcpp and rclpy use the same underlying logging infrastructure, the configuration options are the same.

Environment variables

The following environment variables control some aspects of the ROS 2 loggers. For each of the environment settings, note that this is a process-wide setting, so it applies to all nodes in that process.

- **ROS_LOG_DIR** - Control the logging directory that is used for writing logging messages to disk (if that is enabled). If non-empty, use the exact directory as specified in this variable. If empty, use the contents of the **ROS_HOME** environment variable to construct a path of the form \$ROS_HOME/.log. In all cases, the ~ character is expanded to the user’s HOME directory.

- **ROS_HOME** - Control the home directory that is used for various ROS files, including logging and config files. In the context of logging, this variable is used to construct a path to a directory for log files. If non-empty, use the contents of this variable for the **ROS_HOME** path. In all cases, the ~ character is expanded to the user’s HOME directory.

- **RCUTILS_LOGGING_USE_STDOUT** - Control what stream output messages go to. If this is unset or 0, use stderr. If this is 1, use stdout.

- **RCUTILS_LOGGING_BUFFERED_STREAM** - Control whether the logging stream (as configured in **RCUTILS_LOGGING_USE_STDOUT**) should be line buffered or unbuffered. If this is unset, use the default of the stream (generally line buffered for stdout, and unbuffered for stderr). If this is 0, force the stream to be unbuffered. If this is 1, force the stream to be line buffered.

- **RCUTILS_COLORIZED_OUTPUT** - Control whether colors are used when outputting messages. If unset, automatically determine based on the platform and whether the console is a TTY. If 0, force disable using colors for output. If 1, force enable using colors for output.

- **RCUTILS_CONSOLE_OUTPUT_FORMAT** - Control the fields that are output for each log message. The available fields are:
  - **{severity}** - The severity level.
  - **{name}** - The name of the logger (may be empty).
  - **{message}** - The log message (may be empty).
  - **{function_name}** - The function name this was called from (may be empty).
  - **{file_name}** - The file name this was called from (may be empty).
  - **{time}** - The time in seconds since the epoch.
  - **{time_as_nanoseconds}** - The time in nanoseconds since the epoch.
  - **{line_number}** - The line number this was called from (may be empty).

If no format is given, a default of **[severity] [time] [name]: {message}** is used.
Node creation

When initializing a ROS 2 node, it is possible to control some aspects of the behavior via node options. Since these are per-node options, they can be set differently for different nodes even when the nodes are composed into a single process.

- **log_levels** - The log level to use for a component within this particular node. This can be set with the following: `ros2 run demo_nodes_cpp talker --ros-args --log-level talker:=DEBUG`
- **external_log_config_file** - The external file to use to configure the backend logger. If it is NULL, the default configuration will be used. Note that the format of this file is backend-specific (and is currently unimplemented for the default backend logger of spdlog). This can be set with the following: `ros2 run demo_nodes_cpp talker --ros-args --log-config-file log-config.txt`
- **log_stdout_disabled** - Whether to disable writing log messages to the console. This can be done with the following: `ros2 run demo_nodes_cpp talker --ros-args --disable-stdout-logs`
- **log_rosout_disabled** - Whether to disable writing log messages out to /rosout. This can significantly save on network bandwidth, but external observers will not be able to monitor logging. This can be done with the following: `ros2 run demo_nodes_cpp talker --ros-args --disable-rosout-logs`
- **log_ext_lib_disabled** - Whether to completely disable the use of an external logger. This may be faster in some cases, but means that logs will not be written to disk. This can be done with the following: `ros2 run demo_nodes_cpp talker --ros-args --disable-external-lib-logs`

Logging subsystem design

The image below shows the five main pieces to the logging subsystem and how they interact.

**rcutils**

rcutils has a logging implementation that can format log messages according to a certain format (see Configuration above), and output those log messages to a console. rcutils implements a complete logging solution, but allows higher-level components to insert themselves into the logging infrastructure in a dependency-injection model. This will become more evident when we talk about the rcl layer below.

Note that this is a **per-process** logging implementation, so anything that is configured at this level will affect the entire process, not just individual nodes.

**rcl_logging_spdlog**

rcl_logging_spdlog implements the rcl_logging_interface API, and thus provides external logging services to the rcl layer. In particular, the rcl_logging_spdlog implementation takes formatted log messages and writes them out to log files on disk using the spdlog library, typically within ~/.ros/log (though this is configurable; see Configuration above).
### rclpy
+ End-user logging calls, e.g.
  node.get_logger().info()

### rclcpp
+ End-user logging calls, e.g.
  RCLCPP_INFO

### rcl
+ Initialization of rutils logging
  (console logging)
+ Initialization of rcl_logging (logging
to disk)
+ Creation of /rosout publisher
  (logging to network)
+ Callback for each logging message

### rutils
+ Complete console logging
  implementation

### rcl_logging
+ Initialization of disk logging
+ Interface between disk logging and
  rcl
The logging subsystem in rcl uses rcutils and rcl_logging_spdlog to provide the bulk of the ROS 2 logging services. When log messages come in, rcl decides where to send them. There are 3 main places that logging messages can be delivered; an individual node may have any combination of them enabled:

- To the console via the rcutils layer
- To disk via the rcl_logging_spdlog layer
- To the /rosout topic on the ROS 2 network via the RMW layer

This is the main ROS 2 C++ API which sits atop the rcl API. In the context of logging, rclcpp provides the RCLCPP_ logging macros; see APIs above for a complete list. When one of the RCLCPP_ macros runs, it checks the current severity level of the node against the severity level of the macro. If the severity level of the macro is greater than or equal to the node severity level, the message will be formatted and output to all of the places that are currently configured. Note that rclcpp uses a global mutex for log calls, so all logging calls within the same process end up being single-threaded.

This is the main ROS 2 Python API which sits atop the rcl API. In the context of logging, rclpy provides the logger. debug-style functions; see APIs above for a complete list. When one of the logger.debug functions runs, it checks the current severity level of the node against the severity level of the macro. If the severity level of the macro is greater than or equal to the node severity level, the message will be formatted and output to all of the places that are currently configured.

**Logging usage**

**C++**

- See the rclcpp logging demo for some simple examples.
- See the logging demo for example usage.
- See the rclcpp documentation for an extensive list of functionality.

**Python**

- See the rclpy examples for example usage of a node’s logger.
- See the rclpy tests for example usage of keyword arguments (e.g. skip_first, once).
Quality of Service settings

Overview

ROS 2 offers a rich variety of Quality of Service (QoS) policies that allow you to tune communication between nodes. With the right set of Quality of Service policies, ROS 2 can be as reliable as TCP or as best-effort as UDP, with many, many possible states in between. Unlike ROS 1, which primarily only supported TCP, ROS 2 benefits from the flexibility of the underlying DDS transport in environments with lossy wireless networks where a “best effort” policy would be more suitable, or in real-time computing systems where the right Quality of Service profile is needed to meet deadlines.

A set of QoS “policies” combine to form a QoS “profile”. Given the complexity of choosing the correct QoS policies for a given scenario, ROS 2 provides a set of predefined QoS profiles for common use cases (e.g. sensor data). At the same time, developers are given the flexibility to control specific policies of the QoS profiles.

QoS profiles can be specified for publishers, subscriptions, service servers and clients. A QoS profile can be applied independently to each instance of the aforementioned entities, but if different profiles are used, it is possible that they will be incompatible, preventing the delivery of messages.

QoS policies

The base QoS profile currently includes settings for the following policies:

- **History**
  - *Keep last*: only store up to N samples, configurable via the queue depth option.
  - *Keep all*: store all samples, subject to the configured resource limits of the underlying middleware.

- **Depth**
  - *Queue size*: only honored if the “history” policy was set to “keep last”.

- **Reliability**
  - *Best effort*: attempt to deliver samples, but may lose them if the network is not robust.
  - *Reliable*: guarantee that samples are delivered, may retry multiple times.

- **Durability**
- **Transient local**: the publisher becomes responsible for persisting samples for “late-joining” subscriptions.
- **Volatile**: no attempt is made to persist samples.

- **Deadline**
  - **Duration**: the expected maximum amount of time between subsequent messages being published to a topic

- **Lifespan**
  - **Duration**: the maximum amount of time between the publishing and the reception of a message without the message being considered stale or expired (expired messages are silently dropped and are effectively never received).

- **Liveliness**
  - **Automatic**: the system will consider all of the node's publishers to be alive for another “lease duration” when any one of its publishers has published a message.
  - **Manual by topic**: the system will consider the publisher to be alive for another “lease duration” if it manually asserts that it is still alive (via a call to the publisher API).

- **Lease Duration**
  - **Duration**: the maximum period of time a publisher has to indicate that it is alive before the system considers it to have lost liveness (losing liveness could be an indication of a failure).

For each of the policies that is not a duration, there is also the option of “system default”, which uses the default of the underlying middleware. For each of the policies that is a duration, there also exists a “default” option that means the duration is unspecified, which the underlying middleware will usually interpret as an infinitely long duration.

**Comparison to ROS 1**

The “history” and “depth” policies in ROS 2 combine to provide functionality akin to the queue size in ROS 1. The “reliability” policy in ROS 2 is akin to the use of either UDPROS (only in roscpp) for “best effort”, or TCPROS (ROS 1 default) for “reliable”. Note however that even the reliable policy in ROS 2 is implemented using UDP, which allows for multicasting if appropriate.

The “durability” policy “transient local”, combined with any depth, provides functionality similar to that of “latching” publishers. The remaining policies in ROS 2 are not akin to anything that is available in ROS 1, meaning that ROS 2 is more featureful than ROS 1 in this respect. It is possible that in the future, even more QoS policies will be available in ROS 2.

**QoS profiles**

Profiles allow developers to focus on their applications without worrying about every QoS setting possible. A QoS profile defines a set of policies that are expected to go well together for a particular use case.

The currently defined QoS profiles are:

- **Default QoS settings for publishers and subscriptions**
  
  In order to make the transition from ROS 1 to ROS 2 easier, exercising a similar network behavior is desirable. By default, publishers and subscriptions in ROS 2 have “keep last” for history with a queue size of 10, “reliable” for reliability, “volatile” for durability, and “system default” for liveness. Deadline, lifespan, and lease durations are also all set to “default”.
• Services

In the same vein as publishers and subscriptions, services are reliable. It is especially important for services to use volatile durability, as otherwise service servers that re-start may receive outdated requests. While the client is protected from receiving multiple responses, the server is not protected from side-effects of receiving the outdated requests.

• Sensor data

For sensor data, in most cases it’s more important to receive readings in a timely fashion, rather than ensuring that all of them arrive. That is, developers want the latest samples as soon as they are captured, at the expense of maybe losing some. For that reason the sensor data profile uses best effort reliability and a smaller queue size.

• Parameters

Parameters in ROS 2 are based on services, and as such have a similar profile. The difference is that parameters use a much larger queue depth so that requests do not get lost when, for example, the parameter client is unable to reach the parameter service server.

• System default

This uses the RMW implementation’s default values for all of the policies. Different RMW implementations may have different defaults.

Click here for the specific policies in use for the above profiles. The settings in these profiles are subject to further tweaks, based on the feedback from the community.

QoS compatibilities

Note: This section refers to publishers and subscriptions but the content applies to service servers and clients in the same manner.

QoS profiles may be configured for publishers and subscriptions independently. A connection between a publisher and a subscription is only made if the pair has compatible QoS profiles.

QoS profile compatibility is determined based on a “Request vs Offered” model. Subscriptions request a QoS profile that is the “minimum quality” that it is willing to accept, and publishers offer a QoS profile that is the “maximum quality” that it is able to provide. Connections are only made if every policy of the requested QoS profile is not more stringent than that of the offered QoS profile. Multiple subscriptions can be connected to a single publisher simultaneously even if their requested QoS profiles are different. The compatibility between a publisher and a subscription is unaffected by the presence of other publishers and subscriptions.

The following tables show the compatibility of the different policy settings and the result:

Compatibility of reliability QoS policies:

<table>
<thead>
<tr>
<th>Publisher</th>
<th>Subscription</th>
<th>Compatible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best effort</td>
<td>Best effort</td>
<td>Yes</td>
</tr>
<tr>
<td>Best effort</td>
<td>Reliable</td>
<td>No</td>
</tr>
<tr>
<td>Reliable</td>
<td>Best effort</td>
<td>Yes</td>
</tr>
<tr>
<td>Reliable</td>
<td>Reliable</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Compatibility of durability QoS policies:
Compatibility of deadline QoS policies:

Assume \( x \) and \( y \) are arbitrary valid duration values.

<table>
<thead>
<tr>
<th>Publisher</th>
<th>Subscription</th>
<th>Compatible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>Default</td>
<td>Yes</td>
</tr>
<tr>
<td>Default</td>
<td>( x )</td>
<td>No</td>
</tr>
<tr>
<td>( x )</td>
<td>Default</td>
<td>Yes</td>
</tr>
<tr>
<td>( x )</td>
<td>( x )</td>
<td>Yes</td>
</tr>
<tr>
<td>( x )</td>
<td>( y ) (where ( y &gt; x ))</td>
<td>Yes</td>
</tr>
<tr>
<td>( x )</td>
<td>( y ) (where ( y &lt; x ))</td>
<td>No</td>
</tr>
</tbody>
</table>

Compatibility of liveliness QoS policies:

<table>
<thead>
<tr>
<th>Publisher</th>
<th>Subscription</th>
<th>Compatible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic</td>
<td>Automatic</td>
<td>Yes</td>
</tr>
<tr>
<td>Automatic</td>
<td>Manual by topic</td>
<td>No</td>
</tr>
<tr>
<td>Manual by topic</td>
<td>Automatic</td>
<td>Yes</td>
</tr>
<tr>
<td>Manual by topic</td>
<td>Manual by topic</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Compatibility of lease duration QoS policies:

Assume \( x \) and \( y \) are arbitrary valid duration values.

<table>
<thead>
<tr>
<th>Publisher</th>
<th>Subscription</th>
<th>Compatible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>Default</td>
<td>Yes</td>
</tr>
<tr>
<td>Default</td>
<td>( x )</td>
<td>No</td>
</tr>
<tr>
<td>( x )</td>
<td>Default</td>
<td>Yes</td>
</tr>
<tr>
<td>( x )</td>
<td>( x )</td>
<td>Yes</td>
</tr>
<tr>
<td>( x )</td>
<td>( y ) (where ( y &gt; x ))</td>
<td>Yes</td>
</tr>
<tr>
<td>( x )</td>
<td>( y ) (where ( y &lt; x ))</td>
<td>No</td>
</tr>
</tbody>
</table>

In order for a connection to be made, all of the policies that affect compatibility must be compatible. For example, even if a requested and offered QoS profile pair has compatible reliability QoS policies, but they have incompatible durability QoS policies, a connection will still not be made.

When connections are not made, no messages will be passed between the publisher and subscription. There are mechanisms to detect this situation, which will be covered in a later section.
Comparison to ROS 1

Historically in ROS 1, any publisher and subscriber with the same message type on the same topic would be connected. The possibility of incompatible requested and offered QoS profiles is something new to be aware of when using ROS 2.

QoS events

Some QoS policies have possible events related to them. Developers may provide each publisher and subscription with callback functions that are triggered by these QoS events and handle them in a way they see fit, similar to how messages received on a topic are handled.

Developers may subscribe to the following QoS events that are associated with a publisher:

- Offered deadline missed
  The publisher has not published a message within the expected duration that was set out by the deadline QoS policy.
- Liveliness lost
  The publisher has failed to indicate its liveliness within the lease duration.
- Offered incompatible QoS
  The publisher has encountered a subscription on the same topic that is requesting a QoS profile that the offered QoS profile cannot satisfy, resulting in no connection between the publisher and that subscription.

Developers may subscribe to the following QoS events that are associated with a subscription:

- Requested deadline missed
  The subscription has not received a message within the expected duration that was set out by the deadline QoS policy.
- Liveliness changed
  The subscription has noticed that one or more publishers on the subscribed topic has failed to indicate their liveliness within the lease duration.
- Requested incompatible QoS
  The subscription has encountered a publisher on the same topic that is offering a QoS profile that does not satisfy the requested QoS profile, resulting in no connection between the subscription and that publisher.

Matched events

In addition to QoS events, matched events can be generated when any publisher and subscription establishes or drops the connection between them. Developers may provide each publisher and subscription with callback functions that are triggered by matched events and handle them in a way they see fit, similar to how messages received on a topic are handled.

Developers can subscribe to this event with a publisher or a subscription.

- publisher: this event happens when it finds a subscription which matches the topic and has compatible QoS or a connected subscription is disconnected
- subscription: this event happens when it finds a publisher which matches the topic and has compatible QoS or a connected publisher is disconnected

There are demos showing how to use the event:
Executors

Table of Contents

- Overview
- Basic use
- Types of Executors
- Callback groups
- Scheduling semantics
- Outlook
- Further information

Overview

Execution management in ROS 2 is handled by Executors. An Executor uses one or more threads of the underlying operating system to invoke the callbacks of subscriptions, timers, service servers, action servers, etc. on incoming messages and events. The explicit Executor class (in executor.hpp in rclcpp, in executors.py in rclpy, or in executor.h in rclc) provides more control over execution management than the spin mechanism in ROS 1, although the basic API is very similar.

In the following, we focus on the C++ Client Library rclcpp.

Basic use

In the simplest case, the main thread is used for processing the incoming messages and events of a Node by calling rclcpp::spin(..) as follows:

```cpp
int main(int argc, char* argv[])
{
    // Some initialization.
    rclcpp::init(argc, argv);
    ...

    // Instantiate a node.
    rclcpp::Node::SharedPtr node = ...

    // Run the executor.
    rclcpp::spin(node);

    // Shutdown and exit.
    return 0;
}
```
The call to `spin(node)` basically expands to an instantiation and invocation of the Single-Threaded Executor, which is the simplest Executor:

```cpp
rclcpp::executors::SingleThreadedExecutor executor;
executor.add_node(node);
executor.spin();
```

By invoking `spin()` of the Executor instance, the current thread starts querying the rcl and middleware layers for incoming messages and other events and calls the corresponding callback functions until the node shuts down. In order not to counteract the QoS settings of the middleware, an incoming message is not stored in a queue on the Client Library layer but kept in the middleware until it is taken for processing by a callback function. (This is a crucial difference to ROS 1.) A `wait set` is used to inform the Executor about available messages on the middleware layer, with one binary flag per queue. The `wait set` is also used to detect when timers expire.

The Single-Threaded Executor is also used by the container process for *components*, i.e. in all cases where nodes are created and executed without an explicit main function.

### Types of Executors

Currently, rclcpp provides three Executor types, derived from a shared parent class:

```
Executor
  / \    \   /
SingleThreadedExecutor  MultiThreadedExecutor  StaticSingleThreadedExecutor
```

The *Multi-Threaded Executor* creates a configurable number of threads to allow for processing multiple messages or events in parallel. The *Static Single-Threaded Executor* optimizes the runtime costs for scanning the structure of a node in terms of subscriptions, timers, service servers, action servers, etc. It performs this scan only once when the node is added, while the other two executors regularly scan for such changes. Therefore, the Static Single-Threaded Executor should be used only with nodes that create all subscriptions, timers, etc. during initialization.

All three executors can be used with multiple nodes by calling `add_node( )` for each node.
In the above example, the one thread of a Static Single-Threaded Executor is used to serve three nodes together. In case of a Multi-Threaded Executor, the actual parallelism depends on the callback groups.

**Callback groups**

ROS 2 allows organizing the callbacks of a node in groups. In rclcpp, such a **callback group** can be created by the `create_callback_group` function of the Node class. In rclpy, the same is done by calling the constructor of the specific callback group type. The callback group must be stored throughout execution of the node (eg. as a class member), or otherwise the executor won’t be able to trigger the callbacks. Then, this callback group can be specified when creating a subscription, timer, etc. - for example by the subscription options:

**C++**

```cpp
my_callback_group = create_callback_group(rclcpp::CallbackGroupType::MutuallyExclusive);
```

```cpp
rclcpp::SubscriptionOptions options;
options.callback_group = my_callback_group;
```

```cpp
my_subscription = create_subscription<Int32>("/topic", rclcpp::SensorDataQoS(),
                                          callback, options);
```

**Python**

```python
my_callback_group = MutuallyExclusiveCallbackGroup()
my_subscription = self.create_subscription(Int32, "/topic", self.callback, qos_profile=1,
                                         callback_group=my_callback_group)
```

All subscriptions, timers, etc. that are created without the indication of a callback group are assigned to the **default callback group**. The default callback group can be queried via `NodeBaseInterface::get_default_callback_group()` in rclcpp and by `Node.default_callback_group` in rclpy.

There are two types of callback groups, where the type has to be specified at instantiation time:

- **Mutually exclusive**: Callbacks of this group must not be executed in parallel.
- **Reentrant**: Callbacks of this group may be executed in parallel.

Callbacks of different callback groups may always be executed in parallel. The Multi-Threaded Executor uses its threads as a pool to process as many callbacks as possible in parallel according to these conditions. For tips on how to use callback groups efficiently, see [Using Callback Groups](#using-callback-groups).

The Executor base class in rclcpp also has the function `add_callback_group()`, which allows distributing callback groups to different Executors. By configuring the underlying threads using the operating system scheduler, specific callbacks can be prioritized over other callbacks. For example, the subscriptions and timers of a control loop can be
prioritized over all other subscriptions and standard services of a node. The examples_rcfcpp_cbg_executor package
provides a demo of this mechanism.

**Scheduling semantics**

If the processing time of the callbacks is shorter than the period with which messages and events occur, the Executor
basically processes them in FIFO order. However, if the processing time of some callbacks is longer, messages and
events will be queued on the lower layers of the stack. The wait set mechanism reports only very little information
about these queues to the Executor. In detail, it only reports whether there are any messages for a certain topic or not.
The Executor uses this information to process the messages (including services and actions) in a round-robin fashion
- but not in FIFO order. The following flow diagram visualizes this scheduling semantics.

![Flow diagram](image)

This semantics was first described in a paper by Casini et al. at ECRTS 2019. (Note: The paper also explains that timer
events are prioritized over all other messages. This prioritization was removed in Eloquent.)

**Outlook**

While the three Executors of rclcpp work well for most applications, there are some issues that make them not suit-
able for real-time applications, which require well-defined execution times, determinism, and custom control over the
execution order. Here is a summary of some of these issues:

1. Complex and mixed scheduling semantics. Ideally you want well defined scheduling semantics to perform a
formal timing analysis.
2. Callbacks may suffer from priority inversion. Higher priority callbacks may be blocked by lower priority call-
backs.
3. No explicit control over the callbacks execution order.
4. No built-in control over triggering for specific topics.
Additionally, the executor overhead in terms of CPU and memory usage is considerable. The Static Single-Threaded Executor reduces this overhead greatly but it might not be enough for some applications.

These issues have been partially addressed by the following developments:

- **rclcpp WaitSet**: The `WaitSet` class of rclcpp allows waiting directly on subscriptions, timers, service servers, action servers, etc. instead of using an Executor. It can be used to implement deterministic, user-defined processing sequences, possibly processing multiple messages from different subscriptions together. The examples_rclcpp_wait_set package provides several examples for the use of this user-level wait set mechanism.

- **rclc Executor**: This Executor from the C Client Library rclc, developed for micro-ROS, gives the user fine-grained control over the execution order of callbacks and allows for custom trigger conditions to activate callbacks. Furthermore, it implements ideas of the Logical Execution Time (LET) semantics.

**Further information**


**Topic statistics**

<table>
<thead>
<tr>
<th>Table of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
</tr>
<tr>
<td>How statistics are calculated</td>
</tr>
<tr>
<td>Types of statistics calculated</td>
</tr>
<tr>
<td>Behavior</td>
</tr>
<tr>
<td>Comparison to ROS 1</td>
</tr>
<tr>
<td>Support</td>
</tr>
</tbody>
</table>

**Overview**

ROS 2 provides integrated measurement of statistics for messages received by any subscription. Allowing a user to collect subscription statistics enables them to characterize the performance of their system or aid in diagnosis of any present issues.

The measurements provided are the received message age and received message period. For each measurement the statistics provided are the average, maximum, minimum, standard deviation, and sample count. These statistics are calculated in a moving window.
How statistics are calculated

Each statistic set is calculated in constant time and constant memory by using the utilities implemented in the libstatistics_collector package. When a new message is received by a subscription, this is a new sample for calculation in the current measurement window. The average calculated is simply a moving average. The maximum, minimum, and sample count are updated upon receipt of each new sample, whereas the standard deviation is calculated using Welford’s online algorithm.

Types of statistics calculated

- Received message period
  - Units: milliseconds
  - Uses the system clock to measure the period between received messages
- Received message age
  - Units: milliseconds
  - Requires a message to have a timestamp populated in the header field in order to calculate the age of the message as sent from a publisher

Behavior

By default, Topic Statistics measurements are not enabled. After enabling this feature for a specific node via the subscription configuration options, both received message age and received message period measurements are enabled for that specific subscription.

The data is published as a statistics_msg/msg/MetricsMessage at a configurable period (default 1 second) to a configurable topic (default /statistics). Note that the publishing period also serves as the sample collection window period.

Since received message period requires a message timestamp in a header field, empty data is published. That is, all statistics values are NaN if no timestamp is found. Publishing NaN values instead of not publishing at all avoids the absence of a signal problem and is meant to explicitly show that a measurement could not be made.

The first sample of each window for the received message period statistic does not yield a measurement. This is because calculating this statistic requires knowing the time the previous message arrived, so subsequent samples in the window yield measurements.

Comparison to ROS 1

Similar to ROS 1 Topic Statistics, both message age and message period are calculated, albeit from the subscription side. Other ROS 1 metrics, e.g., the number of dropped messages or traffic volume, are currently not provided.
Support

This feature is currently supported in ROS 2 Foxy for C++ only (rclcpp). Future work and improvements, such as Python support, can be found here.

Overview and usage of RQt

Table of Contents

- Overview
- System setup
  - Installing From Debian
  - Building From Source
- RQt Components Structure
- Advantage of RQt framework
- Further Reading

Overview

RQt is a graphical user interface framework that implements various tools and interfaces in the form of plugins. One can run all the existing GUI tools as dockable windows within RQt. The tools can still run in a traditional standalone method, but RQt makes it easier to manage all the various windows in a single screen layout.

You can run any RQt tools/plugins easily by:

```
rqt
```

This GUI allows you to choose any available plugins on your system. You can also run plugins in standalone windows. For example, RQt Python Console:

```
ros2 run rqt_py_console rqt_py_console
```

Users can create their own plugins for RQt with either Python or C++. To see what RQt plugins are available for your system, run:

```
ros2 pkg list
```

And then look for packages that start with `rqt_`. 
System setup

Installing From Debian

```bash
sudo apt install ros-iron-rqt*
```

Building From Source

See *Building RQt from Source*.

RQt Components Structure

RQt consists of two metapackages:

- `rqt` - core infrastucture modules.
- `rqt_common_plugins` - Commonly useful debugging tools.

Advantage of RQt framework

Compared to building your own GUIs from scratch:

- Standardized common procedures for GUI (start-shutdown hook, restore previous states).
- Multiple widgets can be docked in a single window.
- Easily turn your existing Qt widgets into RQt plugins.
- Expect support at ROS Answers (ROS community website for the questions).

From system architecture’s perspective:

- Support multi-platform (basically wherever QT and ROS run) and multi-language (Python, C++).
- Manageable lifecycle: RQt plugins using a common API makes maintainance & reuse easier.

Further Reading

- ROS 2 Discourse [announcement of porting to ROS 2](#)
- RQt for ROS 1 [documentation](#)
- Brief overview of RQt (from a Willow Garage intern blog post)
Composition

Table of Contents

• ROS 1 - Nodes vs. Nodelets
• ROS 2 - Unified API
• Writing a Component
• Using Components
• Practical application

ROS 1 - Nodes vs. Nodelets

In ROS 1 you can write your code either as a ROS node or as a ROS nodelet. ROS 1 nodes are compiled into executables. ROS 1 nodelets on the other hand are compiled into a shared library which is then loaded at runtime by a container process.

ROS 2 - Unified API

In ROS 2 the recommended way of writing your code is similar to a nodelet - we call it a Component. This makes it easy to add common concepts to existing code, like a life cycle. Having different APIs, which was the biggest drawback in ROS 1, is avoided in ROS 2 since both approaches use the same API.

Note: It is still possible to use the node-like style of “writing your own main” but for the common case it is not recommended.

By making the process layout a deploy-time decision the user can choose between:

• running multiple nodes in separate processes with the benefits of process/fault isolation as well as easier debugging of individual nodes and
• running multiple nodes in a single process with the lower overhead and optionally more efficient communication (see Intra Process Communication).

Additionally ros2 launch can be used to automate these actions through specialized launch actions.

Writing a Component

Since a component is only built into a shared library, it doesn’t have a main function (see Talker source code). A component is commonly a subclass of rclcpp::Node. Since it is not in control of the thread, it shouldn’t perform any long running or blocking tasks in its constructor. Instead, it can use timers to get periodic notifications. Additionally, it can create publishers, subscriptions, servers, and clients.

An important aspect of making such a class a component is that the class registers itself using macros from the package rclcpp_components (see the last line in the source code). This makes the component discoverable when its library is being loaded into a running process - it acts as kind of an entry point.

Additionally, once a component is created, it must be registered with the index to be discoverable by the tooling.
add_library(talker_component SHARED src/talker_component.cpp)

rclcpp_components_register_nodes(talker_component "composition::Talker")

# To register multiple components in the same shared library, use multiple calls
# rclcpp_components_register_nodes(talker_component "composition::Talker2")

Note: In order for the component_container to be able to find desired components, it must be executed or launched from a shell that has sourced the corresponding workspace.

Using Components

The composition package contains a couple of different approaches on how to use components. The three most common ones are:

1. Start a (generic container process) and call the ROS service load_node offered by the container. The ROS service will then load the component specified by the passed package name and library name and start executing it within the running process. Instead of calling the ROS service programmatically you can also use a command line tool to invoke the ROS service with the passed command line arguments.

2. Create a custom executable containing multiple nodes which are known at compile time. This approach requires that each component has a header file (which is not strictly needed for the first case).

3. Create a launch file and use ros2 launch to create a container process with multiple components loaded.

Practical application

Try the Composition demos.

Cross-compilation

Table of Contents

- Overview
- How does it work?
- Cross-compiling ROS 2

Overview

Open Robotics provides pre-built ROS 2 packages for multiple platforms, but a number of developers still rely on cross-compilation for various reasons such as:

- The development machine does not match the target system.
- Tuning the build for specific core architecture (e.g. setting -mcpu=cortex-a53 -mfpu=neon-fp-armv8 when building for Raspberry Pi3).
- Targeting a file system other than the ones supported by the pre-built images released by Open Robotics.
How does it work?

Cross-compiling simple software (e.g. no dependencies on external libraries) is relatively simple and only requiring a cross-compiler toolchain to be used instead of the native toolchain.

There are a number of factors which make this process more complex:

- The software being built must support the target architecture. Architecture specific code must be properly isolated and enabled during the build according to the target architecture. Examples include assembly code.
- All dependencies (e.g. libraries) must be present, either as pre-built or cross-compiled packages, before the target software using them is cross-compiled.
- When building software stacks (as opposed to standalone software) using build tools (e.g. colcon), it is expected that the build tool provides a mechanism to allow the developer to enable cross-compilation on the underlying build system used by each piece of software in the stack.

Cross-compiling ROS 2

The ROS 2 cross-compile tool is under shared ownership of Open Robotics and ROS Tooling Working Group. It is a Python script that compiles ROS 2 source files for supported target architectures using an emulator in a docker container. Detailed design of the tool can be found on ROS 2 design. Instructions to use the tool are in the cross_compile package.

If you are using an older version, please follow the cross-compilation guide.

ROS 2 Security

Table of Contents

- Overview
- The Security Enclave
- Security Files
  - Enclave Identity
  - Enclave Permissions
  - Private Keys
- Security Environment Variables
- Learn More

Overview

ROS 2 includes the ability to secure communications among nodes within the ROS 2 computational graph. Similar to discovery, security happens through the underlying ROS 2 middleware (provided it has support for the corresponding security plugins). No additional software installation is needed to enable security; however, the middleware requires configuration files for each ROS graph participant. These files enable encryption and authentication, and define policies both for individual nodes and for the overall ROS graph. ROS 2 also adds a master “on/off” switch to control security behavior.
ROS utilities can create the authoritative trust anchor for a ROS application, or an external certificate authority can be used.

Built-in ROS 2 security features enable control over communications throughout the ROS graph. This not only allows for encrypting data in transit between ROS domain participants, but also enables authentication of participants sending data, ensures the integrity of data being sent, and enables domain-wide access controls.

ROS 2 security services are provided by the underlying Data Distribution Service (DDS) which is used for communications between nodes. DDS vendors provide open source and commercial DDS implementations that work with ROS. However, in order to create a specification-compliant implementation of DDS, all vendors must include security plugins as outlined in the DDS Security Specification. ROS security features take advantage of these DDS security plugins to provide policy-based encryption, authentication and access control. DDS and ROS security is enabled through predefined configuration files and environment variables.

**The Security Enclave**

A security enclave encapsulates a single policy for protecting ROS communications. The enclave may set policy for multiple nodes, for an entire ROS graph, or any combination of protected ROS processes and devices. Security enclaves can be flexibly mapped to processes, users, or devices at deployment. Adjusting this default behavior becomes important for optimizing communications and for complex systems. See the ROS 2 Security Enclaves design document for additional details.

**Security Files**

A ROS 2 security enclave is established with six files as outlined by the DDS specification. Three of these files define an enclave’s identity, while three other files define the permissions to be granted to the enclave. All six files reside in a single directory, and nodes launched without a qualified enclave path use files in the default root level enclave.

**Enclave Identity**

The Identity Certificate Authority file `identity_ca.cert.pem` acts as the trust anchor used to identify participants. Each enclave also holds its unique identifying certificate in the file `cert.pem`, and the associated private key in the file `key.pem`. Because the `cert.pem` certificate has been signed by identity certificate, when a participant presents this certificate to other domain members, they are able to validate the participant’s identity using their own copy of the identity certificate. This valid certificate exchange allows the enclave to securely establish trusted communications with other participants. The enclave does not not share the `key.pem` private key, but only uses it for decryption and message signing.

**Enclave Permissions**

The Permissions Certificate Authority file `permissions_ca.cert.pem` serves as the trust anchor to grant permissions to security enclaves. This certificate is used to create the signed file `governance.p7s`, an XML document which defines domain-wide protection policies. Similarly the XML file `permissions.p7s` outlines permissions of this particular enclave and has been signed by the Permissions CA. Domain members use a copy of the permissions CA to validate these signed files and grant the requested access.

Although these two certificate authorities enable separate workflows for identity and permissions, often the same certificate serves as both the identity and the permissions authority.
Private Keys

The identity and permissions certificates also have associated private key files. Add new enclaves to the domain by signing their Certificate Signing Request (CSR) with the identity certificate’s private key. Similarly, grant permissions for a new enclave by signing a permissions XML document with the permission certificate’s private key.

Security Environment Variables

The environment variable ROS_SECURITY_ENABLE acts as the enclave’s master “on/off” switch for ROS 2 security features. Security has been turned off by default, so security features will not be enabled even when the proper security files are present. In order to enable ROS 2 security, set this environment variable to true (case sensitive).

Once security has been enabled, the environment variable ROS_SECURITY_STRATEGY defines how domain participants handle problems when launching participants. Security features depend on certificates and properly signed configuration files, yet by default, an improperly configured participant will still launch successfully but without security features. In order to enforce strict compliance with security settings and fail to launch non-compliant enclaves, set this environment variable to Enforce (case sensitive).

Additional security-related environment variables can be found in the ROS 2 DDS-Security Integration design document. These variables generally assist ROS in managing enclaves and locating the security files.

Learn More

For more information and hands-on exercises enabling ROS 2 communications security, see the Setting up security.

Tf2

Overview

tf2 is the transform library, which lets the user keep track of multiple coordinate frames over time. tf2 maintains the relationship between coordinate frames in a tree structure buffered in time and lets the user transform points, vectors, etc. between any two coordinate frames at any desired point in time.
Properties of tf2

A robotic system typically has many 3D coordinate frames that change over time, such as a world frame, base frame, gripper frame, head frame, etc. tf2 keeps track of all these frames over time, and allows you to ask questions like:

- Where was the head frame relative to the world frame 5 seconds ago?
- What is the pose of the object in my gripper relative to my base?
- What is the current pose of the base frame in the map frame?

tf2 can operate in a distributed system. This means all the information about the coordinate frames of a robot is available to all ROS 2 components on any computer in the system. tf2 can have every component in your distributed system build its own transform information database or have a central node that gathers and stores all transform information.
Tutorials

We created a set of tutorials that walks you through using tf2, step by step. You can get started on the introduction to tf2 tutorial. For a complete list of all tf2 and tf2-related tutorials check out the tutorials page.

There are essentially two main tasks that any user would use tf2 for, listening for transforms and broadcasting transforms.

If you want to use tf2 to transform between coordinate frames, your nodes will need to listen for transforms. What you will do is receive and buffer all coordinate frames that are broadcasted in the system, and query for specific transforms between frames. Check out the “Writing a listener” tutorial (Python) (C++) to learn more.

To extend the capabilities of a robot, you will need to start broadcasting transforms. Broadcasting transforms means to send out the relative pose of coordinate frames to the rest of the system. A system can have many broadcasters that each provide information about a different part of the robot. Check out the “Writing a broadcaster” tutorial (Python) (C++) to learn more.

In addition to that, tf2 can broadcast static transforms that do not change over time. This mainly saves storage and lookup time, but also reduces the publishing overhead. You should note that static transforms are published once and assumed to not change, so no history is stored. If you want to define static transforms in your tf2 tree, take a look at the “Writing a static broadcaster” (Python) (C++) tutorial.

You can also learn how to add fixed and dynamic frames to your tf2 tree in the “Adding a frame” (Python) (C++) tutorial.

Once you are finished with the basic tutorials, you can move on to learn about tf2 and time. The tf2 and time tutorial (Python) (C++) teaches the basic principles of tf2 and time. The advanced tutorial about tf2 and time (Python) (C++) teaches the principles of time traveling with tf2.

Paper

There is a paper on tf2 presented at TePRA 2013: tf: The transform library.

Advanced Concepts

These conceptual documents are intended for developers who plan to modify or contribute to the ROS 2 core.

The build system

The build system is what allows developers to build their ROS 2 code as needed. ROS 2 relies heavily on the division of code into packages, with each package containing a manifest file (package.xml). This manifest file contains essential
metadata about the package, including its dependencies on other packages. This manifest is required for the meta-build tool to function.

The ROS 2 build system consists of 3 major concepts.

**Build tool**

This is the software that controls the compilation and testing of a single package. In ROS 2 this is usually CMake for C++, and setuptools for Python, but other build tools are supported.

**Build helpers**

These are helper functions that hook into the build tool to developer experience. ROS 2 packages typically rely on the ament series of packages for this. ament consists of a few important repositories which are all in the GitHub organization.

**The ament_package package**

Located on GitHub at ament/ament_package, this repository contains a single ament Python package that provides various utilities for ament packages, e.g. templates for environment hooks.

All ament packages must contain a single package.xml file at the root of the package regardless of their underlying build system. The package.xml “manifest” file contains information that is required in order to process and operate on a package. This package information includes things like the package’s name, which is globally unique, and the package’s dependencies. The package.xml file also serves as the marker file which indicates the location of the package on the file system.

Parsing of the package.xml files is provided by catkin_pkg (as in ROS 1), while functionality to locate packages by searching the file system for these package.xml files is provided by build tools such as colcon.

package.xml Package manifest file which marks the root of a package and contains meta information about the package including its name, version, description, maintainer, license, dependencies, and more. The contents of the manifest are in machine readable XML format and the contents are described in the REPs 127 and 140, with the possibility of further modifications in future REPs.

So anytime some package is referred to as an ament package, it means that it is a single unit of software (source code, build files, tests, documentation, and other resources) which is described using a package.xml manifest file.

ament package Any package which contains a package.xml and follows the packaging guidelines of ament, regardless of the underlying build system.

Since the term ament package is build system agnostic, there can be different kinds of ament packages, e.g. ament CMake package, ament Python package, etc.

Here is a list of common package types that you might run into in this software stack:

- **CMake package** Any package containing a plain CMake project and a package.xml manifest file.
- **ament CMake package** A CMake package that also follows the ament packaging guidelines.
- **Python package** Any package containing a setuptools based Python project and a package.xml manifest file.
- **ament Python package** A Python package that also follows the ament packaging guidelines.
The ament_cmake repository

Located on GitHub at ament/ament_cmake, this repository contains many “ament CMake” and pure CMake packages which provide the infrastructure in CMake that is required to create “ament CMake” packages. In this context “ament CMake” packages means: ament packages that are built using CMake. So the packages in this repository provide the necessary CMake functions/macros and CMake Modules to facilitate creating more “ament CMake” (or ament_cmake) packages. Packages of this type are identified with the <build_type>ament_cmake</build_type> tag in the <export> tag of the package.xml file.

The packages in this repository are extremely modular, but there is a single “bottleneck” package called ament_cmake. Anyone can depend on the ament_cmake package to get all of the aggregate functionality of the packages in this repository. Here a list of the packages in the repository along with a short description:

- **ament_cmake**
  - aggregates all other packages in this repository, users need only to depend on this

- **ament_cmake_auto**
  - provides convenience CMake functions which automatically handle a lot of the tedious parts of writing a package's CMakeLists.txt file

- **ament_cmake_core**
  - provides all built-in core concepts for ament, e.g. environment hooks, resource indexing, symbolic linking install and others

- **ament_cmake_gmock**
  - adds convenience functions for making gmock based unit tests

- **ament_cmake_gtest**
  - adds convenience functions for making QTest based automated tests

- **ament_cmake_nose**
  - adds convenience functions for making nosetests based python automated tests

- **ament_cmake_python**
  - provides CMake functions for packages that contain Python code
  - see the ament_cmake_python user documentation

- **ament_cmake_test**
  - aggregates different kinds of tests, e.g. QTest and nosetests, under a single target using CTest

The ament_cmake_core package contains a lot of the CMake infrastructure that makes it possible to cleanly pass information between packages using conventional interfaces. This makes the packages have more decoupled build interfaces with other packages, promoting their reuse and encouraging conventions in the build systems of different packages. For instance, it provides a standard way to pass include directories, libraries, definitions, and dependencies between packages so that consumers of this information can access this information in a conventional way.

The ament_cmake_core package also provides features of the ament build system like symbolic link installation, which allows you to symbolically link files from either the source space or the build space into the install space rather than copying them. This allows you to install once and then edit non-generated resources like Python code and configuration files without having to rerun the install step for them to take effect. This feature essentially replaces the “devel space” from catkin because it has most of the advantages with few of the complications or drawbacks.

Another feature provided by ament_cmake_core is the package resource indexing which is a way for packages to indicate that they contain a resource of some type. The design of this feature makes it much more efficient to answer simple questions like what packages are in this prefix (e.g. /usr/local) because it only requires that you list the files...
in a single possible location under that prefix. You can read more about this feature in the design docs for the resource index.

Like catkin, ament_cmake_core also provides environment setup files and package specific environment hooks. The environment setup files, often named something like setup.bash, are a place for package developers to define changes to the environment that are needed to utilize their package. The developers are able to do this using an “environment hook” which is basically an arbitrary bit of shell code that can set or modify environment variables, define shell functions, setup auto-completion rules, etc... This feature is how, for example, ROS 1 set the ROS_DISTRRO environment variable without catkin knowing anything about the ROS distribution.

The ament_lint repository

Located on GitHub at ament/ament_lint, this repository provides several packages which provide linting and testing services in a convenient and consistent manner. Currently there are packages to support C++ style linting using uncrustify, static C++ code checks using cppcheck, checking for copyright in source code, Python style linting using pep8, and other things. The list of helper packages will likely grow in the future.

Meta-build tool

This is a piece of software that knows how to topologically order a group of packages, and build or test them in the correct dependency order. This software will call into the Build Tool to do the actual work of compiling, testing, and installing the package.

In ROS 2, the tool named colcon is used for this.

Internal ROS 2 interfaces

The internal ROS interfaces are public C APIs that are intended for use by developers who are creating client libraries or adding a new underlying middleware, but are not intended for use by typical ROS users. The ROS client libraries provide the user facing APIs that most ROS users are familiar with, and may come in a variety of programming languages.
Internal API Architecture Overview

There are two main internal interfaces:

- the ROS middleware interface (rmw API)
- the ROS client library interface (rcl API)

The rmw API is the interface between the ROS 2 software stack and the underlying middleware implementation. The underlying middleware used for ROS 2 is either a DDS or RTPS implementation, and is responsible for discovery, publish and subscribe mechanics, request-reply mechanics for services, and serialization of message types.

The rcl API is a slightly higher level API which is used to implement the client libraries and does not touch the middleware implementation directly, but rather does so through the ROS middleware interface (rmw API) abstraction.

As the diagram shows, these APIs are stacked such that the typical ROS user will use the client library API, e.g. rclcpp, to implement their code (executable or library). The implementation of the client libraries, e.g. rclcpp, use the rcl interface which provides access to the ROS graph and graph events. The rcl implementation in turn uses the rmw API to access the ROS graph. The purpose of the rcl implementation is to provide a common implementation for more complex ROS concepts and utilities that may be used by various client libraries, while remaining agnostic to the underlying middleware being used. The purpose of the rmw interface is to capture the absolute minimum middleware functionality needed to support ROS’s client libraries. Finally, the implementation of the rmw API is provided by a middleware implementation specific package, e.g. rmw_fastrtps_cpp, the library of which is compiled against vendor specific DDS interfaces and types.

In the diagram above there is also a box labeled ros_to_dds, and the purpose of this box is to represent a category of possible packages which allow the user to access DDS vendor specific objects and settings using the ROS equivalents.
One of the goals of this abstraction interface is to completely insulate the ROS user space code from the middleware being used, so that changing DDS vendors or even middleware technology has a minimal impact on the users code. However, we recognize that on occasion it is useful to reach into the implementation and manually adjust settings despite the consequences that might have. By requiring the use of one of these packages in order to access the underlying DDS vendor’s objects, we can avoid exposing vendor specific symbols and headers in the normal interface. It also makes it easy to see what code is potentially violating the vendor portability by inspecting the package’s dependencies to see if one of these ros_to_dds packages are being used.

Type Specific Interfaces

All along the way there are some parts of the APIs that are necessarily specific to the message types being exchanged, e.g. publishing a message or subscribing to a topic, and therefore require generated code for each message type. The following diagram layouts the path from user defined rosidl files, e.g. .msg files, to the type specific code used by the user and system to perform type specific functions:

![Diagram showing the flow from rosidl files to user facing code]

The right hand side of the diagram shows how the .msg files are passed directly to language specific code generators, e.g. rosidl_generator_cpp or rosidl_generator_py. These generators are responsible for creating the code that the user will include (or import) and use as the in-memory representation of the messages that were defined in the .msg files. For example, consider the message std_msgs/String, a user might use this file in C++ with the statement #include <std_msgs/msg/string.hpp>, or they might use the statement from std_msgs.msg import String in Python. These statements work because of the files generated by these language specific (but middleware agnostic) generator packages.

Separately, the .msg files are used to generate type support code for each type. In this context, type support means: meta data or functions that are specific to a given type and that are used by the system to perform particular tasks for the
given type. The type support for a given message might include things like a list of the names and types for each field in the message. It might also contain a reference to code that can perform particular tasks for that type, e.g. publish a message.

**Static Type Support**

When the type support references code to do particular functions for a specific message type, that code sometimes needs to do middleware specific work. For example, consider the type specific publish function, when using “vendor A” the function will need to call some of “vendor A”’s API, but when using “vendor B” it will need to call “vendor B”’s API. To allow for middleware vendor specific code, the user defined .msg files may result in the generation of vendor specific code. This vendor specific code is still hidden from the user through the type support abstraction, which is similar to how the “Private Implementation” (or Pimpl) pattern works.

**Static Type Support with DDS**

For middleware vendors based on DDS, and specifically those which generate code based on the OMG IDL files (.idl files), the user defined rosidl files (.msg files) are converted into equivalent OMG IDL files (.idl files). From these OMG IDL files, vendor specific code is created and then used within the type specific functions which are referenced by the type support for a given type. The above diagram shows this on the left hand side, where the .msg files are consumed by the rosidl_dds package to produce .idl files, and then those .idl files are given to language specific and DDS vendor specific type support generation packages.

For example, consider the Fast DDS implementation, which has a package called rosidl_typesupport_fastrtps_cpp. This package is responsible for generating code to handle things like converting a C++ message object into a serialized octet buffer to be written over the network. This code, while specific to Fast DDS, is still not exposed to the user because of the abstraction in the type support code.

**Dynamic Type Support**

Another way to implement type support is to have generic functions for things like publishing to a topic, rather than generating a version of the function for each message type. In order to accomplish this, this generic function needs some meta information about the message type being published, things like a list of field names and types in the order in which they appear in the message type. Then to publish a message, you call a generic publish function and pass a message to be published along with a structure which contains the necessary meta data about the message type. This is referred to as “dynamic” type support, as opposed to “static” type support which requires generated versions of a function for each type.

The above diagram shows the flow from user defined rosidl files to generated user facing code. It is very similar to the diagram for static type support, and differs only in how the type support is generated which is represented by the left hand side of the diagram. In dynamic type support the .msg files are converted directly into user facing code.

This code is also middleware agnostic, because it only contains meta information about the messages. The function to actually do the work, e.g. publishing to a topic, is generic to the message type and will make any necessary calls to the middleware specific APIs. Note that rather than dds vendor specific packages providing the type support code, which is the case in static type support, this method has middleware agnostic package for each language, e.g. rosidl_typesupport_introspection_c and rosidl_typesupport_introspection_cpp. The introspection part of the package name refers to the ability to introspect any message instance with the generated meta data for the message type. This is the fundamental capability that allows for generic implementations of functions like “publish to a topic”.

This approach has the advantage that all generated code is middleware agnostic, which means it can be reused for different middleware implementations, so long as they allow for dynamic type support. It also results in less generated code, which reduces compile time and code size.
However, dynamic type support requires that the underlying middleware support a similar form of dynamic type support. In the case of DDS the DDS-XTypes standard allows for publishing of messages using meta information rather than generated code. DDS-XTypes, or something like it, is required in the underlying middleware in order to support dynamic type support. Also, this approach to type support is normally slower than the static type support alternative. The type specific generated code in static type support can be written to be more efficient as it does not need to iterate over the message type's meta data to perform things like serialization.

The **rcl** repository

The ROS Client Library interface (**rcl** API) can be used by client libraries (e.g. **rcl**, **rclcpp**, **rclpy**, etc.) in order to avoid duplicating logic and features. By reusing the **rcl** API, client libraries can be smaller and more consistent with each other. Some parts of the client library are intentionally left out of the **rcl** API because the language idiomatic method should be used to implement those parts of the system. A good example of this is the execution model, which **rcl** does not address at all. Instead the client library should provide a language idiomatic solution like **pthreads** in C, **std::thread** in C++11, and **threading.Thread** in Python. Generally the **rcl** interface provides functions that are not specific to a language pattern and are not specific to a particular message type.

The **rcl** API is located in the **ros2/rcl** repository on GitHub and contains the interface as C headers. The **rcl** C implementation is provided by the **rcl** package in the same repository. This implementation avoids direct contact with the middleware by instead using the **rmw** and **rosidl** APIs.

For a complete definition of the **rcl** API, see the **rcl** docs.
The `rmw` repository

The ROS middleware interface (`rmw API`) is the minimal set of primitive middleware capabilities needed to build ROS on top. Providers of different middleware implementations must implement this interface in order to support the entire ROS stack on top. Currently all of the middleware implementations are for different DDS vendors.

The `rmw API` is located in the `ros2/rmw` repository. The `rmw package` contains the C headers which define the interface, the implementation of which is provided by the various packages of `rmw` implementations for different DDS vendors.

For a definition of the `rmw API`, see the `rmw docs`.

The `rosidl` repository

The `rosidl API` consists of a few message related static functions and types along with a definition of what code should be generated by messages in different languages. The generated message code specified in the `API` will be language specific, but may or may not reuse generated code for other languages. The generated message code specified in the `API` contains things like the message data structure, functions for construction, destruction, etc. The `API` will also implement a way to get the type support structure for the message type, which is used when publishing or subscribing to a topic of that message type.

There are several repositories that play a role in the `rosidl API` and implementation.

The `rosidl` repository, located on GitHub at `ros2/rosidl`, defines the message IDL syntax, i.e. syntax of `.msg` files, `.srv` files, etc., and contains packages for parsing the files, for providing CMake infrastructure to generate code from the messages, for generating implementation agnostic code (headers and source files), and for establishing the default set of generators. The repository contains these packages:

- `rosidl_cmake`: provides CMake functions and modules for generating code from `rosidl` files, e.g. `.msg` files, `.srv` files, etc.
- `rosidl_default_generators`: defines the list of default generators which ensures that they are installed as dependencies, but other injected generators can also be used.
- `rosidl_generator_c`: provides tools to generate C header files (.h) for `rosidl` files.
- `rosidl_generator_cpp`: provides tools to generate C++ header files (.hpp) for `rosidl` files.
- `rosidl_generator_py`: provides tools to generate Python modules for `rosidl` files.
- `rosidl_parser`: provides Python `API` for parsing `rosidl` files.

Generators for other languages, e.g. `rosidl_generator_java`, are hosted externally (in different repositories) but would use the same mechanism that the above generators use to “register” themselves as a `rosidl` generator.

In addition to the aforementioned packages for parsing and generating headers for the `rosidl` files, the `rosidl` repository also contains packages concerned with “type support” for the message types defined in the files. Type support refers to the ability to interpret and manipulate the information represented by ROS message instances of particular types (publishing the messages, for example). Type support can either be provided by code that is generated at compile time or it can be done programmatically based on the contents of the `rosidl` file, e.g. the `.msg` or `.srv` file, and the data received, by introspecting the data. In the case of the latter, where type support is done through runtime interpretation of the messages, the message code generated by ROS 2 can be agnostic to the `rmw` implementation. The packages that provide this type support through introspection of the data are:

- `rosidl_typesupport_introspection_c`: provides tools for generating C code for supporting `rosidl` message data types.
- `rosidl_typesupport_introspection_cpp`: provides tools for generating C++ code for supporting `rosidl` message data types.
In the case where type support is to be generated at compile time instead of being generated programmatically, a package specific to the rmw implementation will need to be used. This is because typically a particular rmw implementation will require data to be stored and manipulated in a manner that is specific to the DDS vendor in order for the DDS implementation to make use of it. See the Type Specific Interfaces section above for more details.

For more information on what exactly is in the rosidl API (static and generated) see this page:

The rcutils repository

ROS 2 C Utilities (rcutils) is a C API composed of macros, functions, and data structures used throughout the ROS 2 codebase. These are mainly used for error handling, commandline argument parsing, and logging which are not specific to the client or middleware layers and can be shared by both.

The rcutils API and implementation are located in the ros2/rcutils repository on GitHub which contains the interface as C headers.

For a complete definition of the rcutils API, see the rcutils docs.

ROS 2 middleware implementations

ROS middleware implementations are sets of packages that implement some of the internal ROS interfaces, e.g. the rmw, rcl, and rosidl APIs.

Common Packages for DDS Middleware Packages

All of the current ROS middleware implementations are based on full or partial DDS implementations. For example, there is a middleware implementation that uses RTI’s Connext DDS and an implementation which uses eProsima’s Fast DDS. Because of this, there are some shared packages amongst most DDS based middleware implementations.

In the ros2/rosidl_dds repository on GitHub, there is the following package:

- rosidl_generator_dds_idl: provides tools to generate DDS .idl files from rosidl files, e.g. .msg files, .srv files, etc.

The rosidl_generator_dds_idl package generates a DDS .idl file for each rosidl file, e.g. .msg file, defined by packages containing messages. Currently DDS based ROS middleware implementations make use of this generator’s output .idl files to generate pre-compiled type support that is vendor specific.
Structure of ROS Middleware Implementations

A ROS middleware implementation is typically made up of a few *packages* in a single repository:

- `<implementation_name>_cmake_module`: contains CMake Module for discovering and exposing required dependencies
- `rmw_<implementation_name>_<language>`: contains the implementation of the rmw API in a particular language, typically C++
- `rosidl_typesupport_<implementation_name>_<language>`: contains tools to generate static type support code for rosidl files, tailored to the implementation in a particular language, typically C or C++

The `<implementation_name>_cmake_module` package contains any CMake Modules and functions needed to find the supporting dependencies for the middleware implementation. For example, `rti_connext_dds_cmake_module` provides wrapper logic around the CMake Module shipped with RTI Connext DDS to make sure that all packages that depend on it will select the same installation of RTI Connext DDS. Similarly, `fastrtps_cmake_module` includes a CMake Module to find eProsima’s Fast DDS and `gurumdds_cmake_module` includes a CMake Module to find GurumNetworks GurumDDS. Not all implementations will have a package like this: for example, Eclipe’s Cyclone DDS already provides a CMake Module which is used directly by its RMW implementation without the need of additional wrappers.

The `rmw_<implementation_name>_<language>` package implements the rmw API in a particular language. The implementation itself can be C++, it just must expose the header’s symbols as `extern "C"` so that C applications can link against it.

The `rosidl_typesupport_<implementation_name>_<language>` package provides a generator which generates DDS code in a particular language. This is done using the .idl files generated by the `rosidl_generator_dds_idl` package and the DDS IDL code generator provided by the DDS vendor. It also generates code for converting ROS message structures to and from DDS message structures. This generator is also responsible for creating a shared library for the message package it is being used in, which is specific to the messages in the message package and to the DDS vendor being used.

As mentioned above, the `rosidl_typesupport_introspection_<language>` may be used instead of a vendor specific type support package if an rmw implementation supports runtime interpretation of messages. This ability to programmatically send and receive types over topics without generating code beforehand is achieved by supporting the DDS X-Types Dynamic Data standard. As such, rmw implementations may provide support for the X-Types standard, and/or provide a package for type support generated at compile time specific to their DDS implementation.

As an example of an rmw implementation repository, the Eclipse Cyclone DDS ROS middleware implementation is on GitHub at ros2/rmw_cyclonedds.

The rmw implementation for Fast DDS is on GitHub at ros2/rmw_fastrtps_cpp.

The rmw implementation for Connext DDS is on GitHub at ros2/rmw_connextdds.

The rmw implementation for GurumDDS is on GitHub at ros/rmw_gurumdds.

**Related Content**

*See the ROS 2 citations* for more explanation of concepts and citable resources.

For a brief video introduction to ROS 2, see this community contributed content:

- Getting started with ROS Part 1: Nodes, Parameters and Topics
- Getting started with ROS Part 2: Services and Actions
4.8.6 Contact

Support

Different types of questions or discussions correspond to different avenues of communication; check the descriptions below to ensure you choose the right method.

Need help troubleshooting your system? First, search ROS Answers to see if others have had similar issues, and if their solution works for you.

If not, ask a new question on ROS Answers. Make sure to add tags, at the very least the ros2 tag and the distro version you are running, e.g. iron. If your question is related to the documentation here, add a tag like docs, or more specifically, tutorials.

Contributing support

ROS 2 users come from a wide range of technical backgrounds, use a variety of different operating systems, and don’t necessarily have any prior experience with ROS (1 or 2). So, it’s important for users with any amount of experience to contribute support.

If you see an issue on ROS Answers that is similar to something you’ve run into yourself, please consider providing some pointers to what helped in your situation. Don’t worry if you aren’t sure if your response is correct. Simply say so, and other community members will jump in if necessary.

Issues

If you identify bugs, have suggestions for improvements, or a question specific to one package, you can open an issue on GitHub.

For example, if you are following the tutorials here and come across an instruction that doesn’t work on your system, you can open an issue in the ros2_documentation repo.

You can search for individual ROS 2 repositories on ROS 2’s GitHub.

Before opening an issue, check if other users have reported similar issues by searching across the ros2 and ament GitHub organizations: example search query.

Next, check ROS Answers to see if someone else has asked your question or reported your issue.

If it has not been reported, feel free to open an issue in the appropriate repository tracker. If it’s not clear which tracker to use for a particular issue, file it in the ros2/ros2 repository and we’ll have a look at it.

When filing an issue, please make sure to:

• Include enough information for another person to understand the issue.

Describe exactly what you were doing or are trying to do, and exactly what, if anything, went wrong. If following a tutorial or online instructions provide a link to the specific instructions.

• Use a descriptive headline or subject line. Bad: “rviz doesn’t work”. Good: “Rviz crashing looking for missing .so after latest apt update”

• Include information about the exact platform, software, versions, and environment relevant to the problem. This includes how you installed the software (from binaries or from source) and which ROS middleware/DDS vendor you are using (if you know it).

• Any warnings or errors. Cut and paste them directly from the terminal window to which they were printed. Please do not re-type or include a screenshot.

• In case of a bug consider providing a short, self contained, correct (compilable), example.
• When discussing any compiling/linking/installation issues, also provide the compiler version

As appropriate, also include your:

• ROS environment variables (env | grep ROS)
• Backtraces
• Relevant config files
• Graphics card model and driver version
• Ogre.log for rviz, if possible (run with rviz -l)
• Bag files and code samples that can reproduce the problem
• Gifs or movies to demonstrate the problem

Pull requests

When you feel comfortable enough to suggest a specific change directly to the code, you can submit a pull request. Pull requests are welcome for any of the ros2 repositories. See the Contributing page for more details and etiquette on how to contribute.

Discussion

To start a discussion with other ROS 2 community members, visit the official ROS Discourse. Content on the Discourse should be high-level; it’s not a place to get questions about code answered, but it would be suitable to start a conversation about best practices or improving standards.

Discussions about ROS 2 development and plans are happening on the “Next Generation ROS” Discourse category. Participating in these discussions is an important way to have a say on how different features of ROS 2 will work and be implemented.

The diverse community behind the ROS ecosystem is one of its greatest assets. We encourage all members of the ROS community to participate in these design discussions so that we can leverage the experience of community members, and keep the varied use cases of ROS in mind.

Etiquette

Assume ‘good faith’: It’s easy to mis-interpret the meaning or tone of comments on the internet. Assuming good faith gives the benefit of the doubt to those trying to help you, avoiding: insulting well meaning community members, and poisoning the mood. Assuming ‘good faith’ when responding almost always works better even if the original response was not in fact in good faith.

Please don’t send your question more than once: The question was seen. If you didn’t get a response then likely nobody has had time to answer you. Alternatively, it could be that nobody knows the answer. In any case, sending it again is poor form and akin to shouting and is likely to aggravate a large number of people. This also applies to crossposting. Try to pick the forum which you think matches best and ask there. If you are referred to a new forum, provide a link to the old discussion.

On https://answers.ros.org you can edit your question to provide more details. The more details that you include in your question the easier it is for others to help you find your solution which makes it more likely for you to get a response.

It’s considered bad form to list your personal deadlines; community members answering questions also have them.

Do not beg for help. If there is someone willing and able to help with your problem, you usually get a response. Asking for faster answers will mostly have a negative effect.
Do not add unrelated content to posts. The content of posts should be focused on the topic at hand and not include unrelated content. Content, links, and images unrelated to the topic are considered spam.

For commercial posts, see also this discussion.

Minimize references to content behind pay walls. The content posted on ROS Discourse and ROS Answers should “generally” be free and open to all users. Links to content behind pay walls such as private journal articles, textbooks, and paid news websites, while helpful and relevant, may not be accessible to all users. Where possible primary sources should be free and open with paid content playing a supporting role.

Single link posts are to be avoided. Generally speaking, posting a single link answer is less helpful and can be easily confused with spam. Moreover, links may degrade over time or be replaced. Paraphrasing a link’s content along with some contextual information and attribution is often much more helpful.

**Private contact**

If you’d like to contact us privately (e.g., if your question contains information sensitive to your organization or project, or if it’s regarding a security issue), you can email us directly at ros@osrfoundation.org.

### 4.8.7 The ROS 2 Project

Check out the resources below to learn more about the advancement of the ROS 2 project.

**Contributing**

A few things to remember before you start contributing to the ROS 2 project.

**Tenets**

- Respect what came before
  
  ROS has been around for more than a decade and is used by developers and across the world. Keep a humble attitude and an open mindset while contributing.

- Engage Open Robotics as early as possible
  
  – Open Robotics acts as a gate-keeper and advocate for the ROS community. Rely on their expertise and technical judgement from the design phase.
  
  – Start discussions with Open Robotics and the community early. Long time ROS contributors may have a clearer vision of the bigger picture. If you implement a feature and send a pull request without discussing with the community first, you are taking the risk of it being rejected, or you may be asked to largely rethink your design.
  
  – Opening issues or using Discourse to socialize an idea before starting the implementation is generally preferable.
• Adopt community best-practices whenever possible instead of ad-hoc processes

Think about your end-user’s experience when developing and contributing. Avoid using non-standard tools or libraries that may not be accessible to everyone.

• Think about the community as a whole

Think about the bigger picture. There are developers building different robots with different constraints. ROS needs to accommodate requirements of the whole community.

There are a number of ways you can contribute to the ROS 2 project.

**Discussions and support**

Some of the easiest ways to contribute to ROS 2 involve engaging in community discussions and support. You can find more information on how to pitch in on the *Contact* page.

**Contributing code**

**Setting up your development environment**

To get started, you’ll want to install from source; follow *the source installation instructions* for your platform.

**Development Guides**

**ROS 2 developer guide**

**Table of Contents**

- *General Principles*
- *Quality Practices*
  - Versioning
  - Change control process
  - Documentation
  - Testing
- *General Practices*
  - Issues
  - Branches
  - Pull requests
  - Library versioning
  - Development process
  - Changes to RMW API
  - Tracking tasks
  - Programming conventions
This page defines the practices and policies we employ when developing ROS 2.

**General Principles**

Some principles are common to all ROS 2 development:

- **Shared ownership**: Everybody working on ROS 2 should feel ownership over all parts of the system. The original author of a chunk of code does not have any special permission or obligation to control or maintain that chunk of code. Everyone is free to propose changes anywhere, to handle any type of ticket, and to review any pull request.

- **Be willing to work on anything**: As a corollary to shared ownership, everybody should be willing to take on any available task and contribute to any aspect of the system.

- **Ask for help**: If you run into trouble on something, ask your fellow developers for help, via tickets, comments, or email, as appropriate.

**Quality Practices**

Packages can ascribe to different levels of quality based on the development practices they adhere to, as per the guidelines in REP 2004: Package Quality Categories. The categories are differentiated by their policies on versioning, testing, documentation, and more.

The following sections are the specific development rules we follow to ensure core packages are of the highest quality (‘Level 1’). We recommend all ROS developers strive to adhere to the following policies to ensure quality across the ROS ecosystem.

**Versioning**

We will use the Semantic Versioning guidelines (semver) for versioning.

We will also adhere to some ROS-specific rules built on top of semver’s full meaning:

- Major version increments (i.e. breaking changes) should not be made within a released ROS distribution.
  - Patch (interface-preserving) and minor (non-breaking) version increments do not break compatibility, so these sorts of changes are allowed within a release.
Major ROS releases are the best time to release breaking changes. If a core package needs multiple breaking changes, they should be merged into their integration branch (e.g. rolling) to allow catching problems in CI quickly, but released together to reduce the number of major releases for ROS users.

Though major increments require a new distribution, a new distribution does not necessarily require a major bump (if development and release can happen without breaking API).

- For compiled code, the ABI is considered part of the public interface. Any change that requires recompiling dependent code is considered major (breaking).
- ABI breaking changes can be made in a minor version bump before a distribution release (getting added to the rolling release).
- We enforce API stability for core packages in Dashing and Eloquent even though their major version components are 0, despite SemVer’s specification regarding initial development.
- Subsequently, packages should strive to reach a mature state and increase to version 1.0.0 so to match semver’s specifications.

Caveats

These rules are best-effort. In unlikely, extreme cases, it may be necessary to break API within a major version/distribution. Whether an unplanned break increments the major or minor version will be assessed on a case-by-case basis.

For example, consider a situation involving released X-turtle, corresponding to major version 1.0.0, and released Y-turtle, corresponding to major version 2.0.0.

If an API-breaking fix is identified to be absolutely necessary in X-turtle, bumping to 2.0.0 is obviously not an option because 2.0.0 already exists.

The solutions for handling X-turtle’s version in such a case, both non-ideal, are:

1. Bumping X-turtle’s minor version: non-ideal because it violates SemVer’s principle that breaking changes must bump the major version.
2. Bumping X-turtle’s major version past Y-turtle (to 3.0.0): non-ideal because the older distro’s version would become higher than the already-available version of a newer distro, which would invalidate/break version-specific conditional code.

The developer will have to decide which solution to use, or more importantly, which principle they are willing to break. We cannot suggest one or the other, but in either case we do require that explicit measures be taken to communicate the disruption and its explanation to users manually (beyond just the version increment).

If there were no Y-turtle, even though the fix would technically just be a patch, X-turtle would have to bump to 2.0.0. This case adheres to SemVer, but breaks from our own rule that major increments should not be introduced in a released distribution.

This is why we consider the versioning rules best-effort. As unlikely as the examples above are, it is important to accurately define our versioning system.
Public API declaration

According to semver, every package must clearly declare a public API. We will use the “Public API Declaration” section of the quality declaration of a package to declare what symbols are part of the public API.

For most C and C++ packages the declaration is any header that it installs. However, it is acceptable to define a set of symbols which are considered private. Avoiding private symbols in headers can help with ABI stability, but is not required.

For other languages like Python, a public API must be explicitly defined, so that it is clear what symbols can be relied on with respect to the versioning guidelines. The public API can also be extended to build artifacts like configuration variables, CMake config files, etc. as well as executables and command-line options and output. Any elements of the public API should be clearly stated in the package’s documentation. If something you are using is not explicitly listed as part of the public API in the package’s documentation, then you cannot depend on it not changing between minor or patch versions.

Deprecation strategy

Where possible, we will also use the tick-tock deprecation and migration strategy for major version increments. New deprecations will come in a new distribution release, accompanied by compiler warnings expressing that the functionality is being deprecated. In the next release, the functionality will be completely removed (no warnings).

Example of function foo deprecated and replaced by function bar:

<table>
<thead>
<tr>
<th>Version</th>
<th>API</th>
</tr>
</thead>
<tbody>
<tr>
<td>X-turtle</td>
<td>void foo();</td>
</tr>
<tr>
<td>Y-turtle</td>
<td>[[deprecated(&quot;use bar()&quot;)]] void foo(); &lt;br&gt; void bar();</td>
</tr>
<tr>
<td>Z-turtle</td>
<td>void bar();</td>
</tr>
</tbody>
</table>

We must not add deprecations after a distribution is released. Deprecations do not necessarily require a major version bump, though. A deprecation can be introduced in a minor version bump if the bump happens before the distro is released (similar to ABI breaking changes).

For example, if X-turtle begins development as 2.0.0, a deprecation can be added in 2.1.0 before X-turtle is released.

We will attempt to maintain compatibility across distros as much as possible. However, like the caveats associated with SemVer, tick-tock or even deprecation in general may be impossible to completely adhere to in certain cases.

Change control process

- All changes must go through a pull request.
- We will enforce the Developer Certificate of Origin (DCO) on pull requests in ROSCore repositories.
  - It requires all commit messages to contain the Signed-off-by line with an email address that matches the commit author.
  - You can pass -s/--signoff to the git commit invocation or write the expected message manually (e.g. Signed-off-by: Your Name Developer <your.name@example.com>).
  - DCO is not required for pull requests that only address whitespace removal, typo correction, and other trivial changes.
- Always run CI jobs for all tier 1 platforms for every pull request and include links to jobs in the pull request. (If you don’t have access to the Jenkins jobs someone will trigger the jobs for you.)
• A minimum of 1 approval from a fellow developer who did not author the pull request is required to consider it approved. Approval is required before merging.
  – Packages may choose to increase this number.
• Any required changes to documentation (API documentation, feature documentation, release notes, etc.) must be proposed before merging related changes.

Guidelines for backporting PRs

When changing an older version of ROS:
• Make sure the features or fixes are accepted and merged in the rolling branch before opening a PR to backport the changes to older versions.
• When backporting to older versions, also consider backporting to any other still supported versions, even non-LTS versions.
• If you are backporting a single PR in its entirety, title the backport PR “[Distro] <name of original PR>”. If backporting a subset of changes from one or multiple PRs, the title should be “[Distro] <description of changes>”.
• Link to all PRs whose changes you’re backporting from the description of your backport PR. In a Dashing backport of a Foxy change, you do not need to link to the Eloquent backport of the same change.

Documentation

All packages should have these documentation elements present in their README or linked to from their README:
• Description and purpose
• Definition and description of the public API
• Examples
• How to build and install (should reference external tools/workflows)
• How to build and run tests
• How to build documentation
• How to develop (useful for describing things like python setup.py develop)
• License and copyright statements

Each source file must have a license and copyright statement, checked with an automated linter.

Each package must have a LICENSE file, typically the Apache 2.0 license, unless the package has an existing permissive license (e.g. rviz uses three-clause BSD).

Each package should describe itself and its purpose assuming, as much as possible, that the reader has stumbled onto it without previous knowledge of ROS or other related projects.

Each package should define and describe its public API so that there is a reasonable expectation for users about what is covered by the semantic versioning policy. Even in C and C++, where the public API can be enforced by API and ABI checking, it is a good opportunity to describe the layout of the code and the function of each part of the code.

It should be easy to take any package and from that package’s documentation understand how to build, run, build and run tests, and build the documentation. Obviously we should avoid repeating ourselves for common workflows, like building a package in a workspace, but the basic workflows should be either described or referenced.
Finally, it should include any documentation for developers. This might include workflows for testing the code using something like `python setup.py develop`, or it might mean describing how to make use of extension points provided by your package.

Examples:

- **capabilities**: [https://docs.ros.org/hydro/api/capabilities/html/](https://docs.ros.org/hydro/api/capabilities/html/)
  
  This one gives an example of docs which describe the public API

- **catkin_tools**: [https://catkin-tools.readthedocs.org/en/latest/development/extending_the_catkin_command.html](https://catkin-tools.readthedocs.org/en/latest/development/extending_the_catkin_command.html)

  This is an example of describing an extension point for a package

*(API docs are not yet being automatically generated)*

**Testing**

All packages should have some level of *system, integration, and/or unit tests*.

**Unit tests** should always be in the package which is being tested and should make use of tools like `Mock` to try and test narrow parts of the codebase in constructed scenarios. Unit tests should not bring in test dependencies that are not testing tools, e.g. `gtest`, `nose`, `pytest`, `mock`, etc.

**Integration tests** can test interactions between parts of the code or between parts of the code and the system. They often test software interfaces in ways that we expect the user to use them. Like Unit tests, Integration tests should be in the package which is being tested and should not bring in non-tool test dependencies unless absolutely necessary, i.e. all non-tool dependencies should only be allowed under extreme scrutiny so they should be avoided if possible.

**System tests** are designed to test end-to-end situations between packages and should be in their own packages to avoid bloating or coupling packages and to avoid circular dependencies.

In general external or cross package test dependencies should be minimized to prevent circular dependencies and tightly coupled test packages.

All packages should have some unit tests and possibly integration tests, but the degree to which they should have them is based on the package’s quality category. The following subsections apply to ‘Level 1’ packages:

**Code coverage**

We will provide line coverage, and achieve line coverage above 95%. If a lower percentage target is justifiable, it must be prominently documented. We may provide branch coverage, or exclude code from coverage (test code, debug code, etc.). We require that coverage increase or stay the same before merging a change, but it may be acceptable to make a change that decreases code coverage with proper justification (e.g. deleting code that was previously covered can cause the percentage to drop).

**Performance**

We strongly recommend performance tests, but recognize they don’t make sense for some packages. If there are performance tests, we will choose to either check each change or before each release or both. We will also require justification for merging a change or making a release that lowers performance.
Linters and static analysis

We will use *ROS code style* and enforce it with linters from `ament_lint_common`. All linters/static analysis that are part of `ament_lint_common` must be used.

The `ament_lint_auto` documentation provides information on running `ament_lint_common`.

General Practices

Some practices are common to all ROS 2 development.

These practices don’t affect package quality level as described in REP 2004, but are still highly recommended for the development process.

Issues

When filing an issue please make sure to:

- Include enough information for another person to understand the issue. In ROS 2, the following points are needed for narrowing down the cause of an issue. Testing with as many alternatives in each category as feasible will be especially helpful.

  - **The operating system and version.** Reasoning: ROS 2 supports multiple platforms, and some bugs are specific to particular versions of operating systems/compilers.

  - **The installation method.** Reasoning: Some issues only manifest if ROS 2 has been installed from “fat archives” or from Debians. This can help us determine if the issue is with the packaging process.

  - **The specific version of ROS 2.** Reasoning: Some bugs may be present in a particular ROS 2 release and later fixed. It is important to know if your installation includes these fixes.

  - **The DDS/RMW implementation being used** (see this page <../../Concepts/Intermediate/About-Different-Middleware-Vendors> for how to determine which one). Reasoning: Communication issues may be specific to the underlying ROS middleware being used.

  - **The ROS 2 client library being used.** Reasoning: This helps us narrow down the layer in the stack at which the issue might be.

- Include a list of steps to reproduce the issue.

- In case of a bug consider to provide a short, self contained, correct (compilable), example. Issues are much more likely to be resolved if others can reproduce them easily.

- Mention troubleshooting steps that have been tried already, including:

  - Upgrading to the latest version of the code, which may include bug fixes that have not been released yet. See this section <building-from-source> and follow the instructions to get the “rolling” branches.

  - Trying with a different RMW implementation. See this page <../../How-To-Guides/Working-with-multiple-RMW-implementations> for how to do that.
Branches

Note: These are just guidelines. It is up to the package maintainer to choose branch names that match their own workflow.

It is good practice to have separate branches in a package’s source repository for each ROS distribution it is targeting. These branches are typically named after the distribution they target. For example, a humble branch for development targeted specifically at the Humble distribution.

Releases are also made from these branches, targeting the appropriate distribution. Development targeted at a specific ROS distribution can happen on the appropriate branch. For example: Development commits targeting foxy are made to the foxy branch, and package releases for foxy are made from that same branch.

Note: This requires the package maintainers to perform backports or forwardports as appropriate to keep all branches up to date with features. The maintainers must also perform general maintenance (bug fixes, etc.) on all branches from which package releases are still made.

For example, if a feature is merged into the Rolling-specific branch (e.g. rolling or main), and that feature is also appropriate to the Humble distribution (does not break API, etc.), then it is good practice to backport the feature to the Humble-specific branch.

The maintainers may make releases for those older distributions if there are new features or bug fixes available.

What about main and rolling?

main typically targets Rolling (and so, the next unreleased ROS distribution), though the maintainers may decide to develop and release from a rolling branch instead.

Pull requests

• A pull request should only focus on one change. Separate changes should go into separate pull requests. See GitHub’s guide to writing the perfect pull request.
• A patch should be minimal in size and avoid any kind of unnecessary changes.
• A pull request must contain minimum number of meaningful commits.
  – You can create new commits while the pull request is under review.
• Before merging a pull request all changes should be squashed into a small number of semantic commits to keep the history clear.
  – But avoid squashing commits while a pull request is under review. Your reviewers might not notice that you made the change, thereby introducing potential for confusion. Plus, you’re going to squash before merging anyway; there’s no benefit to doing it early.
• Any developer is welcome to review and approve a pull request (see General Principles).
• When you are working on a change that is not ready for review or to be merged, use a draft pull request. When that change is ready for review, move the pull request out of the draft state. Note that if you want early feedback from specific people on a draft pull request, you can @ mention them in the pull request’s description or in a comment on the pull request.
• If your pull request depends on other pull requests, link to each depended on pull request by adding – Depends on <link> at the top of your pull request’s description. Doing so helps reviewers understand the context of the pull request.
• When you start reviewing a pull request, comment on the pull request so that other developers know that you’re reviewing it.

• Pull-request review is not read-only, with the reviewer making comments and then waiting for the author to address them. As a reviewer, feel free to make minor improvements (typos, style issues, etc.) in-place. As the opener of a pull-request, if you are working in a fork, checking the box to allow edits from upstream contributors will assist with the aforementioned. As a reviewer, also feel free to make more substantial improvements, but consider putting them in a separate branch (either mention the new branch in a comment, or open another pull request from the new branch to the original branch).

• Any developer (the author, the reviewer, or somebody else) can merge any approved pull request.

Library versioning

We will version all libraries within a package together. This means that libraries inherit their version from the package. This keeps library and package versions from diverging and shares reasoning with the policy of releasing packages which share a repository together. If you need libraries to have different versions then consider splitting them into different packages.

Development process

• The default branch (in most cases the rolling branch) must always build, pass all tests and compile without warnings. If at any time there is a regression it is the top priority to restore at least the previous state.

• Always build with tests enabled.

• Always run tests locally after changes and before proposing them in a pull request. Besides using automated tests, also run the modified code path manually to ensure that the patch works as intended.

• Always run CI jobs for all platforms for every pull request and include links to the jobs in the pull request.

For more details on recommended software development workflow, see Software Development Lifecycle section.

Changes to RMW API

When updating RMW API, it is required that RMW implementations for the Tier 1 middleware libraries are updated as well. For example, a new function `rmw_foo()` introduced to the RMW API must be implemented in the following packages (as of ROS Galactic):

• `rmw_connextdds`
• `rmw_cyclonedds`
• `rmw_fastrtps`

Updates for non-Tier 1 middleware libraries should also be considered if feasible (e.g. depending on the size of the change). See REP-2000 for the list of middleware libraries and their tiers.
Tracking tasks

To help organize work on ROS 2, the core ROS 2 development team uses kanban-style GitHub project boards. Not all issues and pull requests are tracked on the project boards, however. A board usually represents an upcoming release or specific project. Tickets can be browsed on a per-repo basis by browsing the ROS 2 repositories’ individual issue pages.

The names and purposes of columns in any given ROS 2 project board vary, but typically follow the same general structure:

- **To do**: Issues that are relevant to the project, ready to be assigned
- **In progress**: Active pull requests on which work is currently in progress
- **In review**: Pull requests where work is complete and ready for review, and for those currently under active review
- **Done**: Pull requests and related issues are merged/closed (for informational purposes)

To request permission to make changes, simply comment on the tickets you’re interested in. Depending on the complexity, it might be useful to describe how you plan to address it. We will update the status (if you don’t have the permission) and you can start working on a pull request. If you contribute regularly we will likely just grant you permission to manage the labels etc. yourself.

Programming conventions

- Defensive programming: ensure that assumptions are held as early as possible. E.g. check every return code and make sure to at least throw an exception until the case is handled more gracefully.
- All error messages must be directed to `stderr`.
- Declare variables in the narrowest scope possible.
- Keep group of items (dependencies, imports, includes, etc.) ordered alphabetically.

C++ specific

- Avoid using direct streaming (`<<`) to `stdout / stderr` to prevent interleaving between multiple threads.
- Avoid using references for `std::shared_ptr` since that subverts the reference counting. If the original instance goes out of scope and the reference is being used it accesses freed memory.

Filesystem layout

The filesystem layout of packages and repositories should follow the same conventions in order to provide a consistent experience for users browsing our source code.
Package layout

- **src**: contains all C and C++ code
  - Also contains C/C++ headers which are not installed
- **include**: contains all C and C++ headers which are installed
  - `<package_name>`: for all C and C++ installed headers they should be folder namespaced by the package name
- `<package_name>`: contains all Python code
- **test**: contains all automated tests and test data
- **config**: contains configuration files, e.g. YAML parameters files and RViz config files
- **doc**: contains all the documentation
- **launch**: contains all launch files
- **package.xml**: as defined by REP-0140 (may be updated for prototyping)
- **CMakeLists.txt**: only ROS packages which use CMake
- **setup.py**: only ROS packages which use Python code only
- **README**: can be rendered on GitHub as a landing page for the project
  - This can be as short or detailed as is convenient, but it should at least link to project documentation
  - Consider putting a CI or code coverage tag in this README
  - It can also be .rst or anything else that GitHub supports
- **CONTRIBUTING**: describes the contribution guidelines
  - This might include license implication, e.g. when using the Apache 2 License.
- **LICENSE**: a copy of the license or licenses for this package
- **CHANGELOG.rst**: REP-0132 compliant changelog

Repository layout

Each package should be in a subfolder which has the same name as the package. If a repository contains only a single package it can optionally be in the root of the repository.

Developer Workflow

We track open tickets and active PRs related to upcoming releases and larger projects using GitHub project boards. The usual workflow is:

- Discuss design (GitHub ticket on the appropriate repository, and a design PR to https://github.com/ros2/design if needed)
- Write implementation on a feature branch on a fork
  - Please check out the developer guide `<Developer-Guide>` for guidelines and best practices
- Write tests
- Enable and run linters
• Run tests locally using `colcon test` (see the *colcon tutorial*)

• Once everything builds locally without warnings and all tests are passing, run CI on your feature branch:
  – Go to `ci.ros2.org`
  – Log in (top right corner)
  – Click on the `ci_launcher` job
  – Click “Build with Parameters” (left column)
  – In the first box “CI_BRANCH_TO_TEST” enter your feature branch name
  – Hit the `build` button

(if you are not a ROS 2 committer, you don’t have access to the CI farm. In that case, ping the reviewer of your PR to run CI for you)

• If your use case requires running code coverage:
  – Go to `ci.ros2.org`
  – Log in (top right corner)
  – Click on the `ci_linux_coverage` job
  – Click “Build with Parameters” (left column)
  – Be sure of leaving “CI_BUILD_ARGS” and “CI_TEST_ARGS” with the default values
  – Hit the `build` button
  – At the end of the document there are instructions on how to interpret the result of the report and calculate the coverage rate

• If the CI job built without warnings, errors and test failures, post the links of your jobs on your PR or high-level ticket aggregating all your PRs (see example [here](#))
  – Note that the markdown for these badges is in the console output of the `ci_launcher` job

• When the PR has been approved:
  – the person who submitted the PR merges it using “Squash and Merge” option so that we keep a clean history
    * If the commits deserve to keep separated: squash all the nitpick/linters/typo ones together and merge the remaining set
      • Note: each PR should target a specific feature so Squash and Merge should make sense 99% of the time
  • Delete the branch once merged

**Architectural Development Practices**

This section describes the ideal lifecycle that should be employed when making large architectural changes to ROS 2.
Software Development Lifecycle

This section describes step-by-step how to plan, design, and implement a new feature:

1. Task Creation
2. Creating the Design Document
3. Design Review
4. Implementation
5. Code Review

Task creation

Tasks requiring changes to critical parts of ROS 2 should have design reviews during early stages of the release cycle. If a design review is happening in the later stages, the changes will be part of a future release.

- An issue should be created in the appropriate ros2 repository, clearly describing the task being worked on.
  - It should have a clear success criteria and highlight the concrete improvements expected from it.
  - If the feature is targeting a ROS release, ensure this is tracked in the ROS release ticket (example).

Writing the design document

Design docs must never include confidential information. Whether or not a design document is required for your change depends on how big the task is.

1. You are making a small change or fixing a bug:
   - A design document is not required, but an issue should be opened in the appropriate repository to track the work and avoid duplication of efforts.

2. You are implementing a new feature or would like to contribute to OSRF-owned infrastructure (like Jenkins CI):
   - Design doc is required and should be contributed to ros2/design to be made accessible on https://design.ros2.org/.
   - You should fork the repository and submit a pull request detailing the design.
   
   Mention the related ros2 issue (for example, Design doc for task ros2/ros2#<issue id>) in the pull request or the commit message. Detailed instructions are on the ROS 2 Contribute page. Design comments will be made directly on the pull request.

If the task is planned to be released with a specific version of ROS, this information should be included in the pull request.
Design document review

Once the design is ready for review, a pull request should be opened and appropriate reviewers should be assigned. It is recommended to include project owner(s) - maintainers of all impacted packages (as defined by package.xml maintainer field, see REP-140) - as reviewers.

- If the design doc is complex or reviewers have conflicting schedules, an optional design review meeting can be set up. In this case,

  **Before the meeting**
  - Send a meeting invite at least one week in advance
  - Meeting duration of one hour is recommended
  - Meeting invite should list all decisions to be made during the review (decisions requiring package maintainer approval)
  - **Meeting required attendees: design pull request reviewers** Meeting optional attendees: all OSRF engineers, if applicable

  **During the meeting**
  - The task owner drives the meeting, presents their ideas and manages discussions to ensure an agreement is reached on time

  **After the meeting**
  - The task owner should send back meeting notes to all attendees
  - If minor issues have been raised about the design:
    - * The task owner should update the design doc pull request based on the feedback
    - * Additional review is not required
  - If major issues have been raised about the design:
    - * It is acceptable to remove sections for which there is no clear agreement
    - * The debatable parts of the design can be resubmitted as a separate task in the future
    - * If removing the debatable parts is not an option, work directly with package owners to reach an agreement

  - Once consensus is reached:
    - Ensure the ros2/design pull request has been merged, if applicable
    - Update and close the GitHub issue associated with this design task

Implementation

Before starting, go through the *Pull requests* section for best practices.

- For each repo to be modified:
  - Modify the code, go to the next step if finished or at regular intervals to backup your work.
  - **Self-review** your changes using `git add -i`.
  - Create a new signed commit using `git commit -s`. 
A pull request should contain minimal semantically meaningful commits (for instance, a large number of 1-line commits is not acceptable). Create new fixup commits while iterating on feedback, or optionally, amend existing commits using `git commit --amend` if you don’t want to create a new commit every time.

Each commit must have a properly written, meaningful, commit message. More instructions here.

Moving files must be done in a separate commit, otherwise git may fail to accurately track the file history.

Either the pull request description or the commit message must contain a reference to the related ros2 issue, so it gets automatically closed when the pull request is merged. See this doc for more details.

Push the new commits.

**Code review**

Once the change is ready for code review:

- Open a pull request for each modified repository.
  - Remember to follow Pull requests best practices.
  - GitHub can be used to create pull requests from the command-line.
  - If the task is planned to be released with a specific version of ROS, this information should be included in each pull request.

- Package owners who reviewed the design document should be mentioned in the pull request.

- Code review SLO: although reviewing pull requests is best-effort, it is helpful to have reviewers comment on pull requests within a week and code authors to reply back to comments within a week, so there is no loss of context.

- Iterate on feedback as usual, amend and update the development branch as needed.

- Once the PR is approved, package maintainers will merge the changes in.

**Build Farm Introduction**

The build farm is located at [ci.ros2.org](http://ci.ros2.org).

Every night we run nightly jobs which build and run all the tests in various scenarios on various platforms. Additionally, we test all pull requests against these platforms before merging.

This is the current set of target platforms and architectures, though it evolves overtime:

- Ubuntu 22.04 Jammy
  - amd64
  - aarch64

- Windows 10
  - amd64

There are several categories of jobs on the buildfarm:

- manual jobs (triggered manually by developers):
  - `ci_linux`: build + test the code on Ubuntu Xenial
  - `ci_linux-aarch64`: build + test the code on Ubuntu Xenial on an ARM 64-bit machine (aarch64)
- ci_linux_coverage: build + test + generation of test coverage
- ci_windows: build + test the code on Windows 10
- ci_launcher: trigger all the jobs listed above

• nightly (run every night):
  - Debug: build + test the code with CMAKE_BUILD_TYPE=Debug
    * nightly_linux_debug
    * nightly_linux-aarch64_debug
    * nightly_win_debug
  - Release: build + test the code with CMAKE_BUILD_TYPE=Release
    * nightly_linux_release
    * nightly_linux-aarch64_release
    * nightly_win_release
  - Repeated: build then run each test up to 20 times or until failed (aka flakiness hunter)
    * nightly_linux_repeated
    * nightly_linux-aarch64_repeated
    * nightly_win_repeated
  - Coverage:
    * nightly_linux_coverage: build + test the code + analyses coverage for c/c++ and python
      · results are exported as a cobertura report

• packaging (run every night; result is bundled into an archive):
  - packaging_linux
  - packaging_windows

Two additional build farms support the ROS / ROS 2 ecosystem by providing building of source and binary packages, continuous integration, testing, and analysis.

For details, frequently asked questions, and troubleshooting see build farms.

Note on Coverage runs

ROS 2 packages are organized in a way that the testing code for a given package is not only contained within the package, but could also be present in a different package. In other words: packages can exercise code belonging to other packages during the testing phase.

To achieve the coverage rate reached by all code available in the ROS 2 core packages it is recommended to run builds using a fixed set of proposed repositories. That set is defined in the default parameters of coverage jobs in Jenkins.
How to read the coverage rate from the buildfarm report

To see the coverage report for a given package:

- When the ci_linux_coverage build finishes, click on Coverage Report
- Scroll down to the Coverage Breakdown by Package table
- In the table, look at the first column called “Name”

The coverage reports in the buildfarm include all the packages that were used in the ROS workspace. The coverage report includes different paths corresponding to the same package:

- Name entries with the form: `src.*.<repository_name>.<package_name>.*` These correspond to the unit test runs available in a package against its own source code
- Name entries with the form: `build.<repository_name>.<package_name>.*` These correspond to the unit test runs available in a package against its files generated at building or configuring time
- Name entries with the form: `install.<package_name>.*` These correspond to the system/integration tests coming from testing runs of other packages

How to calculate the coverage rate from the buildfarm report

Get the combined unit coverage rate using the automatic script:

- From the ci_linux_coverage Jenkins build copy the URL of the build
- Download the `get_coverage_ros2_pkg` script
- Execute the script: `./get_coverage_ros2_pkg.py <jenkins_build_url> <ros2_package_name>`
- Grab the results from the “Combined unit testing” final line in the output of the script

Alternative: get the combined unit coverage rate from coverage report (require manual calculation):

- When the ci_linux_coverage build finishes, click on Cobertura Coverage Report
- Scroll down to the Coverage Breakdown by Package table
- In the table, under the first column “Name”, look for (where `<package_name>` is your package under testing):
  - all the directories under the pattern `src.*.<repository_name>.<package_name>.*` grab the two absolute values in the column “Lines”.
  - all the directories under the pattern `build/.<repository_name>.*` grab the two absolute values in the column “Lines”.
- With the previous selection: for each cell, the first value is the lines tested and the second is the total lines of code. Aggregate all rows for getting the total of the lines tested and the total of lines of code under test. Divide to get the coverage rate.
How to measure coverage locally using lcov (Ubuntu)

To measure coverage on your own machine, install `lcov`.

```
sudo apt install -y lcov
```

The rest of this section assumes you are working from your colcon workspace. Compile in debug with coverage flags. Feel free to use colcon flags to target specific packages.

```
colcon build --cmake-args -DCMAKE_BUILD_TYPE=Debug -DCMAKE_CXX_FLAGS="${CMAKE_CXX_FLAGS}" --coverage -DCMAKE_C_FLAGS="${CMAKE_C_FLAGS} --coverage"
```

`lcov` requires an initial baseline, which you can produce with the following command. Update the output file location for your needs.

```
lcov --no-external --capture --initial --directory . --output-file ~/ros2_base.info
```

Run tests for the packages that matter for your coverage measurements. For example, if measuring `rclcpp` also with `test_rclcpp`

```
colcon test --packages-select rclcpp test_rclcpp
```

Capture the lcov results with a similar command this time dropping the `--initial` flag.

```
lcov --no-external --capture --directory . --output-file ~/ros2.info
```

Combine the trace .info files:

```
lcov --add-tracefile ~/ros2_base.info --add-tracefile ~/ros2.info --output-file ~/ros2_coverage.info
```

Generate html for easy visualization and annotation of covered lines.

```
mkdir -p coverage
genhtml ~/ros2_coverage.info --output-directory coverage
```

Code style and language versions

Table of Contents

- C
  - Standard
  - Style
- C++
  - Standard
  - Style
- Python
  - Version
In order to achieve a consistent looking product we will all follow externally (if possible) defined style guidelines for each language. For other things like package layout or documentation layout we will need to come up with our own guidelines, drawing on current, popular styles in use now.

Additionally, wherever possible, developers should use integrated tools to allow them to check that these guidelines are followed in their editors. For example, everyone should have a PEP8 checker built into their editor to cut down on review iterations related to style.

Also where possible, packages should check style as part of their unit tests to help with the automated detection of style issues (see ament_lint_auto).

C

Standard

We will target C99.

Style

We will use Python’s PEP7 for our C style guide, with some modifications and additions:

- We will target C99, as we do not need to support C89 (as PEP7 recommends)
  - rationale: among other things it allows us to use both // and /* */ style comments
  - rationale: C99 is pretty much ubiquitous now
- C++ style // comments are allowed
- (optional) Always place literals on the left-hand side of comparison operators, e.g. \( \theta == \text{ret} \) instead of \( \text{ret} == \theta \)
  - rationale: \( \text{ret} == \theta \) too easily turns into \( \text{ret} = \theta \) by accident
  - optional because when using -Wall (or equivalent) modern compilers will warn you when this happens

All of the following modifications only apply if we are not writing Python modules:

- Do not use Py_ as a prefix for everything
  - instead use a CamelCase version of the package name or other appropriate prefix
- The stuff about documentation strings doesn’t apply

We can use the pep7 python module for style checking. The editor integration seems slim, we may need to look into automated checking for C in more detail.
C++

Standard

Iron targets C++17.

Style

We will use the Google C++ Style Guide, with some modifications:

Line Length

• Our maximum line length is 100 characters.

File Extensions

• Header files should use the .hpp extension.
  – rationale: Allow tools to determine content of files, C++ or C.
• Implementation files should use the .cpp extension.
  – rationale: Allow tools to determine content of files, C++ or C.

Variable Naming

• For global variables use lowercase with underscores prefixed with g_
  – rationale: keep variable naming case consistent across the project
  – rationale: easy to tell the scope of a variable at a glance
  – consistency across languages

Function and Method Naming

• Google style guide says CamelCase, but the C++ std library’s style of snake_case is also allowed
  – rationale: ROS 2 core packages currently use snake_case
    * reason: either an historical oversight or a personal preference that didn’t get checked by the linter
    * reason for not changing: retroactively changing would be too disruptive
  – other considerations:
    * cpplint.py does not check this case (hard to enforce other than with review)
    * snake_case can result in more consistency across languages
  – specific guidance:
    * for existing projects, prefer the existing style
    * for new projects, either is acceptable, but a preference for matching related existing projects is advised
final decision is always developer discretion
· special cases like function pointers, callable types, etc. may require bending the rules
* Note that classes should still use CamelCase by default

Access Control

- Drop requirement for all class members to be private and therefore require accessors
  - rationale: this is overly constraining for user API design
  - we should prefer private members, only making them public when they are needed
  - we should consider using accessors before choosing to allow direct member access
  - we should have a good reason for allowing direct member access, other than because it is convenient for us

Exceptions

- Exceptions are allowed
  - rationale: this is a new codebase, so the legacy argument doesn’t apply to us
  - rationale: for user-facing API’s it is more idiomatic C++ to have exceptions
  - Exceptions in destructors should be explicitly avoided
- We should consider avoiding Exceptions if we intend to wrap the resulting API in C
  - rationale: it will make it easier to wrap in C
  - rationale: most of our dependencies in code we intend to wrap in C do not use exceptions anyways

Function-like Objects

- No restrictions on Lambda’s or std::function or std::bind

Boost

- Boost should be avoided unless absolutely required.

Comments and Doc Comments

- Use /// and /** */ comments for documentation purposes and // style comments for notes and general comments
  - Class and Function comments should use /// and /** */ style comments
  - rationale: these are recommended for Doxygen and Sphinx in C/C++
  - rationale: mixing */ */ and // is convenient for block commenting out code which contains comments
  - Descriptions of how the code works or notes within classes and functions should use // style comments
Pointer Syntax Alignment

• Use `char * c;` instead of `char* c;` or `char *c;` because of this scenario `char* c, *d, *e;`

Class Privacy Keywords

• Do not put 1 space before `public:`, `private:`, or `protected:`, it is more consistent for all indentions to be a multiple of 2
  – rationale: most editors don’t like indentions which are not a multiple of the (soft) tab size
  – Use zero spaces before `public:`, `private:`, or `protected:`, or 2 spaces
  – If you use 2 spaces before, indent other class statements by 2 additional spaces
  – Prefer zero spaces, i.e. `public:`, `private:`, or `protected:` in the same column as the class

Nested Templates

• Never add whitespace to nested templates
  – Prefer `set<list<string>>` (C++11 feature) to `set<list<string>>` or `set< list<string> >`

Always Use Braces

• Always use braces following `if`, `else`, `do`, `while`, and `for`, even when the body is a single line.
  – rationale: less opportunity for visual ambiguity and for complications due to use of macros in the body

Open Versus Cuddled Braces

• Use open braces for `function`, `class`, `enum`, and `struct` definitions, but cuddle braces on `if`, `else`, `while`, `for`, etc...
  – Exception: when an `if` (or `while`, etc.) condition is long enough to require line-wrapping, then use an open brace (i.e., don’t cuddle).
• When a function call cannot fit on one line, wrap at the open parenthesis (not in between arguments) and start them on the next line with a 2-space indent. Continue with the 2-space indent on subsequent lines for more arguments. (Note that the Google style guide is internally contradictory on this point.)
  – Same goes for `if` (and `while`, etc.) conditions that are too long to fit on one line.

Examples

This is OK:

```c
int main(int argc, char **argv)
{
    if (condition) {
        return 0;
    } else {
        return 1;
    }
}(continues on next page)
```
if (this && that || both) {
...
}

// Long condition; open brace
if (this && that || both && this && that || both && this && that) {
...
}

// Short function call
call_func(foo, bar);

// Long function call; wrap at the open parenthesis
call_func(
  foo, bar, foo, bar, foo, bar, foo, bar, foo, bar, foo, bar, foo, bar, foo, bar, foo,
  bar,
  foo, bar, foo, bar, foo, bar, foo, bar, foo, bar, foo, bar, foo, bar, foo, bar, foo,
  bar);

// Very long function argument; separate it for readability
call_func(
  bang,
  foooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo,
  bar, bat);

This is not OK:

int main(int argc, char **argv) {
  return 0;
}

if (this &&
    that ||
  both) {
...
}

Use open braces rather than excessive indentation, e.g. for distinguishing constructor code from constructor initializer lists.

This is OK:

ReturnType LongClassName::ReallyReallyReallyLongFunctionName(
  Type par_name1, // 2 space indent
  Type par_name2,
  Type par_name3)
{
  DoSomething(); // 2 space indent

  (continues on next page)


```cpp
... 
}

MyClass::MyClass(int var)  
: some_var_(var),  
  some_other_var_(var + 1)  
{
  ...  
  DoSomething();  
  ...  
}

This is **not** OK, even weird (the google way?):

```cpp
ReturnType LongClassName::ReallyReallyReallyLongFunctionName(  
    Type par_name1,  // 4 space indent  
    Type par_name2,  
    Type par_name3) {  
  DoSomething();  // 2 space indent  
  ...  
}

MyClass::MyClass(int var)  
: some_var_(var),  // 4 space indent  
  some_other_var_(var + 1) {  // lined up  
  ...  
  DoSomething();  
  ...  
}
```

### Linters

We check these styles with a combination of Google’s `cpplint.py` and `uncrustify`. We provide command line tools with custom configurations:

- `ament_clang_format`: configuration
- `ament_cpplint`
- `ament_uncrustify`: configuration

Some formatters such as `ament_uncrustify` and `ament_clang_format` support `--reformat` options to apply changes in place.

We also run other tools to detect and eliminate as many warnings as possible. Here’s a non-exhaustive list of additional things we try to do on all of our packages:

- use compiler flags like `-Wall` `-Wextra` `-Wpedantic`
- run static code analysis like `cppcheck`, which we have integrated in `ament_cppcheck`.
**Python**

**Version**

We will target Python 3 for our development.

**Style**

We will use the PEP8 guidelines for code format. We chose the following more precise rule where PEP 8 leaves some freedom:

- We allow up to 100 characters per line (fifth paragraph).
- We pick single quotes over double quotes as long as no escaping is necessary.
- We prefer hanging indents for continuation lines.

Tools like the (ament_)pycodestyle Python package should be used in unit-test and/or editor integration for checking Python code style.

The pycodestyle configuration used in the linter is here.

Integration with editors:

- atom: https://atom.io/packages/linter-pycodestyle
- emacs: https://www.emacswiki.org/emacs/PythonProgrammingInEmacs
- Sublime Text: https://sublime.wbond.net/packages/SublimeLinter-flake8
- vim: https://github.com/nvie/vim-flake8

**CMake**

**Version**

We will target CMake 3.8.

**Style**

Since there is not an existing CMake style guide we will define our own:

- Use lowercase command names (find_package, not FIND_PACKAGE).
- Use snake_case identifiers (variables, functions, macros).
- Use empty else() and end...() commands.
- No whitespace before (‘s.
- Use two spaces of indentation, do not use tabs.
- Do not use aligned indentation for parameters of multi-line macro invocations. Use two spaces only.
- Prefer functions with set(PARENT_SCOPE) to macros.
- When using macros prefix local variables with _ or a reasonable prefix.
Markdown / reStructured Text / docblocks

Style

The following rules to format text is intended to increase readability as well as versioning.

- \[.md, .rst only\] Each section title should be preceded by one empty line and succeeded by one empty line.
  - Rationale: It expedites to get an overview about the structure when screening the document.

- \[.rst only\] In reStructured Text the headings should follow the hierarchy described in the Sphinx style guide:
  - # with overline (only once, used for the document title)
  - * with overline
  - =
  - 
  - ^
  - “
  - Rationale: A consistent hierarchy expedites getting an idea about the nesting level when screening the document.

- \[.mdonly\] In Markdown the headings should follow the ATX-style described in the Markdown syntax documentation
  - ATX-style headers use 1-6 hash characters (#) at the start of the line to denote header levels 1-6.
  - A space between the hashes and the header title should be used (such as # Heading 1) to make it easier to visually separate them.
  - Justification for the ATX-style preference comes from the Google Markdown style guide
  - Rationale: ATX-style headers are easier to search and maintain, and make the first two header levels consistent with the other levels.

- \[any\] Each sentence must start on a new line.
  - Rationale: For longer paragraphs a single change in the beginning makes the diff unreadable since it carries forward through the whole paragraph.

- \[any\] Each sentence can optionally be wrapped to keep each line short.

- \[any\] The lines should not have any trailing white spaces.

- \[.md, .rst only\] A code block must be preceded and succeeded by an empty line.
  - Rationale: Whitespace is significant only directly before and directly after fenced code blocks. Following these instructions will ensure that highlighting works properly and consistently.

- \[.md, .rst only\] A code block should specify a syntax (e.g. bash).
Quality guide: ensuring code quality

Table of Contents

- Static code analysis as part of the ament package build
- Static Thread Safety Analysis via Code Annotation
- Dynamic analysis (data races & deadlocks)

This page gives guidance about how to improve the software quality of ROS 2 packages, focusing on more specific areas than the Quality Practices section of the Developer Guide.

The sections below intend to address ROS 2 core, application and ecosystem packages and the core client libraries, C++ and Python. The solutions presented are motivated by design and implementation considerations to improve quality attributes like “Reliability”, “Security”, “Maintainability”, “Determinism”, etc. which relate to non-functional requirements.

Static code analysis as part of the ament package build

Context:

- You have developed your C++ production code.
- You have created a ROS 2 package with build support with ament.

Problem:

- Library level static code analysis is not run as part of the package build procedure.
- Library level static code analysis needs to be executed manually.
- Risk of forgetting to execute library level static code analysis before building a new package version.

Solution:

- Use the integration capabilities of ament to execute static code analysis as part of the package build procedure.

Implementation:

- Insert into the packages CMakeLists.txt file.

```cpp
if(BUILD_TESTING)
  find_package(ament_lint_auto REQUIRED)
  ament_lint_auto_find_test_dependencies()
  ...
endif()
```

- Insert the ament_lint test dependencies into the packages package.xml file.

```xml
<package format="2">
  ...
  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>
</package>
```

(continues on next page)
Examples:

- rclcpp:
  - rclcpp/rclcpp/CMakeLists.txt
  - rclcpp/rclcpp/package.xml

- rclcpp_lifecycle:
  - rclcpp/rclcpp_lifecycle/CMakeLists.txt
  - rclcpp/rclcpp_lifecycle/package.xml

Resulting context:

- The static code analysis tools supported by ament are run as part of the package build.
- Static code analysis tools not supported by ament need to be executed separately.

Static Thread Safety Analysis via Code Annotation

Context:

- You are developing/debugging your multithreaded C++ production code
- You access data from multiple threads in C++ code

Problem:

- Data races and deadlocks can lead to critical bugs.

Solution:

- Utilize Clang’s static Thread Safety Analysis by annotating threaded code

Context For Implementation:

To enable Thread Safety Analysis, code must be annotated to let the compiler know more about the semantics of the code. These annotations are Clang-specific attributes - e.g. `__attribute__((capability()))`. Instead of using those attributes directly, ROS 2 provides preprocessor macros that are erased when using other compilers.

These macros can be found in `rcpputils/thread_safety_annotations.hpp`

The Thread Safety Analysis documentation states that thread safety analysis can be used with any threading library, but it does require that the threading API be wrapped in classes and methods which have the appropriate annotations.

We have decided that we want ROS 2 developers to be able to use `std::` threading primitives directly for their development. We do not want to provide our own wrapped types as is suggested above.

There are three C++ standard libraries to be aware of:

- The GNU standard library `libstdc++` - default on Linux, explicitly via the compiler option `-stdlib=libstdc++`
- The LLVM standard library `libc++` (also called `libcxx`) - default on macOS, explicitly set by the compiler option `-stdlib=libc++`
- The Windows C++ Standard Library - not relevant to this use case
libcxx annotates its std::mutex and std::lock_guard implementations for Thread Safety Analysis. When using GNU libstdc++, those annotations are not present, so Thread Safety Analysis cannot be used on non-wrapped std:: types.

Therefore, to use Thread Safety Analysis directly with std:: types, we must use libcxx

Implementation:

The code migration suggestions here are by no means complete - when writing (or annotating existing) threaded code, you are encouraged to utilize as many of the annotations as is logical for your use case. However, this step-by-step is a great place to start!

- Enabling Analysis for Package/Target

  When the C++ compiler is Clang, enable the -Wthread-safety flag. Example below for CMake-based projects

  ```
  if(CMAKE_CXX_COMPILER_ID MATCHES "Clang")
    add_compile_options(-Wthread-safety)
      # for your whole package
    target_compile_options(${MY_TARGET} PUBLIC -Wthread-safety)
      # for a single,
        library or executable
  endif()
  ```

- Annotating Code

  - Step 1 - Annotate data members
    - Find anywhere that std::mutex is used to protect some member data
    - Add the RCPPUTILS_TSA_GUARDED_BY(mutex_name) annotation to the data that is protected by the mutex

  ```
  class Foo {
    public:
      void incr(int amount) {
        std::lock_guard<std::mutex> lock(mutex_);
        bar += amount;
      }

      void get() const {
        return bar;
      }
  }
  ```

  - Step 2 - Fix Warnings
    - In the above example - Foo::get will produce a compiler warning! To fix it, lock before returning bar

  ```
  void get() const {
    std::lock_guard<std::mutex> lock(mutex_);
    return bar;
  }
  ```

- Step 3 - (Optional but Recommended) Refactor Existing Code to Private-Mutex Pattern

4.8. ROS 2 Documentation 1343
A recommended pattern in threaded C++ code is to always keep your \texttt{mutex} as a \texttt{private:} member of the data structure. This makes data safety the concern of the containing structure, offloading that responsibility from users of the structure and minimizing the surface area of affected code.

Making your locks private may require rethinking the interfaces to your data. This is a great exercise - here are a few things to consider:

\begin{itemize}
  \item You may want to provide specialized interfaces for performing analysis that requires complex locking logic, e.g. counting members in a filtered set of a mutex-guarded map structure, instead of actually returning the underlying structure to consumers
  \item Consider copying to avoid blocking, where the amount of data is small. This can let other threads get on with accessing the shared data, which can potentially lead to better overall performance.
\end{itemize}

- Step 4 - (Optional) Enable Negative Capability Analysis

  \url{https://clang.llvm.org/docs/ThreadSafetyAnalysis.html#negative-capabilities}

Negative Capability Analysis lets you specify “this lock must not be held when calling this function”. It can reveal potential deadlock cases that other annotations cannot.

\begin{itemize}
  \item Where you specified \texttt{-Wthread-safety}, add the additional flag \texttt{-Wthread-safety-negative}
  \item On any function that acquires a lock, use the \texttt{RCPPUTILS_TSA_REQUIRES(!mutex)} pattern
\end{itemize}

- How to run the analysis

  - The ROS CI build farm runs a nightly job with libcxx, which will surface any issues in the ROS 2 core stack by being marked “Unstable” when Thread Safety Analysis raises warnings
  - For local runs, you have the following options, all equivalent

\begin{itemize}
  \item Use the \texttt{colcon clang-libcxx} mixin (see the documentation for configuring mixins)

\begin{verbatim}
colcon build --mixin clang-libcxx
\end{verbatim}

  \item Passing compiler to CMake

\begin{verbatim}
colcon build --cmake-args -DCMAKE_C_COMPILER=clang -DCMAKE_CXX_COMPILER=clang++ -DCMAKE_CXX_FLAGS="-stdlib=libc++ -D_LIBCPP_ENABLE_THREAD_SAFETY_ANNOTATIONS" -DFORCE_BUILD_VENDOR_PKG=ON --no-warn-unused-cli
\end{verbatim}

  \item Overriding system compiler

\begin{verbatim}
CC=clang CXX=clang++ colcon build --cmake-args -DCMAKE_CXX_FLAGS='"-stdlib=libc++ -D_LIBCPP_ENABLE_THREAD_SAFETY_ANNOTATIONS"' -DFORCE_BUILD_VENDOR_PKG=ON --no-warn-unused-cli
\end{verbatim}
\end{verbatim}

\end{itemize}

\textbf{Resulting Context:}

\begin{itemize}
  \item Potential deadlocks and race conditions will be surfaced at compile time, when using Clang and libcxx
Dynamic analysis (data races & deadlocks)

Context:
- You are developing/debugging your multithreaded C++ production code.
- You use pthreads or C++11 threading + llvm libc++ (in case of ThreadSanitizer).
- You do not use Libc/libstdc++ static linking (in case of ThreadSanitizer).
- You do not build non-position-independent executables (in case of ThreadSanitizer).

Problem:
- Data races and deadlocks can lead to critical bugs.
- Data races and deadlocks cannot be detected using static analysis (reason: limitation of static analysis).
- Data races and deadlocks must not show up during development debugging / testing (reason: usually not all possible control paths through production code exercised).

Solution:
- Use a dynamic analysis tool which focuses on finding data races and deadlocks (here clang ThreadSanitizer).

Implementation:
- Compile and link the production code with clang using the option -fsanitize=thread (this instruments the production code).
- In case different production code shall be executed during analysis consider conditional compilation e.g. ThreadSanitizers _has_feature(thread_sanitizer).
- In case some code shall not be instrumented consider ThreadSanitizers _/attribute*/_/((no_sanitize("thread"))).
- In case some files shall not be instrumented consider file or function-level exclusion ThreadSanitizers blacklisting, more specific: ThreadSanitizers Sanitizer Special Case List or with ThreadSanitizers no_sanitize("thread") and use the option --fsanitize-blacklist.

Resulting context:
- Higher chance to find data races and deadlocks in production code before deploying it.
- Analysis result may lack reliability, tool in beta phase stage (in case of ThreadSanitizer).
- Overhead due to production code instrumentation (maintenance of separate branches for instrumented/not instrumented production code, etc.).
- Instrumented code needs more memory per thread (in case of ThreadSanitizer).
- Instrumented code maps a lot virtual address space (in case of ThreadSanitizer).

Migration guide from ROS 1
There are two different kinds of package migrations:

- Migrating the source code of an existing package from ROS 1 to ROS 2 with the intent that a significant part of the source code will stay the same or at least similar. An example for this could be pluginlib where the source code is maintained in different branches within the same repository and commonly patches can be ported between those branches when necessary.

- Implementing the same or similar functionality of a ROS 1 package for ROS 2 but with the assumption that the source code will be significantly different. An example for this could be roscpp in ROS 1 and rclcpp in ROS 2 which are separate repositories and don’t share any code.

This article focuses on the former case and describes the high-level steps to migrate a ROS 1 package to ROS 2. It does not aim to be a step-by-step migration instruction and is not considered the final “solution”. Future versions will aim to make migration smoother and less effort up to the point of maintaining a single package from the same branch for ROS 1 as well as ROS 2.

**Prerequisites**

Before being able to migrate a ROS 1 package to ROS 2 all of its dependencies must be available in ROS 2.

**Migration steps**

- Package manifests
- Metapackages
- Message, service, and action definitions
- Build system
- Update source code
Package manifests

ROS 2 doesn’t support format 1 of the package specification but only newer format versions (2 and higher). Therefore the package.xml file must be updated to at least format 2 if it uses format 1. Since ROS 1 supports all formats it is safe to perform that conversion in the ROS 1 package.

Some packages might have different names in ROS 2 so the dependencies might need to be updated accordingly.

Metapackages

ROS 2 doesn’t have a special package type for metapackages. Metapackages can still exist as regular packages that only contain runtime dependencies. When migrating metapackages from ROS 1, simply remove the <metapackage /> tag in your package manifest.

Message, service, and action definitions

Message files must end in .msg and must be located in the subfolder msg. Service files must end in .srv and must be located in the subfolder srv. Actions files must end in .action and must be located in the subfolder action.

These files might need to be updated to comply with the ROS Interface definition. Some primitive types have been removed and the types duration and time which were builtin types in ROS 1 have been replaced with normal message definitions and must be used from the builtin_interfaces package. Also some naming conventions are stricter than in ROS 1.

In your package.xml:

- Add <buildtool_depend>rosidl_default_generators</buildtool_depend>.
- Add <exec_depend>rosidl_default_runtime</exec_depend>.
- For each dependent message package, add <depend>message_package</depend>.

In your CMakeLists.txt:

- Start by enabling C++14

```cmake
set(CMAKE_CXX_STANDARD 14)
```

- Add find_package(rosidl_default_generators REQUIRED)
- For each dependent message package, add find_package(message_package REQUIRED) and replace the CMake function call to generate_messages with rosidl_generate_interfaces.

This will replace add_message_files and add_service_files listing of all the message and service files, which can be removed.

Build system

The build system in ROS 2 is called ament and the build tool is colcon. Ament is built on CMake: ament_cmake provides CMake functions to make writing CMakeLists.txt files easier.
**Build tool**

Instead of using `catkin_make`, `catkin_make_isolated` or `catkin build` ROS 2 uses the command line tool `colcon` to build and install a set of packages.

**Pure Python package**

If the ROS 1 package uses CMake only to invoke the `setup.py` file and does not contain anything beside Python code (e.g. also no messages, services, etc.) it should be converted into a pure Python package in ROS 2:

- Update or add the build type in the `package.xml` file:

  ```xml
  <export>
  <build_type>ament_python</build_type>
  </export>
  ```

- Remove the `CMakeLists.txt` file
- Update the `setup.py` file to be a standard Python setup script

ROS 2 supports Python 3 only. While each package can choose to also support Python 2 it must invoke executables with Python 3 if it uses any API provided by other ROS 2 packages.

**Update the `CMakeLists.txt` to use ament_cmake**

Apply the following changes to use `ament_cmake` instead of `catkin`:

- Set the build type in the `package.xml` file export section:

  ```xml
  <export>
  <build_type>ament_cmake</build_type>
  </export>
  ```

- Replace the `find_package` invocation with `catkin` and the `COMPONENTS` with:

  ```
  find_package(ament_cmake REQUIRED)
  find_package(component1 REQUIRED)
  # ...
  find_package(componentN REQUIRED)
  ```

- Move and update the `catkin_package` invocation with:
  - Invoke `ament_package` instead but `after` all targets have been registered.
  - The only valid argument for `ament_package` is `CONFIG_EXTRAS`. All other arguments are covered by separate functions which all need to be invoked `before` `ament_package`:
    * Instead of passing `CATKIN_DEPENDS` ... call `ament_export_dependencies(...)` before.
    * Instead of passing `INCLUDE_DIRS` ... call `ament_export_include_directories(...)` before.
    * Instead of passing `LIBRARIES` ... call `ament_export_libraries(...)` before.
  - TODO document `ament_export_targets` (``ament_export_interfaces`` in Eloquent and older)?
- Replace the invocation of `add_message_files`, `add_service_files` and `generate_messages` with `rosidl_generate_interfaces`. 

---

1348 Chapter 4. Contributing to the documentation
The first argument is the target_name. If you’re building just one library it’s `${PROJECT_NAME}`

Followed by the list of message filenames, relative to the package root.

* If you will be using the list of filenames multiple times, it is recommended to compose a list of message files and pass the list to the function for clarity.

The final multi-value-keyword argument for `generate_messages` is `DEPENDENCIES` which requires the list of dependent message packages.

```
rosidl_generate_interfaces(${PROJECT_NAME}
  ${msg_files}
  DEPENDENCIES_std_msgs
)
```

• Remove any occurrences of the `devel space`. Related CMake variables like `CATKIN-devel_PREFIX` do not exist anymore.

  The `CATKIN_DEPENDS` and `DEPENDS` arguments are passed to the new function `ament_export_dependencies`.

  `CATKIN_GLOBAL_BIN_DESTINATION`: bin
  `CATKIN_GLOBALINCLUDE_DESTINATION`: include
  `CATKIN_GLOBAL_LIB_DESTINATION`: lib
  `CATKIN_GLOBAL_LIBEXEC_DESTINATION`: lib
  `CATKIN_GLOBAL_SHARE_DESTINATION`: share
  `CATKIN_PACKAGE_BIN_DESTINATION`: lib/${PROJECT_NAME}
  `CATKIN_PACKAGE_INCLUDE_DESTINATION`: include/${PROJECT_NAME}
  `CATKIN_PACKAGE_LIB_DESTINATION`: lib
  `CATKIN_PACKAGE_SHARE_DESTINATION`: share/${PROJECT_NAME}

**Unit tests**

If you are using gtest:

Replace `CATKIN_ENABLE_TESTING` with `BUILD_TESTING`. Replace `catkin_add_gtest` with `ament_add_gtest`.

```bash
- if (CATKIN_ENABLE_TESTING)
  - find_package(GTest REQUIRED) # or rostest
  - include_directories(${GTEST_INCLUDE_DIRS})
  - catkin_add_gtest(${PROJECT_NAME}-some-test src/test/some_test.cpp)
  - target_link_libraries(${PROJECT_NAME}-some-test
    ${PROJECT_NAME}_some_dependency
    ${catkin_LIBRARIES}
    ${GTEST_LIBRARIES})
- endif()
+ if (BUILD_TESTING)
+   find_package(ament_cmake_gtest REQUIRED)
+   ament_add_gtest(${PROJECT_NAME}-some-test src/test/test_something.cpp)
+   ament_target_dependencies(${PROJECT_NAME}-some-test
+ "rclcpp"
```

(continues on next page)
Add `<test_depend>ament_cmake_gtest</test_depend>` to your package.xml.

```xml
- <test_depend>rostest</test_depend>
+ <test_depend>ament_cmake_gtest</test_depend>
```

**Linters**

In ROS 2 we are working to maintain clean code using linters. The styles for different languages are defined in our Developer Guide `<Developer-Guide>`.

If you are starting a project from scratch it is recommended to follow the style guide and turn on the automatic linter unit tests by adding these lines just below `if(BUILD_TESTING)` (until alpha 5 this was `AMENT_ENABLE_TESTING`).

```cmake
find_package(ament_lint_auto REQUIRED)
ament_lint_auto_find_test_dependencies()
```

You will also need to add the following dependencies to your package.xml:

```xml
<test_depend>ament_lint_auto</test_depend>
<test_depend>ament_lint_common</test_depend>
```

**Continue to use catkin in CMake**

ROS 2 uses ament as the build system but for backward compatibility ROS 2 has a package called `catkin` which provides almost the same API as catkin in ROS 1. In order to use this backward compatibility API the `CMakeLists.txt` must only be updated to call the function `catkin_ament_package()` after all targets.

**NOTE:** This has not been implemented yet and is only an idea at the moment. Due to the number of changes related to dependencies it has not yet been decided if this compatibility API is useful enough to justify the effort.

**Update source code**

**Messages, services, and actions**

The namespace of ROS 2 messages, services, and actions use a subnamespace (`msg`, `srv`, or `action`, respectively) after the package name. Therefore an include looks like: `#include <my_interfaces/msg/my_message.hpp>`. The C++ type is then named: `my_interfaces::msg::MyMessage`.

Shared pointer types are provided as typedefs within the message structs: `my_interfaces::msg::MyMessage::SharedPtr` as well as `my_interfaces::msg::MyMessage::ConstSharedPtr`.

For more details please see the article about the generated C++ interfaces.

The migration requires includes to change by:

- inserting the subfolder `msg` between the package name and message datatype
- changing the included filename from CamelCase to underscore separation
• changing from *.h to *.hpp

```cpp
// ROS 1 style is in comments, ROS 2 follows, uncommented.
// #include <geometry_msgs/PointStamped.h>
#include <geometry_msgs/msg/point_stamped.hpp>

// geometry_msgs::PointStamped point_stamped;
geometry_msgs::msg::PointStamped point_stamped;
```

The migration requires code to insert the msg namespace into all instances.

**Use of service objects**

Service callbacks in ROS 2 do not have boolean return values. Instead of returning false on failures, throwing exceptions is recommended.

```cpp
// ROS 1 style is in comments, ROS 2 follows, uncommented.
// #include "nav_msgs/GetMap.h"
#include "nav_msgs/srv/get_map.hpp"

// bool service_callback(
//   nav_msgs::GetMap::Request & request,
//   nav_msgs::GetMap::Response & response)
void service_callback(
  const std::shared_ptr<nav_msgs::srv::GetMap::Request> request,
  std::shared_ptr<nav_msgs::srv::GetMap::Response> response)
{
  // ...
  // return true; // or false for failure
}
```

**Usages of ros::Time**

For usages of ros::Time:

• Replace all instances of ros::Time with rclcpp::Time

• If your messages or code makes use of std_msgs::Time:
  – Convert all instances of std_msgs::Time to builtin_interfaces::msg::Time
  – Convert all #include "std_msgs/time.h" to #include "builtin_interfaces/msg/time.hpp"
  – Convert all instances using the std_msgs::Time field nsec to the builtin_interfaces::msg::Time field nanosec
Usages of ros::Rate

There is an equivalent type rclcpp::Rate object which is basically a drop in replacement for ros::Rate.

ROS client library

Python migration guide from ROS 1

Node Initialization

In ROS 1:

```python
rospy.init_node('asdf')
rospy.loginfo('Created node')
```

In ROS 2:

```python
rclpy.init(args=sys.argv)
node = rclpy.create_node('asdf')
node.get_logger().info('Created node')
```

ROS Parameters

In ROS 1:

```python
port = rospy.get_param('port', '/dev/ttyUSB0')
assert isinstance(port, str), 'port parameter must be a str'

baudrate = rospy.get_param('baudrate', 115200)
assert isinstance(baudrate, int), 'baudrate parameter must be an integer'

rospy.logwarn('port: ' + port)
```

In ROS 2:

```python
port = node.declare_parameter('port', '/dev/ttyUSB0').value
assert isinstance(port, str), 'port parameter must be a str'

baudrate = node.declare_parameter('baudrate', 115200).value
assert isinstance(baudrate, int), 'baudrate parameter must be an integer'

node.get_logger().warn('port: ' + port)
```
Creating a Publisher

In ROS 1:

```python
pub = rospy.Publisher('chatter', String)
```

In ROS 2:

```python
pub = node.create_publisher(String, 'chatter')
```

Creating a Subscriber

In ROS 1:

```python
sub = rospy.Subscriber('chatter', String, callback)
```

In ROS 2:

```python
sub = node.create_subscription(String, 'chatter', callback)
```

Creating a Service

In ROS 1:

```python
srv = rospy.Service('add_two_ints', AddTwoInts, add_two_ints_callback)
```

In ROS 2:

```python
srv = node.create_service(AddTwoInts, 'add_two_ints', add_two_ints_callback)
```

Creating a Service Client

In ROS 1:

```python
rospy.wait_for_service('add_two_ints')
add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
resp = add_two_ints(req)
```

In ROS 2:

```python
add_two_ints = node.create_client(AddTwoInts, 'add_two_ints')
while not add_two_ints.wait_for_service(timeout_sec=1.0):
    node.get_logger().info('service not available, waiting again...')
resp = add_two_ints.call_async(req)
rclpy.spin_until_future_complete(node, resp)
```

NOTE: Others to be written
Boost

Much of the functionality previously provided by Boost has been integrated into the C++ standard library. As such we would like to take advantage of the new core features and avoid the dependency on boost where possible.

Shared Pointers

To switch shared pointers from boost to standard C++ replace instances of:

- `#include <boost/shared_ptr.hpp>` with `#include <memory>`
- `boost::shared_ptr` with `std::shared_ptr`

There may also be variants such as `weak_ptr` which you want to convert as well.

Also it is recommended practice to use `using` instead of `typedef`. `using` has the ability to work better in templated logic. For details see here

Thread/Mutexes

Another common part of boost used in ROS codebases are mutexes in `boost::thread`.

- Replace `boost::mutex::scoped_lock` with `std::unique_lock<std::mutex>`
- Replace `boost::mutex` with `std::mutex`
- Replace `#include <boost/thread/mutex.hpp>` with `#include <mutex>`

Unordered Map

Replace:

- `#include <boost/unordered_map.hpp>` with `#include <unordered_map>`
- `boost::unordered_map` with `std::unordered_map`

function

Replace:

- `#include <boost/function.hpp>` with `#include <functional>`
- `boost::function` with `std::function`

Parameters

In ROS 1, parameters are associated with a central server that allowed retrieving parameters at runtime through the use of the network APIs. In ROS 2, parameters are associated per node and are configurable at runtime with ROS services.

- See [ROS 2 Parameter design document](#) for more details about the system model.
- See [ROS 2 CLI usage](#) for a better understanding of how the CLI tools work and its differences with ROS 1 tooling.
- See [Migrating YAML parameter files from ROS 1 to ROS 2](#) to see how YAML parameter files are parsed in ROS 2 and their differences with ROS implementation.
Launch files

While launch files in ROS 1 are always specified using .xml files, ROS 2 supports Python scripts to enable more flexibility (see launch package) as well as XML and YAML files. See separate tutorial <../../How-To-Guides/Launch-files-migration-guide> on migrating launch files from ROS 1 to ROS 2.

Example: Converting an existing ROS 1 package to use ROS 2

Let's say that we have simple ROS 1 package called talker that uses roscpp in one node, called talker. This package is in a catkin workspace, located at ~/ros1_talker.

The ROS 1 code

Here's the directory layout of our catkin workspace:

```bash
$ cd ~/ros1_talker
$ find .
.
./src
./src/talker
./src/talker/package.xml
./src/talker/CMakeLists.txt
./src/talker/talker.cpp
```

Here is the content of those three files:

src/talker/package.xml:

```xml
<package>
  <name>talker</name>
  <version>0.0.0</version>
  <description>talker</description>
  <maintainer email="gerkey@osrfoundation.org">Brian Gerkey</maintainer>
  <license>Apache 2.0</license>
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>std_msgs</build_depend>
  <run_depend>roscpp</run_depend>
  <run_depend>std_msgs</run_depend>
</package>
```

src/talker/CMakeLists.txt:

```cmake
# CMakeLists.txt
# Copyright (c) 2015, Brian Gerkey
# All rights reserved.
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions are met:
# 1. Redistributions of source code must retain the above copyright notice,
#    this list of conditions and the following disclaimer.
# 2. Redistributions in binary form must reproduce the above copyright notice,
#    this list of conditions and the following disclaimer in the documentation
#    and/or other materials provided with the distribution.
# 3. Neither the name of the Brian Gerkey nor the names of its contributors
#    may be used to endorse or promote products derived from this software
#    without specific prior written permission.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
# "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
# TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
# PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
# HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
# SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED
# TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
# PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY
# OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
# NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
# SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

cmake_minimum_required(VERSION 2.8.3)
project(talker)
find_package(catkin REQUIRED COMPONENTS roscpp std_msgs)
catkin_package()
include_directories(${catkin_INCLUDE_DIRS})
add_executable(talker talker.cpp)
target_link_libraries(talker ${(catkin_LIBRARIES)})
install(TARGETS talker
  RUNTIME DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION})
```
src/talker/talker.cpp:

```cpp
#include <sstream>
#include "ros/ros.h"
#include "std_msgs/String.h"

int main(int argc, char **argv)
{
    ros::init(argc, argv, "talker");
    ros::NodeHandle n;
    ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
    ros::Rate loop_rate(10);
    int count = 0;
    std_msgs::String msg;
    while (ros::ok())
    {
        std::stringstream ss;
        ss << "hello world " << count++;
        msg.data = ss.str();
        ROS_INFO("%s", msg.data.c_str());
        chatter_pub.publish(msg);
        ros::spinOnce();
        loop_rate.sleep();
    }
    return 0;
}
```

Building the ROS 1 code

We source an environment setup file (in this case for Jade using bash), then we build our package using `catkin_make install`:

```
./opt/_ros/jade/setup.bash
cd ~/ros1_talker
catkin_make install
```

Running the ROS 1 node

If there's not already one running, we start a `roscore`, first sourcing the setup file from our `catkin` install tree (the system setup file at `/opt/ROS/jade/setup.bash` would also work here):

```
./~ros1_talker/install/setup.bash
roscore
```

In another shell, we run the node from the `catkin` install space using `rosrun`, again sourcing the setup file first (in this case it must be the one from our workspace):

```
./~ros1_talker/install/setup.bash
rosrun talker talker
```
Migrating to ROS 2

Let’s start by creating a new workspace in which to work:

```bash
mkdir ~/ros2_talker
cd ~/ros2_talker
```

We’ll copy the source tree from our ROS 1 package into that workspace, where we can modify it:

```bash
mkdir src
cp -a ~/ros1_talker/src/talker src
```

Now we’ll modify the C++ code in the node. The ROS 2 C++ library, called rclcpp, provides a different API from that provided by roscpp. The concepts are very similar between the two libraries, which makes the changes reasonably straightforward to make.

Included headers

In place of `ros/ros.h`, which gave us access to the roscpp library API, we need to include `rclcpp/rclcpp.hpp`, which gives us access to the rclcpp library API:

```cpp
#include "ros/ros.h"
#include "rclcpp/rclcpp.hpp"
```

To get the `std_msgs/String` message definition, in place of `std_msgs/String.h`, we need to include `std_msgs/msg/string.hpp`:

```cpp
#include "std_msgs/String.h"
#include "std_msgs/msg/string.hpp"
```

Changing C++ library calls

Instead of passing the node’s name to the library initialization call, we do the initialization, then pass the node name to the creation of the node object (we can use the `auto` keyword because now we’re requiring a C++14 compiler):

```cpp
// rclcpp::init(argc, argv);
// auto node = rclcpp::Node::make_shared("talker");

auto node = rclcpp::Node::make_shared("talker");
```

The creation of the publisher and rate objects looks pretty similar, with some changes to the names of namespace and methods.

```cpp
// ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
// auto chatter_pub = node->create_publisher<std_msgs::msg::String>("chatter", 1000);

auto chatter_pub = node->create_publisher<std_msgs::msg::String>("chatter", 1000);
```

To further control how message delivery is handled, a quality of service (QoS) profile could be passed in. The default profile is `rmw_qos_profile_default`. For more details, see the design document and concept overview <../../Concepts/Intermediate/About-Quality-of-Service-Settings>.
The creation of the outgoing message is different in the namespace:

```cpp
// std_msgs::String msg;
std_msgs::msg::String msg;
```

In place of `ros::ok()`, we call `rclcpp::ok()`:

```cpp
// while (ros::ok())
while (rclcpp::ok())
```

Inside the publishing loop, we access the `data` field as before:

```cpp
msg.data = ss.str();
```

To print a console message, instead of using `ROS_INFO()`, we use `RCLCPP_INFO()` and its various cousins. The key difference is that `RCLCPP_INFO()` takes a Logger object as the first argument.

```cpp
// ROS_INFO("%s", msg.data.c_str());
RCLCPP_INFO(node->get_logger(), "%s
", msg.data.c_str());
```

Publishing the message is the same as before:

```cpp
chatter_pub->publish(msg);
```

Spinning (i.e., letting the communications system process any pending incoming/outgoing messages) is different in that the call now takes the node as an argument:

```cpp
// ros::spinOnce();
rclcpp::spin_some(node);
```

Sleeping using the rate object is unchanged.

Putting it all together, the new `talker.cpp` looks like this:

```cpp
#include <sstream>
#include "ros/ros.h"
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/String.h"
#include "std_msgs/msg/string.hpp"

int main(int argc, char **argv)
{
    // ros::init(argc, argv, "talker");
    // ros::NodeHandle n;
    rclcpp::init(argc, argv);
    auto node = rclcpp::Node::make_shared("talker");
    // ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
    // ros::Rate loop_rate(10);
    auto chatter_pub = node->create_publisher<std_msgs::msg::String>("chatter", 1000);
    rclcpp::Rate loop_rate(10);
    int count = 0;
    // std_msgs::String msg;
    std_msgs::msg::String msg;
    // while (ros::ok())
    while (rclcpp::ok())
    {
    (continues on next page)
std::stringstream ss;
ss << "hello world " << count++;
msg.data = ss.str();
ROS_INFO("%s", msg.data.c_str());
RCLCPP_INFO(node->get_logger(), "%s
", msg.data.c_str());
chatter_pub->publish(msg);
chatter_pub->get_logger()->info("%s
");
ROS_INFO("%s", msg.data.c_str());
chatter_pub->publish(msg);
<package format="2">
  <name>talker</name>
  <version>0.0.0</version>
  <description>talker</description>
  <maintainer email="gerkey@osrfoundation.org">Brian Gerkey</maintainer>
  <license>Apache License 2.0</license>
</package>

<!-- <buildtool_depend>catkin</buildtool_depend> -->
<buildtool_depend>ament_cmake</buildtool_depend>
<!-- <build_depend>roscpp</build_depend> -->
<!-- <run_depend>roscpp</run_depend> -->
<!-- <run_depend>std_msgs</run_depend> -->
<depend>rclcpp</depend>
<depend>std_msgs</depend>
<export>
  <build_type>ament_cmake</build_type>
</export>
</package>

TODO: show simpler version of this file just using the ``<depend>`` tag, which is enabled by version 2 of the package format (also supported in ``catkin`` so, strictly speaking, orthogonal to ROS 2).

### Changing the CMake code

ROS 2 relies on a higher version of CMake:

```cmake
#cmake_minimum_required(VERSION 2.8.3)
cmake_minimum_required(VERSION 3.5)
```

ROS 2 relies on the C++14 standard. Depending on what compiler you’re using, support for C++14 might not be enabled by default. Using gcc 5.3 (which is what is used on Ubuntu Xenial), we need to enable it explicitly, which we do by adding this line near the top of the file:

```cmake
set(CMAKE_CXX_STANDARD 14)
```

The preferred way to work on all platforms is this:

```cmake
if(NOT CMAKE_CXX_STANDARD)
  set(CMAKE_CXX_STANDARD 14)
endif()
if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()
```

Using catkin, we specify the packages we want to build against by passing them as COMPONENTS arguments when initially finding catkin itself. With ament_cmake, we find each package individually, starting with ament_cmake:

```cmake
#find_package(catkin REQUIRED COMPONENTS roscpp std_msgs)
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)
```

System dependencies can be found as before:
find_package(Boost REQUIRED COMPONENTS system filesystem thread)

We call `catkin_package()` to auto-generate things like CMake configuration files for other packages that use our package. Whereas that call happens before specifying targets to build, we now call the analogous `ament_package()` after the targets:

```
# catkin_package()
# At the bottom of the file:
ament_package()
```

The only directories that need to be manually included are local directories and dependencies that are not ament packages:

```
#include_directories(${catkin_INCLUDE_DIRS})
include_directories(include ${Boost_INCLUDE_DIRS})
```

A better alternative is to specify include directories for each target individually, rather than including all the directories for all targets:

```
target_include_directories(target PUBLIC include ${Boost_INCLUDE_DIRS})
```

Similar to how we found each dependent package separately, we need to link each one to the build target. To link with dependent packages that are ament packages, instead of using `target_link_libraries()`, `ament_target_dependencies()` is a more concise and more thorough way of handling build flags. It automatically handles both the include directories defined in `_INCLUDE_DIRS` and linking libraries defined in `_LIBRARIES`.

```
#target_link_libraries(talker ${catkin_LIBRARIES})
ament_target_dependencies(talker rclcpp std_msgs)
```

To link with packages that are not ament packages, such as system dependencies like Boost, or a library being built in the same CMakeLists.txt, use `target_link_libraries()`:

```
target_link_libraries(target ${Boost_LIBRARIES})
```

For installation, `catkin` defines variables like `CATKIN_PACKAGE_BIN_DESTINATION`. With `ament_cmake`, we just give a path relative to the installation root, like `bin` for executables:

```
#install(TARGETS talker
# RUNTIME DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION})
install(TARGETS talker
DESTINATION lib/${PROJECT_NAME})
```

Optionally, we can install and export the included directories for downstream packages:

```
install(DIRECTORY include/
DESTINATION include)
ament_export_include_directories(include)
```

Optionally, we can export dependencies for downstream packages:

```
ament_export_dependencies(std_msgs)
```

Putting it all together, the new CMakeLists.txt looks like this:
#cmake_minimum_required(VERSION 2.8.3)
cmake_minimum_required(VERSION 3.5)
project(talker)
if(NOT CMAKE_CXX_STANDARD)
    set(CMAKE_CXX_STANDARD 14)
endif()
if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
    add_compile_options(-Wall -Wextra -Wpedantic)
endif()

#find_package(catkin REQUIRED COMPONENTS roscpp std_msgs)
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)
#catkin_package()
#include_directories(${catkin_INCLUDE_DIRS})
include_directories(include)
add_executable(talker talker.cpp)
#target_link_libraries(talker ${catkin_LIBRARIES})
ament_target_dependencies(talker
    rclcpp
    std_msgs)
#install(TARGETS talker
#    RUNTIME DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION})
install(TARGETS talker
    DESTINATION lib/${PROJECT_NAME})
install(DIRECTORY include/
    DESTINATION include)
ament_export_include_directories(include)
ament_export_dependencies(std_msgs)
ament_package()

TODO: Show what this would look like with `ament_auto`.

**Building the ROS 2 code**

We source an environment setup file (in this case the one generated by following the ROS 2 installation tutorial, which builds in `~/ros2_ws`, then we build our package using colcon build:

```
. ~/ros2_ws/install/setup.bash
cd ~/ros2_talker
colcon build
```
Running the ROS 2 node

Because we installed the talker executable into bin, after sourcing the setup file, from our install tree, we can invoke it by name directly (also, there is not yet a ROS 2 equivalent for rosrn):

```
. ~/ros2_ws/install/setup.bash
talker
```

Update scripts

ROS CLI arguments

Since ROS Eloquent <../../Releases/Release-Eloquent-Elusor>, ROS arguments should be scoped with --ros-args and a trailing -- (the trailing double dash may be elided if no arguments follow it).

Remapping names is similar to ROS 1, taking on the form from:=to, except that it must be preceded by a --remap (or -r) flag. For example:

```
ros2 run some_package some_ros_executable --ros-args -r foo:=bar
```

We use a similar syntax for parameters, using the --param (or -p) flag:

```
ros2 run some_package some_ros_executable --ros-args -p my_param:=value
```

Note, this is different than using a leading underscore in ROS 1.

To change a node name use __node (the ROS 1 equivalent is __name):

```
ros2 run some_package some_ros_executable --ros-args -r __node:=new_node_name
```

Note the use of the -r flag. The same remap flag is needed for changing the namespace __ns:

```
ros2 run some_package some_ros_executable --ros-args -r __ns:=/new/namespace
```

There is no equivalent in ROS 2 for the following ROS 1 keys:

- __log
- __ip
- __hostname
- __master

For more information, see the design document.
Quick reference

<table>
<thead>
<tr>
<th>Feature</th>
<th>ROS 1</th>
<th>ROS 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>remapping</td>
<td>foo:=bar</td>
<td>-r foo:=bar</td>
</tr>
<tr>
<td>parameters</td>
<td>_foo:=bar</td>
<td>-p foo:=bar</td>
</tr>
<tr>
<td>node name</td>
<td>__name:=foo</td>
<td>-r __node:=foo</td>
</tr>
<tr>
<td>namespace</td>
<td>__ns:=foo</td>
<td>-r __ns:=foo</td>
</tr>
</tbody>
</table>

More examples and tools

- Launch File migrator that converts a ROS 1 XML launch file to a ROS 2 Python launch file: https://github.com/aws-robotics/ros2-launch-file-migrator
- Amazon has exposed their tools for porting ROS 1 robots to ROS 2 https://github.com/awslabs/ros2-migration-tools/tree/master/porting_tools

Licensing

In ROS 2 our recommended license is the Apache 2.0 License. In ROS 1 our recommended license was the 3-Clause BSD License.

For any new project we recommend using the Apache 2.0 License, whether ROS 1 or ROS 2.

However, when migrating code from ROS 1 to ROS 2 we cannot simply change the license. The existing license must be preserved for any preexisting contributions.

To that end if a package is being migrated we recommend keeping the existing license and continuing to contribute to that package under the existing OSI license, which we expect to be the BSD license for core elements.

This will keep things clear and easy to understand.

Changing the License

It is possible to change the license, however you will need to contact all the contributors and get permission. For most packages this is likely to be a significant effort and not worth considering. If the package has a small set of contributors then this may be feasible.

ROS Build Farms

The ROS build farms are an important infrastructure to support the ROS ecosystem, provided and maintained by Open Robotics. They provide building of source and binary packages, continuous integration, testing, and analysis for ROS 1 and ROS 2 packages. There are two hosted instance for open source packages:
1. https://build.ros.org/ for ROS 1 packages
2. https://build.ros2.org/ for ROS 2 packages

If you are going to use any of the provided infrastructure please consider signing up for the build farm discussion forum in order to receive notifications, e.g., about any upcoming changes.

**Jobs and Deployment**

The ROS build farms perform several different jobs. For each job type you will find a detailed description of what they do and how they work:

- **release jobs** generate binary packages, e.g., debian packages
- **devel jobs** build and test ROS packages within a single repository on a polling basis
- **pull_request jobs** build and test ROS packages within a single repository triggered by webhooks
- **CI jobs** build and test ROS packages across repositories with the option of using artifacts from other CI jobs to speed up the build
- **doc jobs** generate the API documentation of packages and extract information from the manifests
- **miscellaneous jobs** perform maintenance tasks and generate informational data to visualize the status of the build farm and its generated artifacts

**Creation and Deployment**

The above jobs are created and deployed when packages are bloomed, i.e. released for ROS 1 or ROS 2. Once blooming is successful and a package is incorporated in one of the ROS distributions (via pull request to rosdistro), the according jobs will be spawned. The names of the jobs encode their type and purpose:

- **release jobs:**
  - `{distro}src__{package}__{platform}__source` build source packages of releases
  - `{distro}bin__{package}__{platform}__binary` build binary packages of releases

  For instance, the binary packaging job of rclcpp on ROS 2 Iron (running on Ubuntu Jammy amd64) is named `Ibin_uJ64__rclcpp__ubuntu_jammy_amd64__binary`.

- **devel jobs:**
  - `{distro}dev__{package}__{platform}` perform a CI build for the releasing branch

- **pull_request jobs:**
  - `{distro}pr__{package}__{platform}` perform a CI build for a pull request

  For instance, the PR job for rclcpp on ROS 2 Iron (running on Ubuntu Jammy amd64) is named `Ipr__rclcpp__ubuntu_jammy_amd64`.

---

1 `{distro}` is the first letter of the ROS distribution, `{platform} ((platf)) names the platform the package is built for (and its short code), and `{package}` is the name of the ROS package being built.
Execution

Execution of the jobs depends on the type of the job:

- **devel jobs** will be triggered every time a commit is done to the respective branch polling based on a configured frequency.
- **pull_request jobs** will be triggered by webhooks from respective pull request of the upstream repository.
- **release jobs** will be triggered once every time a new package version is released, i.e., a new rosdistro pull request was accepted for this package. The source jobs are triggered by a version change in the rosdistro distribution file, the binary jobs are triggered by their source counterpart.

Frequency Asked Questions (FAQ) and Troubleshooting

1. **I get Jenkins mails from failing build farm jobs. What do I do?**

   Go to the job that raised the issue. You find the link on top of the Jenkins email. Once you followed the link to the build job, click **Console Output** on the left, then click **Full Log**. This will give you the full console output of the failing build. Try to find the top-most error as it is usually the most important and other errors might be follow-ups.

   The bottom of the email might read 'apt-src build [...] failed. This is usually because of an error building the package. This usually hints at missing dependencies, see 2.

2. **I seem to be missing a dependency, how do I find out which one?**

   You basically have two options, a. is easier but may take several iterations, b. is more elaborate and gives you the full insight as well as local debugging.

   a) Inspect the release job that raised the issue (see 1.) and localize the cmake dependency issue. To do so, browse to the cmake section, e.g., navigate to the **build binarydeb** section through the menu on the left in case of a ubuntu/debian build job. The **CMake Error** will typically hint at a dependency required by the cmake configuration but missing in the package manifest. Once you have fixed the dependency in the manifest, do a new release of your package and wait for feedback from the build farms or...

   b) To get the full insight and faster, local debugging, you can run the release jobs locally. This allows to iterate the manifest locally until all dependencies are fixed.

3. **Why do release jobs fail when devel jobs / my github actions / my local builds succeed?**

   There are several potential reasons for this. First, release jobs build against a minimal ROS installation to check if all dependencies are properly declared in the package manifest. Devel jobs / github actions / local builds may be performed in an environment that has the dependencies already installed, therefore does not notice dependency issues. Second, they might build different versions of the source code. While devel jobs / github actions / local builds usually build the latest version from the upstream repository, release jobs build the source code of the latest release, i.e., the source code in the respective upstream branches of the release repository.

---

2 The **upstream** repository is the repository containing the original source code of the respective ROS 1 / ROS 2 package.

3 The **release** repository is the repository that ROS 2 infrastructure uses for releasing packages, see [https://github.com/ros2-gbp/](https://github.com/ros2-gbp/).
Further Reading

The following links provide more details and insights into the build farms:

- http://wiki.ros.org/regression_tests#Setting_up_Your_Computer_for_Prerelease
- http://wiki.ros.org/buildfarm - ROS wiki entry for the ROS 1 build farm (partially outdated)
- https://github.com/ros-infrastructure/cookbook-ros-buildfarm - Installs and configures ROS build farm machines

Windows Tips and Tricks

Table of Contents

- Maximum Path Length
- Symbol Visibility
  - Visibility Control Headers
  - WINDOWS_EXPORT_ALL_SYMBOLS Target Property
- Debug builds
- Forward-slash vs. back-slash
- Patching vendored packages
- Windows slow timers (slowness in general)
- Shells

ROS 2 supports Windows 10 as a Tier 1 platform, which means that all code that goes into the ROS 2 core must support Windows. For those used to traditional development on Linux or other Unix-like systems, developing on Windows can be a bit of a challenge. This document aims to lay out some of those differences.

Maximum Path Length

By default, Windows has a maximum path length of 260 characters. Practically speaking, 4 of those characters are always used by the drive letter, colon, initial backslash, and final NULL character. That means that only 256 characters are available for the sum of all parts of the path. This has two practical consequences for ROS 2:

- Some of the ROS 2 internal path names are fairly long. Because of this, we always recommend using a short path name for the root of your ROS 2 directory, like C:\dev.
- When building ROS 2 from source, the default isolated build mode of colcon can generate very long path names. To avoid these very long path names, use --merge-install when building on Windows.

Note: It is possible to change Windows to have much longer maximum path lengths. See this article for more information.
Symbol Visibility

The Microsoft Visual C++ Compiler (MSVC) exposes symbols from a Dynamic Link Library (DLL) only if they are explicitly exported. The clang and gcc compilers have an option to do the same, but it is off by default. As a result, when a library previously built on Linux is built on Windows, other libraries may be unable to resolve the external symbols. Below are examples of common error messages which can be caused by symbols not being exposed:

```
error C2448: '__attribute__': function-style initializer appears to be a function declaration
'visibility': identifier not found
```

Symbol Visibility also impacts binary loading. If you are finding that a composable node does not run or a Qt Visualizer isn't working, it may be that the hosting process can not find an expected symbol export from the binary. To diagnose this on Windows, the Windows developer tools includes a program called Gflags to enable various options. One of those options is called Loader Snaps which enables you to detect load failures while debugging. Please visit the Microsoft Documentation for more information on Gflags and Loaders snaps.

Two solutions to exportsymbolson Windows are Visibility Control Headers and the WINDOWS_EXPORT_ALL_SYMBOLS property. Microsoft recommends ROS developers use Visibility Control Headers to control the export of symbols from a binary. Visibility Control Headers provide more control over the symbol export macro and offer other benefits including smaller binary size and reduced link times.

Visibility Control Headers

The purpose of Visibility Control Headers headers is to define a macro for each shared library which correctly declares symbols as dllimport or dllexport. This is decided based on whether the library is being consumed or being built itself. The logic in the macro also takes the compiler into account and includes logic to select the appropriate syntax. The GCC visibility documentation includes step by step instructions for adding explicit symbol visibility to a library “yielding the highest quality code with the greatest reductions in binary size, load times and link times”. A header named `visibility_control.h` can be placed in the `includes` folder for each library as shown in the example below. The example below shows how a visibility control header would be added for a `my_lib` library with a class called `example_class`. Add a visibility header to the include folder for the library. The boiler plate logic is used with the library name used in the macro to make it unique in the project. In another library, `MY_LIB` would be replaced with the library name.

```
#ifndef MY_LIB__VISIBILITY_CONTROL_H_
#define MY_LIB__VISIBILITY_CONTROL_H_
#if defined _WIN32 || defined __CYGWIN__
#elif defined __GNUC__
    #define MY_LIB_EXPORT __attribute__ ((dlllexport))
    #define MY_LIB_IMPORT __attribute__ ((dllimport))
#else
    #define MY_LIB_EXPORT __declspec(dllexport)
    #define MY_LIB_IMPORT __declspec(dllimport)
#endif
#if defined MY_LIB_BUILDING_LIBRARY
    #define MY_LIB_PUBLIC MY_LIB_EXPORT
#endif
```

(continues on next page)
#define MY_LIB_PUBLIC MY_LIB_IMPORT
#endif
#define MY_LIB_PUBLIC_TYPE MY_LIB_PUBLIC
#define MY_LIB_LOCAL
#endif
// Linux visibility settings
#define MY_LIB_PUBLIC_TYPE
#endif
#endif // MY_LIB_VISIBILITY_CONTROL_H_

For a complete example of this header, see rviz_rendering.

To use the macro, add MY_LIB_PUBLIC before symbols which need to be visible to external libraries. For example:

```cpp
Class MY_LIB_PUBLIC example_class {}
MY_LIB_PUBLIC void example_function (){}
```

In order to build your library with correctly exported symbols, you will need to add the following to your CMakeLists.txt file:

```cmake
target_compile_definitions(${PROJECT_NAME}
PRIVATE "MY_LIB_BUILDING_LIBRARY")
```

**WINDOWS_EXPORT_ALL_SYMBOLS Target Property**

CMake implements the WINDOWS_EXPORT_ALL_SYMBOLS property on Windows, which causes function symbols to be automatically exported. More detail of how it works can be found in the [WINDOWS_EXPORT_ALL_SYMBOLS CMake Documentation](#). The property can be implemented by adding the following to the CMakeLists file:

```cmake
set_target_properties(${LIB_NAME} PROPERTIES WINDOWS_EXPORT_ALL_SYMBOLS TRUE)
```

If there is more than one library in a CMakeLists file you will need to call `set_target_properties` on each of them separately.

Note that a binary on Windows can only export 65,536 symbols. If a binary exports more than that, you will get an error and should use the visibility_control headers. There is an exception to this method in the case of global data symbols. For example, a global static data member like the one below.

```cpp
class Example_class
{
public:
static const int Global_data_num;
```

In these cases `dllimport/dlexport` must be applied explicitly. This can be done using `generate_export_header` as described in the following article: [Create dlls on Windows withoutdeclspec() using new CMake export all feature](#).

Finally, it is important that the header file that exports the symbols be included into at least one of the .cpp files in the package so that the macros will get expanded and placed into the resulting binary. Otherwise the symbols will still not be callable.
Debug builds

When building in Debug mode on Windows, several very important things change. The first is that all DLLs get _d automatically appended to the library name. So if the library is called libfoo.dll, in Debug mode it will be libfoo_d.dll. The dynamic linker on Windows also knows to look for libraries of that form, so it will not find libraries without the _d prefix. Additionally, Windows turns on a whole set of compile-time and run-time checks in Debug mode that is far more strict than Release builds. For these reasons, it is a good idea to run a Windows Debug build and test on many pull requests.

Forward-slash vs. back-slash

In Windows the default path separator is a backslash (\), which differs from the forward-slash (/) used in Linux and macOS. Most of the Windows APIs can deal with either as a path separator, but this is not universally true. For instance, the cmd.exe shell can only do tab-completion when using the backslash character, not the forward-slash. For maximum compatibility on Windows, a backslash should always be used as the path separator on Windows.

Patching vendored packages

When vendoring a package in ROS 2, it is often necessary to apply a patch to fix a bug, add a feature, etc. The typical way to do this is to modify the ExternalProject_add call to add a PATCH command, using the patch executable. Unfortunately, the patch executable as delivered by chocolatey requires Administrator access to run. The workaround is to use git apply-patch when applying patches to external projects.

git apply-patch has its own issues in that it only works properly when applied to a git repository. For that reason, external projects should always use the GIT method to obtain the project and then use the PATCH_COMMAND to invoke git apply-patch.

An example usage of all of the above looks something like:

```bash
ExternalProject_Add(mylibrary-$(version)
    GIT_REPOSITORY https://github.com/lib/mylibrary.git
    GIT_TAG $(version)
    GIT_CONFIG advice.detachedHead=false
    # Suppress git update due to https://gitlab.kitware.com/cmake/cmake/-/issues/16419
    # See https://github.com/ament/uncrustify_vendor/pull/22 for details
    UPDATE_COMMAND ""
    TIMEOUT 600
    CMAKE_ARGS
        -DCMAKE_INSTALL_PREFIX=${CMAKE_CURRENT_BINARY_DIR}/${PROJECT_NAME}_install
        ${extra_cmake_args}
        -Wno-dev
        PATCH_COMMAND
            ${CMAKE_COMMAND} -E chdir <SOURCE_DIR> git apply -p1 --ignore-space-change --
            whitespace=nowarn ${CMAKE_CURRENT_SOURCE_DIR}/install-patch.diff
)
```
Windows slow timers (slowness in general)

Software running on Windows is, in general, much slower than that running on Linux. This is due to a number of factors, from the default time slice (every 20 ms, according to the documentation), to the number of anti-virus and anti-malware processes running, to the number of background processes running. Because of all of this, tests should never expect tight timing on Windows. All tests should have generous timeouts, and only expect events to happen eventually (this will also prevent tests from being flakey on Linux).

Shells

There are two main command-line shells on Windows: the venerable `cmd.exe`, and PowerShell.

`cmd.exe` is the command shell that most closely emulates the old DOS shell, though with greatly enhanced capabilities. It is completely text based, and only understands DOS/Windows batch files.

PowerShell is the newer, object-based shell that Microsoft recommends for most new applications. It understands `.ps1` files for configuration.

ROS 2 supports both `cmd.exe` and PowerShell, so any changes (especially to things like `ament` or `colcon`) should be tested on both.

Contributing to ROS 2 Documentation

Contributions to this site are most welcome. This page explains how to contribute to ROS 2 Documentation. Please be sure to read the below sections carefully before contributing.

The site is built using Sphinx, and more particularly using Sphinx multiversion.
Branch structure

The source code of documentation is located in the ROS 2 Documentation GitHub repository. This repository is set up with one branch per ROS 2 distribution to handle differences between the distributions. If a change is common to all ROS 2 distributions, it should be made to the rolling branch (and then will be backported as appropriate). If a change is specific to a particular ROS 2 distribution, it should be made to the respective branch.

Source structure

The source files for the site are all located under the source subdirectory. Templates for various sphinx plugins are located under source/_templates. The root directory contains configuration and files required to locally build the site for testing.

Building the site locally

Start by installing requirements located in the requirements.txt file:

Linux

The next command does a user-specific install, which requires ~/.local/bin/ to be added to $PATH:

```
pip3 install --user --upgrade -r requirements.txt
```

macOS

```
pip3 install --user --upgrade -r requirements.txt
```

Windows

```
python -m pip install --user --upgrade -r requirements.txt
```

In order for Sphinx to be able to generate diagrams, the dot command must be available.

Linux

```
sudo apt update ; sudo apt install graphviz
```

macOS

```
brew install graphviz
```

Windows

Download an installer from the Graphviz Download page and install it. Make sure to allow the installer to add it to the Windows %PATH%, otherwise Sphinx will not be able to find it.
Building the site for one branch

To build the site for just this branch, type `make html` at the top-level of the repository. This is the recommended way to test out local changes.

```
make html
```

The build process can take some time. To see the output, open `build/html/index.html` in your browser.

You can also run the documentation tests locally (using `doc8`) with the following command:

```
make test
```

Building the site for all branches

To build the site for all branches, type `make multiversion` from the `rolling` branch. This has two drawbacks:

1. The multiversion plugin doesn’t understand how to do incremental builds, so it always rebuilds everything. This can be slow.
2. When typing `make multiversion`, it will always check out exactly the branches listed in the `conf.py` file. That means that local changes will not be shown.

To show local changes in the multiversion output, you must first commit the changes to a local branch. Then you must edit the `conf.py` file and change the `smv_branch_whitelist` variable to point to your branch.

Checking for broken links

To check for broken links on the site, run:

```
make linkcheck
```

This will check the entire site for broken links, and output the results to the screen and `build/linkcheck`.

Writing pages

The ROS 2 documentation website uses the reStructuredText format, which is the default plaintext markup language used by Sphinx. This section is a brief introduction to reStructuredText concepts, syntax, and best practices.

You can refer to reStructuredText User Documentation for a detailed technical specification.

Table of Contents

There are two types of directives used for the generation of a table of contents, .. toctree:: and .. contents::.

The .. toctree:: is used in top-level pages like `Tutorials.rst` to set ordering and visibility of its child pages. This directive creates both left navigation panel and in-page navigation links to the child pages listed. It helps readers to understand the structure of separate documentation sections and navigate between pages.

```
.. toctree::
   :maxdepth: 1
```

4.8. ROS 2 Documentation
The .. contents:: directive is used for the generation of a table of contents for that particular page. It parses all present headings in a page and builds an in-page nested table of contents. It helps readers to see an overview of the content and navigate inside a page.

The .. contents:: directive supports the definition of maximum depth of nested sections. Using `:depth:` 2 will only show Sections and Subsections in the table of contents.

```
.. contents:: Table of Contents
   :depth: 2
   :local:
```

**Headings**

There are four main Heading types used in the documentation. Note that the number of symbols has to match the length of the title.

```
Page Title Header
=================

Section Header
--------------

2 Subsection Header
^^^^^^^^^^^^^^^^^^

2.4 Subsubsection Header
~~~~~~~~~~~~~~~~~~~~~~~~
```

We usually use one digit for numbering subsections and two digits (dot separated) for numbering subsubsections in Tutorials and How-To-Guides.

**Lists**

Stars * are used for listing unordered items with bullet points and number sign #. is used for listing numbered items. Both of them support nested definitions and will render accordingly.

```
* bullet point
  * bullet point nested
  * bullet point nested

* bullet point
```

```
#. first listed item
#. second listed item
```
Code Formatting

In-text code can be formatted using backticks for showing highlighted code.

```
In-text code can be formatted using `backticks` for showing `highlighted` code.
```

Code blocks inside a page need to be captured using .. code-block:: directive. .. code-block:: supports code highlighting for syntaxes like C++, YAML, console, bash, and more. Code inside the directive needs to be indented.

```
.. code-block:: C++

    int main(int argc, char** argv)
    {
        rclcpp::init(argc, argv);
        rclcpp::spin(std::make_shared<ParametersClass>());
        rclcpp::shutdown();
        return 0;
    }
```

Images

Images can be inserted using the .. image:: directive.

```
.. image:: images/turtlesim_follow1.png
```

References and Links

External links

The syntax of creating links to external web pages is shown below.

```
`ROS Docs <https://docs.ros.org>`_
```

The above link will appear as ROS Docs. Note the underscore after the final single quote.

Internal links

The :doc: directive is used to create in-text links to other pages.

```
:doc:`Quality of Service <../Tutorials/Quality-of-Service>`
```

Note that the relative path to the file is used.

The ref directive is used to make links to specific parts of a page. These could be headings, images or code sections inside the current or different page.

Definition of explicit target right before the desired object is required. In the example below, the target is defined as _talker-listener one line before the heading Try some examples.
.. _talker-listener:

Try some examples
------------------

Now the link from any page in the documentation to that header can be created.

```
:ref:`talker-listener demo <talker-listener>`
```

This link will navigate a reader to the target page with an HTML anchor link #talker-listener.

**Macros**

Macros can be used to simplify writing documentation that targets multiple distributions.

Use a macro by including the macro name in curly braces. For example, when generating the docs for Rolling on the rolling branch:

<table>
<thead>
<tr>
<th>Use</th>
<th>Becomes (for Rolling)</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{DISTRO}</code></td>
<td>rolling</td>
<td>ros-{DISTRO}-pkg</td>
</tr>
<tr>
<td><code>{DISTRO TITLE}</code></td>
<td>Rolling</td>
<td>ROS 2 <code>{DISTRO TITLE}</code></td>
</tr>
<tr>
<td><code>{DISTRO TITLE FULL}</code></td>
<td>Rolling Ridley</td>
<td>ROS 2 <code>{DISTRO TITLE FULL}</code></td>
</tr>
<tr>
<td><code>{REPOS_FILE_BRANCH}</code></td>
<td>rolling</td>
<td>git checkout <code>{REPOS_FILE_BRANCH}</code></td>
</tr>
</tbody>
</table>

The same file can be used on multiple branches (i.e., for multiple distros) and the generated content will be distro-specific.

**What to work on**

We have identified a number of tasks that could be worked on by community members: they can be listed by searching across the ROS 2 repositories for issues labeled as “help wanted”. If you see something on that list that you would like to work on, please comment on the item to let others know that you are looking into it.

We also have a label for issues that we think should be more accessible for first-time contributors, labeled “good first issue”. If you are interested in contributing to the ROS 2 project, we encourage you to take a look at those issues first. If you’d like to cast a wider net, we welcome contributions on any open issue (or others that you might propose), particularly tasks that have a milestone signifying they’re targeted for the next ROS 2 release (the milestone will be the next release’s e.g. ‘crystal’).

If you have some code to contribute that fixes a bug or improves documentation, please submit it as a pull request to the relevant repository. For larger changes, it is a good idea to discuss the proposal on the ROS 2 forum before you start to work on it so that you can identify if someone else is already working on something similar. If your proposal involves changes to the APIs, it is especially recommended that you discuss the approach before starting work.
Submitting your code changes

Code contributions should be made via pull requests to the appropriate ros2 repositories. We ask all contributors to follow the practices explained in the developer guide. Please be sure to run tests for your code changes because most packages have tests that check that the code complies with our style guidelines.

Becoming a core maintainer

The ROS 2 maintainers ensure that the project is generally making progress. The responsibilities of the maintainers include:

- Reviewing incoming code contributions for style, quality, and overall fit into the goals of the repository/ROS 2.
- Ensuring that CI continues to stay green.
- Merging pull requests that meet the quality and CI standards above.
- Addressing issues opened up by users.

Each repository in the ros2 and ament organizations has a separate set of maintainers. Becoming a maintainer of one or more of those repositories is an invitation-only process, and generally involves the following steps:

- Within the last year, have a substantial number of code contributions to the repository.
- Within the last year, do a substantial number of reviews on incoming pull requests to the repository.

Approximately every 3 months, the ROS 2 team will review the contributions in all of the repositories and send out invitations to new maintainers. Once the invitation is accepted, the new maintainer will be asked to go through a short training process on the mechanisms and policies of the ROS 2 repositories. After that training process is completed, the new maintainer will be given write access to the appropriate repositories.

Features Status

The features listed below are available in the current ROS 2 release. Unless otherwise specified, the features are available for all supported platforms (Ubuntu 22.04 (Jammy), Windows 10), DDS implementations (eProsima Fast DDS, RTI Connext DDS, and Eclipse Cyclone DDS) and programming language client libraries (C++ and Python). For planned future development, see the Roadmap.
<table>
<thead>
<tr>
<th>Functionality</th>
<th>Link</th>
<th>Fine print</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discovery, transport and serialization over DDS</td>
<td>Article</td>
<td></td>
</tr>
<tr>
<td>Support for <em>multiple DDS implementations</em>, chosen at runtime</td>
<td>Concept, How-to Guide</td>
<td>Currently Eclipse Cyclone DDS, eProsima Fast DDS, and RTI Connext DDS are fully supported.</td>
</tr>
<tr>
<td>Common core client library that is wrapped by language-specific libraries</td>
<td>Details</td>
<td></td>
</tr>
<tr>
<td>Publish/subscribe over topics</td>
<td>Sample code, Article</td>
<td></td>
</tr>
<tr>
<td>Clients and services</td>
<td>Sample code</td>
<td></td>
</tr>
<tr>
<td>Set/retrieve parameters</td>
<td>Sample code</td>
<td></td>
</tr>
<tr>
<td>ROS 1 - ROS 2 communication bridge</td>
<td>Tutorial</td>
<td>Available for topics and services, not yet available for actions.</td>
</tr>
<tr>
<td>Quality of service settings for handling non-ideal networks</td>
<td>Demo</td>
<td></td>
</tr>
<tr>
<td>Inter- and intra-process communication using the same API</td>
<td>Demo</td>
<td>Currently only in C++.</td>
</tr>
<tr>
<td>Composition of node components at compile, link, load, or run time</td>
<td>Demo</td>
<td>Currently only in C++.</td>
</tr>
<tr>
<td>Multiple executors (at level of callback groups) in same node</td>
<td>Demo</td>
<td>Only in C++.</td>
</tr>
<tr>
<td>Support for nodes with managed lifecycles</td>
<td>Demo</td>
<td>Currently only in C++.</td>
</tr>
<tr>
<td>DDS-Security support</td>
<td>Demo</td>
<td></td>
</tr>
<tr>
<td>Command-line introspection tools using an extensible framework</td>
<td>Concept</td>
<td></td>
</tr>
<tr>
<td>Launch system for coordinating multiple nodes</td>
<td>Tutorial</td>
<td></td>
</tr>
<tr>
<td>Namespace support for nodes and topics</td>
<td>Article</td>
<td></td>
</tr>
<tr>
<td>Static remapping of ROS names</td>
<td>How-to Guide</td>
<td></td>
</tr>
<tr>
<td>Demos of an all-ROS 2 mobile robot</td>
<td>Demo</td>
<td></td>
</tr>
<tr>
<td>Preliminary support for real-time code</td>
<td>Demo, demo</td>
<td>Linux only. Not available for Fast RTPS.</td>
</tr>
<tr>
<td>Preliminary support for “bare-metal” microcontrollers</td>
<td>Wiki</td>
<td></td>
</tr>
<tr>
<td>Content filtering subscription</td>
<td>Demo</td>
<td>Currently only in C++.</td>
</tr>
</tbody>
</table>

Besides core features of the platform, the biggest impact of ROS comes from its available packages. The following are a few high-profile packages which are available in the latest release:

- `gazebo_ros_pkgs`
- `image_transport`
- `navigation2`
- `rosbag2`
- `RQt`
- `RViz2`
Feature Ideas

The following are feature ideas in no specific order. This list contains features that we think are important and can make for good contributions to ROS 2. Please get in touch with us before digging into a new feature. We can offer guidance, and connect you with other developers.

Design / Concept

- IDL format
  - Leverage new features like grouping constants into enums
  - Extend usage to .idl files with just constants and/or declare parameters with ranges
  - Revisit constraints of IDL interface naming, see ros2/design#220
- Create migration plan for ROS 1 -> ROS 2 transition
- Uniqueness of node names, see ros2/design#187
- Specific “API” of a node in terms of topics / services / etc. in a descriptive format, see ros2/design#266

Infrastructure and tools

- Building
  - Provision macOS
  - Windows and macOS packages
  - Support profiles in colcon
- Documentation
  - Deprecate https://design.ros2.org. Content should move to either an REP, to https://github.com/ros2/ros2_documentation, or be removed.
  - Fix per-package documentation builder to be able to document build artifacts, i.e. messages, services, actions, etc.
  - Add documentation for implementing a new RMW.
— Provide three different kinds of content:
  ∗ “demos” to show features and cover them with tests
  ∗ “examples” to show a simple/minimalistic usage which might have multiple ways to do something
  ∗ “tutorials” which contain more comments and anchors for the wiki (teaching one recommended way)

New features

The trailing stars indicate the rough effort: 1 star for small, 2 stars for medium, 3 stars for large.

• Logging improvements [* / **]
  – Configuration specified in a file
  – Per-logger configuration (enabling e.g. rqt_logger_level)

• Time related
  – Support rate and sleep based on clock

• Additional Graph API features [** / ***]
  – Introspect QoS setting for all (especially remote) topics
  – a la ROS 1 Master API: https://wiki.ros.org/ROS/Master_API
  – Event-based notification
  – Requires knowledge of the rmw interface which needs to be extended

• Executo
  – Performance improvements (mostly around the waitset)
  – Deterministic ordering (fair scheduling)
  – Decouple waitables

• Message generation
  – Catch-up message generation for languages not supported out-of-the-box
  – Mangle field names in message to avoid language specific keywords
  – Improve generator performance by running them in the same Python interpreter

• Launch
  – Support for launching multi-node executables (i.e. manual composition)
  – Extend launch XML/YAML support: events and event handlers, tag namespaces and aliasing

• Rosbag
  – Support recording services (and actions)

• ros1_bridge
  – Support bridging actions

• RMW configuration
  – Unified standard way of configuring the middleware

• Remapping [** / ***]
  – Dynamic remapping and aliasing through a Service interface
• Type masquerading [***]
  – a la ROS 1’s message traits: https://wiki.ros.org/roscpp/Overview/MessagesSerializationAndAdaptingTypes
  – Requires knowledge of the typesupport system

• Expand on real-time safety [***]
  – For services, clients, and parameters
  – Expose more quality of service parameters related to real-time performance
  – Real-time-safe intra-process messaging

• Multi-robot supporting features and demos [***]
  – Undesired that all nodes across all robots share the same domain (and discover each other)
  – Design how to “partition” the system

• Support more DDS / RTPS implementations:
  – RTI Connext DDS Micro (implemented, not enabled by default or officially supported).

• Security improvements:
  – More granularity in security configuration (allow authentication only, authentication and encryption, etc.) [*]
  – Integrate DDS-Security logging plugin (unified way to aggregate security events and report them to the users through a ROS Interface) [**]
  – Key storage security (right now, keys are just stored in the filesystem) [**]
  – More user friendly interface (make it easier to specify security config). Maybe a Qt GUI? This GUI could also assist in distributing keys somehow. [***]
  – A way to say “please secure this running system” with some UI that would auto-generate keys and policies for everything that is currently running. [***]
  – If there are hardware-specific features for securing keys or accelerating encryption/signing messages, that could be interesting to add to DDS/RTPS implementations that don’t use it already. [***]

Reducing Technical Debt

• Fix flaky tests on https://ci.ros2.org/view/nightly.

• Ability to run (all) unit tests with tools e.g. valgrind, clang-tidy, clang static analysis (scan-build), ASAN, TSAN, UBSAN, etc.

• API review, specifically user-facing APIs in rclcpp and rclpy

• Refactor the rclcpp API into separate packages focused on a single aspect, rclcpp should afterward still provide the combined user-facing API

• Revisit message allocators, consider using std::polymorphic_allocator to address problems

• Synchronize / reconcile design docs with the implementation.

• Address / classify pending tickets

• Address TODOs in code / docs

• Remove tinyxml as a dependency
This page describes planned work for ROS 2. The set of planned features and development efforts should provide insight into the overall direction of ROS 2. If you would like to see other features on the roadmap, please get in touch with us at info@openrobotics.org.

Iron Roadmap

Iron Irwini is the ROS 2 release expected in May 2023. See release for a detailed timeline.

The items in the roadmap below are the major features being worked on by the ROS 2 community. The “Size” is an estimated size of the task, where Small means person-days to complete, Medium means person-weeks to complete, and Large means person-months to complete.

If you are working on a feature for ROS 2 Iron and would like to have it listed, please open a pull request to ROS 2 Documentation. If you’d like to take on one of these tasks, please get in touch with us.
<table>
<thead>
<tr>
<th>Task</th>
<th>Size</th>
<th>Owner</th>
<th>Expected Completion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Various improvements and port of further functionality for Diagnostics</td>
<td>Medium</td>
<td>Bosch</td>
<td>Q1 2023</td>
</tr>
<tr>
<td>License linter and copyright file generator for binary packages</td>
<td>Medium</td>
<td>Bosch</td>
<td>Q2 2023</td>
</tr>
<tr>
<td>rclcpp Dispatcher Executor for non-POSIX OS</td>
<td>Medium</td>
<td>Bosch</td>
<td>Q1 2023</td>
</tr>
<tr>
<td>Improve rclcpp executor performance</td>
<td>Large</td>
<td>Open Robotics</td>
<td>Q4 2022</td>
</tr>
<tr>
<td>DDS User Experience - Improve the out-of-the-box experience</td>
<td>Large</td>
<td>Open Robotics</td>
<td>Q1 2022</td>
</tr>
<tr>
<td>DDS User Experience - Configuration of Initial Peers for Discovery</td>
<td>Small</td>
<td>Open Robotics</td>
<td>Q4 2022</td>
</tr>
<tr>
<td>DDS User Experience - Develop a configuration tool</td>
<td>Medium</td>
<td>Open Robotics</td>
<td>Q4 2022</td>
</tr>
<tr>
<td>Python per-package documentation generation</td>
<td>Small</td>
<td>Open Robotics</td>
<td>Q4 2022</td>
</tr>
<tr>
<td>rclpy performance with large messages</td>
<td>Medium</td>
<td>Open Robotics</td>
<td>Q1 2023</td>
</tr>
<tr>
<td>ROS 1 to ROS 2 migration documentation and tools</td>
<td>Medium</td>
<td>Open Robotics</td>
<td>Q2 2023</td>
</tr>
<tr>
<td>SDF integration</td>
<td>Medium</td>
<td>Open Robotics</td>
<td>Q1 2023</td>
</tr>
<tr>
<td>Better error message for launch (stretch goal)</td>
<td>Medium</td>
<td>Open Robotics</td>
<td>Q2 2023</td>
</tr>
<tr>
<td>Relaunch of individual nodes in a complex system (stretch goal)</td>
<td>Small</td>
<td>Open Robotics</td>
<td>Q2 2023</td>
</tr>
<tr>
<td>Logging configuration and features (stretch goal)</td>
<td>Medium</td>
<td>Open Robotics</td>
<td>Q2 2023</td>
</tr>
<tr>
<td>Iron release</td>
<td>Large</td>
<td>Open Robotics</td>
<td>Q2 2023</td>
</tr>
<tr>
<td>ContentFiltering fallback in rcl</td>
<td>Large</td>
<td>Sony Group Corporation</td>
<td>Q1 2023</td>
</tr>
<tr>
<td>on_pub/sub_matched callback support</td>
<td>Medium</td>
<td>Sony Group Corporation</td>
<td>Q1 2023</td>
</tr>
<tr>
<td>ROS 2 core ContentFiltering Enhancement</td>
<td>Medium</td>
<td>Sony Group Corporation</td>
<td>Q2 2023</td>
</tr>
</tbody>
</table>

Additional project-specific roadmaps can be found in the links below:

- **MoveIt2**: https://moveit.ros.org/documentation/contributing/roadmap/
- **Nav2**: https://navigation.ros.org/roadmap/roadmap.html

**Planned releases**

Please see the *Distributions page* for the timeline of and information about future distributions.

**Contributing to ROS 2**

Looking for something to work on, or just want to help out? Here are a few resources to get you going.

1. The *Contributing* guide describes how to make a contribution to ROS 2.
2. Check out the list of *Feature Ideas* for inspiration.
3. For more information on the design of ROS 2 please see design.ros2.org.
4. The core code for ROS 2 is in the ros2 GitHub organization.
5. The Discourse forum/mailing list for discussing ROS 2 design is ng-ros.
6. Questions should be asked on ROS answers, make sure to include at least the ros2 tag and the rosdistro version you are running, e.g. iron.

**ROSCon Talks**

The following ROSCon talks have been given on ROS 2 and provide information about the workings of ROS 2 and various demos:

<table>
<thead>
<tr>
<th>Title</th>
<th>Links</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panel: The ROS 2 Developer Experience</td>
<td>video</td>
</tr>
<tr>
<td>Wearable ROS: Development of wearable robot system using ROS 2</td>
<td>video</td>
</tr>
<tr>
<td>Building ROS 2 enabled Android apps with C++</td>
<td>video</td>
</tr>
<tr>
<td>Distributed Robotics Simulator with Unreal Engine</td>
<td>video</td>
</tr>
<tr>
<td>Tools and processes for improving the certifiability of ROS 2</td>
<td>video</td>
</tr>
<tr>
<td>Failover ROS Framework : Consensus-based node redundancy</td>
<td>video</td>
</tr>
<tr>
<td>ROS 2 and Gazebo Integration Best Practices</td>
<td>video</td>
</tr>
<tr>
<td>Chain-Aware ROS Evaluation Tool (CARET)</td>
<td>video</td>
</tr>
<tr>
<td>ROS 2 network monitoring</td>
<td>video</td>
</tr>
<tr>
<td>How custom tasks are defined, assigned, and executed in Open-RMF</td>
<td>video</td>
</tr>
<tr>
<td>A practitioner's guide to ros2_control</td>
<td>video</td>
</tr>
<tr>
<td>Zenoh: How to Make ROS 2 Work at any Scale and Integrate with Anything</td>
<td>video</td>
</tr>
<tr>
<td>A case study in optics manufacturing with MoveIt2 and ros2_control</td>
<td>video</td>
</tr>
<tr>
<td>20/20 Robot Vision - How to setup cameras in ROS 1 &amp; ROS 2 using camera_aravis</td>
<td>video</td>
</tr>
<tr>
<td>Filter your ROS 2 content</td>
<td>video</td>
</tr>
<tr>
<td>Evolving Message Types, and Other Interfaces, Over Time</td>
<td>video</td>
</tr>
<tr>
<td>Migrating from ROS1 to ROS 2 - choosing the right bridge</td>
<td>video</td>
</tr>
<tr>
<td>On Use of Nav2 Smac Planners</td>
<td>video</td>
</tr>
<tr>
<td>Bazel and ROS 2 – building large scale safety applications</td>
<td>video</td>
</tr>
<tr>
<td>Native Rust components for ROS 2</td>
<td>video</td>
</tr>
<tr>
<td>The ROS build farm and you: How ROS packages you release become binary packages.</td>
<td>video</td>
</tr>
<tr>
<td>mROS 2: yet another runtime environment onto embedded devices</td>
<td>video</td>
</tr>
<tr>
<td>ROS 2 &amp; Edge Impulse: Embedded AI in robotics applications</td>
<td>video</td>
</tr>
<tr>
<td>micro-Ros goes Automotive: supporting AUTOSAR-based microcontrollers</td>
<td>video</td>
</tr>
<tr>
<td>An open architecture for Hardware Acceleration in ROS 2</td>
<td>video</td>
</tr>
<tr>
<td>ROS 2 and the Crazyflie: Aerial swarms and Autonomy with a tiny flying robot</td>
<td>video</td>
</tr>
<tr>
<td>A Raspberry Pi image with ROS 2 + RT and a customizable image builder</td>
<td>video</td>
</tr>
</tbody>
</table>

2021
<table>
<thead>
<tr>
<th>Title</th>
<th>Links</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMACC2</td>
<td>video</td>
</tr>
<tr>
<td>rospyn2: Convert a ROS1 node to ROS2 by changing only one line of code</td>
<td>video</td>
</tr>
<tr>
<td>Hello Robot: Democratizing Mobile Manipulation with ROS</td>
<td>video</td>
</tr>
<tr>
<td>Challenges for ROS2 in Autonomous Agricultural Applications</td>
<td>video</td>
</tr>
<tr>
<td>ESP32 microcontroller robot with Navigation 2 ROS 2 running in the Cloud</td>
<td>video</td>
</tr>
<tr>
<td>Middleware interconnecting ROS/ROS2 with the EtherCAT protocol</td>
<td>video</td>
</tr>
<tr>
<td>Build and Manage Cloud-enhanced ROS Robots with AWS IoT Greengrass 2.0</td>
<td>video</td>
</tr>
<tr>
<td>Driving Autonomy in Mobile Robotics with ROS2 and PX4</td>
<td>video</td>
</tr>
<tr>
<td>Integrating ROS 2 with existing DDS networks</td>
<td>video</td>
</tr>
<tr>
<td>VIPER: Volatiles Investigating Polar Exploration Rover</td>
<td>video</td>
</tr>
<tr>
<td>Chronicles of Caching and Containerising CI for Nav2</td>
<td>video</td>
</tr>
<tr>
<td>A New Way to Interact with PointCloud2 Messages</td>
<td>video</td>
</tr>
<tr>
<td>ROS 2 Grasp Library – Acceleration for 3D Object Pose Detection</td>
<td>video</td>
</tr>
<tr>
<td>ros2_control: The future of ros_control</td>
<td>video</td>
</tr>
<tr>
<td>ROS 2 Rolling Ridley</td>
<td>video</td>
</tr>
<tr>
<td>Hybrid Planning - Enabling Reactive Manipulation with MoveIt 2</td>
<td>video</td>
</tr>
<tr>
<td>Apex.OS Cert: Taking ROS 2 from prototype into production</td>
<td>video</td>
</tr>
<tr>
<td>Making a robot ROS 2 powered - a case study using the UR manipulators</td>
<td>video</td>
</tr>
<tr>
<td>Online Trajectory Generation and Admittance Control in ROS2</td>
<td>video</td>
</tr>
<tr>
<td>VSCode, Docker and ROS2</td>
<td>video</td>
</tr>
<tr>
<td>ROS 2 Content Filtered Topics</td>
<td>video</td>
</tr>
<tr>
<td>Rosbag2 for Power Users</td>
<td>video</td>
</tr>
<tr>
<td>We Have Ignition: The Next Generation of Gazebo Simulation</td>
<td>video</td>
</tr>
<tr>
<td>Using NVIDIA Isaac Sim For Synthetic Data Generation &amp; ROS Development</td>
<td>video</td>
</tr>
<tr>
<td>Augmented Reality and Robotics</td>
<td>video</td>
</tr>
<tr>
<td>Modeling sensors for simulation of ROS applications in Unity</td>
<td>video</td>
</tr>
<tr>
<td>Panel: Simulation Tools for ROS</td>
<td>video</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Title</th>
<th>Links</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accelerating Innovation with ROS: Lessons in Healthcare</td>
<td>video</td>
</tr>
<tr>
<td>Panel: Software Quality in Robotics</td>
<td>video</td>
</tr>
<tr>
<td>Panel: ROS Agriculture</td>
<td>video</td>
</tr>
<tr>
<td>Achieving Generality and Robustness in Semantic Segmentation</td>
<td>video</td>
</tr>
<tr>
<td>Navigation2: The Next Generation Navigation System</td>
<td>video</td>
</tr>
<tr>
<td>CHAMP Quadruped Control</td>
<td>video</td>
</tr>
<tr>
<td>Kiwibot: ROS2 in the atoms delivery industry</td>
<td>video</td>
</tr>
<tr>
<td>MoveItWorld</td>
<td>video</td>
</tr>
<tr>
<td>OpenCV</td>
<td>video</td>
</tr>
<tr>
<td>ROBOTIS TurtleBot3</td>
<td>video</td>
</tr>
<tr>
<td>Autoware</td>
<td>video</td>
</tr>
<tr>
<td>Dronecode</td>
<td>video</td>
</tr>
<tr>
<td>FIWARE</td>
<td>video</td>
</tr>
<tr>
<td>Lightning Talks and Sponsor Videos 1</td>
<td>video</td>
</tr>
<tr>
<td>Lightning Talks and Sponsor Videos 2</td>
<td>video</td>
</tr>
<tr>
<td>Lightning Talks and Sponsor Videos 3</td>
<td>video</td>
</tr>
<tr>
<td>Lightning Talks and Sponsor Videos 4</td>
<td>video</td>
</tr>
</tbody>
</table>

2019
<table>
<thead>
<tr>
<th>Title</th>
<th>Links</th>
</tr>
</thead>
<tbody>
<tr>
<td>Migrating a large ROS 1 codebase to ROS 2</td>
<td>slides / video</td>
</tr>
<tr>
<td>The New Architecture of Gazebo Wrappers for ROS 2</td>
<td>slides / video</td>
</tr>
<tr>
<td>Migrating to ROS 2: Advice from Rover Robotics</td>
<td>slides / video</td>
</tr>
<tr>
<td>ROS 2 on VxWorks</td>
<td>slides / video</td>
</tr>
<tr>
<td>Navigation2 Overview</td>
<td>slides / video</td>
</tr>
<tr>
<td>Launch Testing - Launch description and integration testing for ros2</td>
<td>slides / video</td>
</tr>
<tr>
<td>ROS 2 for Consumer Robotics: : The iRobot use-case</td>
<td>slides / video</td>
</tr>
<tr>
<td>Composable Nodes in ROS2</td>
<td>slides / video</td>
</tr>
<tr>
<td>Concurrency in ROS 1 and ROS 2</td>
<td>slides / video</td>
</tr>
<tr>
<td>A True Zero-Copy RMW Implementation for ROS2</td>
<td>slides / video</td>
</tr>
<tr>
<td>ROS2 Real-Time Behavior: Static Memory Allocation</td>
<td>video</td>
</tr>
<tr>
<td>PackML2: State Machine Based System Programming, Monitoring and Control in ROS2</td>
<td>slides / video</td>
</tr>
<tr>
<td>Quality of Service Policies for ROS2 Communications</td>
<td>slides / video</td>
</tr>
<tr>
<td>Micro-ROS: ROS2 on Microcontrollers</td>
<td>slides / video</td>
</tr>
<tr>
<td>ROS2 on Large Scale Systems: Discovery Server</td>
<td>video</td>
</tr>
<tr>
<td>Bridging Your Transitions from ROS 1 to ROS 2</td>
<td>slides / video</td>
</tr>
<tr>
<td>Markup Extensions for ROS 2 Launch</td>
<td>slides / video</td>
</tr>
</tbody>
</table>

2018

<table>
<thead>
<tr>
<th>Title</th>
<th>Links</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hands-on ROS 2: A Walkthrough</td>
<td>slides / video</td>
</tr>
<tr>
<td>Launch for ROS 2</td>
<td>slides / video</td>
</tr>
</tbody>
</table>

2017

<table>
<thead>
<tr>
<th>Title</th>
<th>Links</th>
</tr>
</thead>
<tbody>
<tr>
<td>The ROS 2 vision for advancing the future of robotics development</td>
<td>slides / video</td>
</tr>
</tbody>
</table>

2016

<table>
<thead>
<tr>
<th>Title</th>
<th>Links</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROS 2 Update - summary of alpha releases, architectural overview</td>
<td>slides / video</td>
</tr>
<tr>
<td>Evaluating the resilience of ROS2 communication layer</td>
<td>slides / video</td>
</tr>
</tbody>
</table>

2015

<table>
<thead>
<tr>
<th>Title</th>
<th>Links</th>
</tr>
</thead>
<tbody>
<tr>
<td>State of ROS 2 - demos and the technology behind</td>
<td>slides / video</td>
</tr>
<tr>
<td>ROS 2 on “small” embedded systems</td>
<td>slides / video</td>
</tr>
<tr>
<td>Real-time control in ROS and ROS 2</td>
<td>slides / video</td>
</tr>
</tbody>
</table>

2014

<table>
<thead>
<tr>
<th>Title</th>
<th>Links</th>
</tr>
</thead>
<tbody>
<tr>
<td>Why you want to use ROS 2</td>
<td>slides / video</td>
</tr>
<tr>
<td>Next-generation ROS: Building on DDS</td>
<td>slides / video</td>
</tr>
</tbody>
</table>
Project Governance

Technical Steering Committee (TSC)

Since the beginning of ROS, the project has been overseen and prioritized primarily by one organization, first Willow Garage and now Open Robotics. That approach has worked well enough, as evidenced by the widespread adoption of ROS around the world.

But with ROS 2, we want to broaden participation to accelerate ROS 2 delivery, starting with these areas: determining the roadmap, developing core tools and libraries, and establishing working groups to focus on important topics. To that end, we’ve established a Technical Steering Committee (TSC). As described in the charter, the TSC comprises representatives of organizations that are contributing to the development of ROS 2, and it has the responsibility to set the technical direction for the project.

Packages relevant to this ROS 2 TSC are listed in REP 2005.

Meeting notes can be found on ROS Discourse.

The current members of the ROS 2 TSC are (23 as of 2022-02-01):

If you are interested in joining the ROS 2 TSC, please inquire via info@openrobotics.org.

ROS 2 Technical Steering Committee Charter

Date 2021-02-06
Version 0.5
Organization info@openrobotics.org

Table of Contents

- Guiding Principles & Rationale
- Responsibilities of the TSC
- Establishment of the TSC
- Community representation
- Meetings
- Working Groups
- Voting
- Project Roles
Guiding Principles & Rationale

The ROS 2 Project (“ROS 2” or “the project”) operates openly, such that plans, decisions, and processes are freely available to the general public in a timely manner. All interested parties should have access to information about the project direction and timeline, from casual observers to companies planning product releases.

Responsibilities of the TSC

The ROS 2 Technical Steering Committee (“TSC”) is responsible for the technical direction that the project takes. The TSC makes decisions on the project roadmap, developer policies and process, release schedule, and other matters that require technical supervision in ROS 2. It is intended that these decisions be final and authoritative, but successful implementation of any decision depends on commitment of the necessary resources from the organizations represented on the TSC.

The TSC aims to ensure the following characteristics of the project:

1. **Independent and transparent governance**
   The project is driven by the community that contributes to and uses the resulting software, with Open Robotics as the founding project sponsor.

2. **Coordination**
   Those individuals and organizations that are contributing the labor and other resources to develop the project are in regular contact with each other to coordinate their plans and activities.

3. **Pace & quality**
   The project advances as quickly as possible given the available resources and without sacrificing quality in the resulting software.

4. **Policies**
   Contributions to the project are received, reviewed, and tested in a manner that is consistent with the requirements for widespread adoption of the resulting software in industry.

Establishment of the TSC

The TSC is made up of individuals who contribute materially to the project and/or individuals who represent organizations that contribute materially to the project and/or selected community representatives. The TSC seeks to coordinate the contributions that members and their organizations are voluntarily making to the project. The TSC cannot compel or oblige any action on the part of a member, his or her organization, or that organization’s employees.

1. Material contribution is the commitment of a minimum level of effort toward developing software and/or documentation that is contributed to the project under the appropriate open source license. This contribution may be in the form of in-kind labor and/or outsourced work. No fees or payments beyond material contribution to the project are required for TSC membership.

2. The minimum level of effort commitment required to qualify for TSC membership is 1 full-time equivalent (FTE). This level may be changed by the TSC via a standard TSC motion and vote.

3. The TSC must have at least three members.

4. TSC membership is not time-limited.
5. There is no maximum size for the TSC. A size limit may be established by the TSC via a standard TSC motion and vote.

6. The TSC may add additional members to the TSC by a standard TSC motion and vote. When considering the addition of a new member, the TSC is responsible for determining whether that potential member’s material contributions to the project are appropriate and sufficient to qualify for membership.

7. A TSC member may be removed from the TSC by voluntary resignation, by a standard TSC motion and vote, or in accordance with the following participation rule:
   - In the case where an individual TSC member – within any three-month period – attends fewer than 1/3 of the regularly scheduled meetings, does not participate in TSC discussions, and does not participate in TSC votes, the member shall be automatically removed from the TSC.

8. The TSC will periodically review the material contributions of TSC members to ensure that they continue to meet the minimum level of effort established by the TSC for TSC membership. In the case where a TSC member’s material contributions fall below that minimum level, the TSC may decide to remove that TSC member by a standard TSC motion and vote.

9. With the exception of Open Robotics, there may be at most one TSC member from any single employer.

10. A TSC member may designate a proxy to participate in the TSC on that member’s behalf.

11. TSC membership is public; the membership list is published via a website that Open Robotics maintains and updates periodically.

12. Open Robotics, as the founding sponsor of the project, has permanent membership on the TSC, hosts TSC meetings, and provides a representative to serve as chairperson.

13. Any organization or individual may apply for membership on the TSC by email to info@openrobotics.org. The ROS 2 TSC applicant intake process is here.

Community representation

The TSC shall include a number of individuals who represent the broader ROS community rather than any particular organization or individual.

1. The number of community representatives shall be set by a decision of the TSC. However, this number shall not be less than three, and shall also not be less than 10% of the number of non-community representative members in the TSC.

2. A community representative serves for two years.
   - The term of service begins on the 1st of December immediately following an election.

3. Fifty percent (50%) of the initial set of community representatives shall serve for only one year in their first term to stagger the community representative terms.

4. A community representative may be removed from the TSC earlier than two years if one of the following conditions is met:
   - The community representative requests to stand down voluntarily.
   - The TSC chairperson deems that the community representative has behaved inappropriately. The TSC chairperson must make public the reason for standing down the community representative.

5. When a community representative is removed from the TSC:
   - If the number of community representatives is below the specified minimum, the TSC shall recommend and the chairperson shall appoint a new community representative to serve the remainder of the removed community representative’s term.
• If the number of community representatives is above the specified minimum, no special action shall be taken.

6. A community representative is selected via an open election.

7. An individual is eligible to stand for election as a community representative if:
   • The maximum number of community representatives from any single organisation is one.
   • The individual has voluntarily nominated themselves.
   • That individual’s nomination has not been vetoed by the TSC chairperson.

8. Community representative elections shall be held in the following manner:
   • On the first Tuesday of September a ROS Discourse post will be made asking for TSC member nominations.
   • The nomination period will last two weeks until the third Tuesday in September.
   • Prospective TSC member nominees must self nominate, but the community is encouraged to use the nomination Discourse thread to encourage other community members to run.
   • To self-nominate prospective TSC members must submit a candidate packet that includes a written biography, headshot, a written statement, and optionally a short video introduction.
   • The written statement shall state why the candidate believes themselves suitable and what they intend to accomplish as a community representative.
   • Between the third Tuesday in September and the first Tuesday in October the ROS 2 TSC will review the candidates for any objections or conflicts of interest. The TSC chair will prepare a ROS Discourse post that includes all of the candidate packets.
   • On the first Tuesday of October the candidate packets will be posted to ROS Discourse along with with the online balloting form.
   • The period between the start of the election process and the close of voting may be used for campaigning by candidates.
   • The eligibility criteria for voters shall be determined by the TSC prior to commencing an election and clearly stated in the election information from the start of nominations until the close of voting.
   • Balloting will be open from the first Tuesday in October until the end of the first full day of ROSCon.
   • Balloting shall be conducted using the Condorcet method.
   • Candidates are encouraged, but not required, to attend ROSCon.
   • The results of the election shall be announced during the closing session of ROSCon along with an announcement post on ROS Discourse.
   • In the event that ROSCon is not held during a given calendar year for any reason, then the foregoing election process shall still be followed, with procedural modifications as necessary, at a time of the TSC’s choosing, but with the results being announced no later than the 15th of December.
Meetings

The TSC meets regularly in-person and/or via an electronic means.

1. A minimum of three TSC members is required to establish a quorum to hold a TSC meeting.

2. Participation in a TSC meeting is open to current members. Other organizations or individuals may be invited by Open Robotics to observe a TSC meeting. TSC members may suggest to Open Robotics the names of individuals or organizations for consideration as potential observers.

3. Minutes from each TSC meeting are published by Open Robotics shortly after the meeting. TSC meetings are not broadcast or recorded verbatim, but they should be treated as public discussions.

Working Groups

The TSC may establish one or more working groups (“WGs”) to discuss specific topics in greater detail than is practical for the TSC as a whole.

1. A WG may be established by the TSC via standard TSC motion and vote. At the time of such establishment, the TSC is responsible for deciding the name, scope, and initial chair of the new WG.

2. A WG may be dissolved by the TSC via standard TSC motion and vote.

3. Each WG must be chaired by a TSC member. The chair of a WG is responsible for all aspects of that WG’s operation, including membership, meeting schedule, and decision-making.

4. A WG may bring findings and/or recommendations to the TSC for discussion and/or decision.

Voting

The TSC aims to operate by consensus. When consensus cannot be reached, decisions should in most cases be made by resource commitment: e.g., if a member wants to include a feature on the roadmap but the rest of the TSC is uninterested, then that member can choose to commit his or her organization’s contribution to development of the desired feature and thereby have it included in the roadmap. In cases where consensus cannot be reached and resource commitment is insufficient or inappropriate, simple majority voting is used, with each member having one vote.

Project Roles

TSC membership does not influence the handling of code contributions, which continues to follow the established federated development model in which contributions are made by pull request and reviewed and approved by the appropriate maintainer(s) prior to merging. Commit rights and maintainer status are earned through code contributions to the relevant package(s).

Confidentiality

The TSC is set up to transparently guide the community. However, to facilitate effective communication, the ability to have some confidential discussions is important. TSC members should share confidential information within the TSC and other members are expected to respect that confidentiality. The following are rules regarding confidentiality in TSC communications, events, and meetings. All other communications may be made freely available to the broader community for transparency. Breaking the rules in this section is sufficient, but not necessary, causality for expulsion of the representative or member company, depending on the nature of the infraction, from the TSC by a standard motion and vote.
1. Statements made by individuals in discussions surrounding a vote or topic of contention which are not reflected in the publicly posted minutes should be considered confidential.

2. Statements made by individuals regarding companies, research groups, individuals, or other entities for which they are not also members should be considered confidential. This is exempt when the entity in question is a member and in attendance to the TSC meeting or event.

3. Statements made in obvious confidence or explicitly stated as confidential regarding their own entity or opinions should remain confidential.

4. Rule a-c may be waived by the TSC member who made the statement only by explicit verbal or written approval. If the existence of a waiver is put into question by either party, this waiver is considered nullified.

5. Affiliates of TSC member organizations or guests of TSC events privy to confidential information will be held to the same standards set out in this section. It is the responsibility of the TSC member issuing invitations or sharing information to inform relevant parties of these confidentiality requirements. If this policy is broken, the TSC member is in full responsibility for the disclosure.

**ROS 2 TSC applicant intake process**

**Objectives**

1. Increase the pool of high quality applicants
2. Help applicants understand what the ROS 2 TSC expects from an application
3. Help ROS 2 TSC better evaluate the applicants

**Process**

1. Applicant inquires about joining TSC via info@openrobotics.org per ROS 2 Governance.
2. Open Robotics responds with the application questions (below).
3. Applicant completes and sends application to info@openrobotics.org.
4. Open Robotics does an initial review and gives applicant feedback if warranted. For example:
   - “consider developing your track record of contribution and then re-apply.”
   - “What you contribute to is not part of ROS 2 Common Packages (REP 2005). Consider getting it added before proceeding with application for more favorable reception.”
5. If application seems worthwhile, Open Robotics passes it to TSC member to “sponsor”.
   - Open Robotics can ask TSC for volunteer(s) per applicant or Open Robotics could have list of TSC members who already volunteered to support this function.
6. Sponsor arranges short 1:1 with applicant to talk about ROS 2 TSC role and responsibilities. Afterwards the applicant can:
   a) proceed with application as-is
   b) revise application
   c) delay submission e.g. to develop their contribution track record
   d) decide not to proceed.
7. [two weeks before TSC meeting]: applications are distributed to TSC members for independent review
• If 2 TSC members respond agreeing to “sponsor” applicant then these proceed to the next step. Applicant should “minimally meet” TSC member criteria before addressing TSC.
• Otherwise Open Robotics informs applicant “Not at this time, please consider applying again later with more contribution history.”

8. [in TSC meeting] Applicant or Sponsor can address TSC. 5 minute presentation with up to 5 additional minutes of Q&A.

9. [in TSC meeting]: Synchronous private/unshareable discussion of the applications after applicant(s) leave the call. TSC members are to be discrete about what has been discussed.

10. [after TSC meeting]: Asynchronous vote on the applicants completed in around 10 days

11. Open Robotics notifies applicant(s) of TSC decision.

Questionnaire for the applicant

The following is to be provided to the applicant by Open Robotics via email in response to them expressing interest in joining the ROS 2 TSC.

1. To which aspect(s) of ROS 2 do you plan to make open source contributions?
   • For example contributions to ROS 2 Common Packages.
   • Are you already making those contributions? Since when? Please explain your history of contribution. If you have none, consider applying after developing some history of contribution.
   • Are there any other kinds of contributions that you are already making or plan? Education, evangelism, et al.?

2. What level of effort do you plan to dedicate to making those contributions?
   • How many FTE (full time equivalent) people will be making those contributions?
   • How many FTE are already doing so?

3. Are these contributions already happening now, and if so, can you point to results produced so far?
   • For example describe or attach details of your contributions to ROS 2 Common Packages.
   • Links to your contributions are helpful.

4. How do these contributions benefit your business or organization?
   • Why do you do this?

Evaluation Guide for TSC members

The following is suggested guidance for TSC members to evaluate applicants:

1. To which aspect(s) of ROS 2 do you plan to make open source contributions?
   • Are the contributions useful to the community?
   • Are the contributions to ROS 2 Common Packages?

2. What level of effort do you plan to dedicate to making those contributions?
   • Give more credence to past contributions
   • Proposed future contributions are important but consider past contribution to evaluate how credible those plans are.
3. Are these contributions already happening now, and if so, can you point to results produced so far?
   • Are the contributions ones that the community and TSC values?
   • Are the contributions to ROS 2 Common Packages (REP 2005)?
   • Emphasis is on what they’ve contributed already, not what they promise to do. If they don’t have enough evidence of contribution, encourage them to apply after developing their contribution track record.

4. How do these contributions benefit your business or organization?
   • This answer helps you evaluate how much focus and investment it already has and whether it seems likely to continue.
   • Do you see that the applicant has a vested interest in this work and the success of ROS 2? Or does it seem they seek to join merely for prestige and marketing?

How to Start a Community Working Group

A ROS working group is a great way to get a bunch of like minded people together to make regular progress towards a shared goal. So what are the steps to making a working group?

1. First try to find a couple of like-minded individuals who are willing to regularly commit their time to a particular area of ROS development. We suggest having at minimum three people who can help orchestrate the working group.

2. Draft up a one page summary of your working group. This summary should broadly cover what topics you would like to cover and what goals you would like to accomplish. Take a look at the other working groups and make sure that you don’t overlap too much.

3. Once you have your ideas roughly distilled take it to the ROS community to find more like minded members. We generally recommend making a post on ROS discourse on the general channel with the title “Proposal for ROS X Working Group”. Include your summary and your current collaborators. You should use this thread to:
   a) Finalize the subjects covered by your working group.
   b) Recruit members for your working group.
   c) Set the agenda for your first couple of working group meetings.
   d) Pick a regular time for your working group meetings (we find that early in the morning PST, generally works for most of the world). At this point you should also make a discourse tag (you use these when you make your post. We suggest something like: “wg-<topic>”.

4. Once you have recruited your working group members you’ll have some leg work to do. At this point you should have a list of working group members. Create a google group by going to Google groups (https://groups.google.com/my-groups) and adding your members. This mailing list should be named <dashed-name>-working-group-invites@googlegroups.com. It should be setup such that:
   • Anyone can join
   • No one can post (aka only owners)
   • Anyone can view members

   The mailing list will be used to distribute the invitations only, any communications should use ROS discourse with the working group’s tag. Please add tfoote@osrfoundation.org as an owner of the mailing list to help with administration.

5. Now that you have an e-mail group you can associate that group with the ROS events calendar. The calendar is where you will schedule your working group meetings. To join the calendar, send an e-mail request to Kat (kat@openrobotics.org) or Tully (tfoote@openrobotics.org).
6. To formalize your group you need to create a working group charter and a git repository to keep your documents and code. We have a working group template that can be found here. Anyone in the group can host the template or you can request that your group be added to the Open Robotics / OSRF organization. The decision is up to the working group.

7. Now that you have a working group charter, e-mail group, and source repository, you can add all of that information to the ROS 2 project governance website by sending a pull request to https://github.com/ros2/ros2_documentation.

8. Now it is time to schedule your first official working group meeting! We recommend announcing the meeting on ROS Discourse approximately one week before the meeting date. Please include the following in your announcement:
   a) An agenda for the meeting.
   b) The date and time for the meeting (protip: discourse has a date/time feature that can auto adjust for each user’s timezone).
   c) URLs / Links / Contact Info for the meeting. Open Robotics uses Google Hangouts, but you can use whatever technology you would like.

When you schedule your meeting, create an event on the ROS Calendar and invite the Google group so that anyone interested can sign up for automatic invitations.

9. Have your meetings! Regularly!
   a) Since we are a global community it can help to record your meetings. If you have an invited lecture we highly recommend recording it for the benefit of the whole community. Make sure the group / lecturer consents to being recorded, and it is wise to do a test run before the event.
      i) On Linux SimpleScreenRecorder works well but you may need to iterate to find the best combination of video conferencing platform, screen recorder, and settings.
      ii) Once your meeting is over, find somewhere online to save the video. Feel free to use your own Youtube channel to post the video; alternatively Open Robotics can host the video on their Vimeo account by sending a download link to kat@openrobotics.org.
   b) Make sure to take notes! You should nominate a scribe or note taker who summarizes the meeting and posts the results of the meeting to discourse.

Working Groups (WG)

As described in its charter, the TSC establishes working groups (WGs) to discuss and make progress on specific topics. The current WGs are (12 as of 2021-01-12):

Client Libraries

- Lead(s): Geoffrey Biggs, Alberto Soragna
- Note: This working group is currently on hiatus. Meetings will resume at some point in the future TBD.
- Resources:
  - Meeting invite group: ros-client-libraries-working-group-invites@googlegroups.com
  - Meeting minutes and agendas
  - Working group charter: https://github.com/ros2-client-libraries-wg/community
  - Discourse tag: wg-client-libraries
Control

• Lead(s): Bence Magyar
• Resources:
  • Website link: https://control.ros.org
  • Meeting invite group ros-control-working-group-invites@googlegroups.com
  • Discourse tag: wg-ros2-control

Embedded Systems

• Lead(s): Lara Moreno, Pablo Garrido
• Resources:
  • 2019-07-29 meeting notes
  • 2019-01-15 meeting notes
  • Meeting invite group ros-embedded-working-group-invites@googlegroups.com
  • Discourse tag: wg-embedded

Middleware

• Lead(s): William Woodall
• Resources:
  • Meeting invite group ros-middleware-working-group-invites@googlegroups.com
  • Discourse tag: wg-middleware

Navigation

• Lead(s): Steve Macenski
• Resources:
  • 2019-03-17 meeting notes
  • Meeting invite group ros-navigation-working-group-invites@googlegroups.com
  • Discourse tag: wg-navigation
  • Discourse Channel: Navigation Stack
  • Slack Group: Nav2 Slack
Manipulation

- Lead(s): Henning Kayser
- Resources:
  - About our working group meetings
  - Meeting invite group ros-manipulation-working-group-invites@googlegroups.com
  - Discourse tag: wg-moveit
  - Discourse Channel: MoveIt

Real-time

- Lead(s): Andrei Kholodnyi, Carlos San Vicente
- Resources:
  - Working group website
  - Working Group Community
  - Meeting invite group ros-real-time-working-group-invites@googlegroups.com
  - Discourse tag: wg-real-time
  - Matrix chat +ros-realtime:matrix.org

Security

- Lead(s): Florencia Cabral
- Resources:
  - ROS 2 Security Working Group Community
  - Meeting invite group ros-security-working-group-invites@googlegroups.com
  - Discourse tag: wg-security
  - Matrix chat +rosorg-security:matrix.org

Rosbag2 and Tooling

- Lead(s): Michael Orlov
- Resources:
  - Charter
  - Meeting Notes
  - Meeting invite group ros-tooling-working-group-invites@googlegroups.com
  - Discourse tag: wg-tooling
  - Matrix chat +ros-tooling-matrix.org

If you’d like to join an existing ROS 2 WG, please contact the appropriate group lead(s) directly. If you’d like to create a new WG, please inquire via info@openrobotics.org.
Working Group Policies

- Meetings should be posted to the Google calendar as well as announced on Discourse.
- Meetings should have notes and be posted to Discourse using appropriate working group tag.
- For attending the groups meetings please join the associated google group to get invites automatically.

Upcoming ROS Events

Upcoming Working group meetings can be found in this Google Calendar. It can be accessed via iCal.
If you have an individual event or series of events that you’d like to post please contact info@openrobotics.org

Marketing

General Use ROS Artwork

The ROS 2 media kit, which includes branding language, high resolution ROS logo graphics, and release images, can be found in the ROS art repository. Please refer to this repository for ROS art work and our branding guidelines.

Stickers, Posters, and Canvas Prints

Open Robotics hosts an online storefront on Zazzle.com with artwork available in various formats.
- Posters Printed on paper with selectable sizes.
- Canvas Prints Printed on wrapped canvas with selectable sizes.
- Sticker Sheets Printed on a sheet, 20 per sheet for the small, 6 per sheet in the large size.

Brochure: Why ROS 2?

Use this brochure to promote the goals and features of ROS 2 and encourage adoption.
License: CC BY-ND 4.0
Available formats:
- A4 (for web/email)
- A4 (for print)
- US Letter (for web/email)
- US Letter (for print)
4.8.8 API Documentation

All ROS 2 package’s documentation is hosted alongside its information on the ROS Index. Searching for packages will yield their information such as released distributions, README.md files, URLs, and other important metadata. A list of all packages for each distribution on the index can be found here. From the right-hand side of a package of interest, the auto-generated API documentation may be found using the “API Docs” button.

You can find the API Documentation for the Client Libraries in the Iron distribution using the links below:

- rclcpp - C++ client library
- rclcpp_lifecycle - C++ lifecycle library
- rclcpp_components - C++ components library
- rclcpp_action - C++ actions library

A raw list of Iron package documentation may be found here.

4.8.9 Related Projects

Gazebo

Gazebo (gazebosim.org) and its predecessor Gazebo Classic are the first open source choice for 3D physics simulation of ROS-based robots.

Large Community Projects

Large community projects involve multiple developers from all over the globe and are typically backed by a dedicated working group (cf. Project Governance).

- ros2_control (control.ros.org): Flexible framework for real-time control of robots implemented with ROS 2.
- MoveIt (moveit.ros.org): A rich platform for building manipulation applications featuring advanced kinematics, motion planning, control, collision checking, and much more.
- micro-ROS (micro.ros.org): A platform for putting ROS 2 onto microcontrollers, starting at less than 100 kB of RAM.

Further Community Projects

The global ROS community develops and maintains hundreds of further packages on top of the core ROS 2 stack. Some of them come with their own websites for documentation. Your best entry point to discover these works is the ROS Index (index.ros.org).

Hint for developers: If you maintain a README.md file in the root of your package folder (which is not necessarily the root of your repository), this file is rendered into the overview page of your package at index.ros.org. The file may be used for documenting your package and supersedes the package documentation pages in the ROS Wiki from ROS 1. See the fmi_adapter package as an example.
Company-driven Projects

Intel ROS 2 Projects

Intel® Robotics Open Source Project (Intel® ROS Project) to enable object detection/location/tracking, people detection, vehicle detection, industry robot arm grasp point analysis with kinds of Intel technologies and platforms, including CPU, GPU, Intel® Movidius™ NCS optimized deep learning backend, FPGA, Intel® RealSense™ camera, etc.

Key Projects

We are working on below ROS 2 projects and publish source code through https://github.com/intel/ or ROS 2 GitHub repo gradually.

- **ROS2 OpenVINO**: ROS 2 package for Intel® Visual Inference and Neural Network Optimization Toolkit to develop multiplatform computer vision solutions.
- **ROS2 RealSense Camera**: ROS 2 package for Intel® RealSense™ D400 serial cameras
- **ROS2 Movidius NCS**: ROS 2 package for object detection with Intel® Movidius™ Neural Computing Stick (NCS).
- **ROS2 Object Messages**: ROS 2 messages for object.
- **ROS2 Object Analytics**: ROS 2 package for object detection, tracking and 2D/3D localization.
- **ROS2 Message Filters**: ROS 2 package for message synchronization with time stamp.
- **ROS2 CV Bridge**: ROS 2 package to bridge with openCV.
- **ROS2 Object Map**: ROS 2 package to mark tag of objects on map when SLAM based on information provided by ROS 2 object analytics.
- **ROS2 Moving Object**: ROS 2 package to provide object motion information (like object velocity on x, y, z axis) based on information provided by ROS 2 object analytics.
- **ROS2 Grasp Library**: ROS 2 package for grasp position analysis, and compatible with MoveIt grasp interfaces.
- **ROS2 Navigation**: ROS 2 package for robot navigation, it’s already integrated to ROS 2 Crystal release.
- **Intel Robot DevKit (SDK)**: An open source project which enables developers to easily and efficiently create, customize, optimize, and deploy a robot software stack to an Autonomous Mobile Robot (AMR) platform based on the Robot Operating System 2 (ROS 2) framework.

Reference

ROS components at: https://wiki.ros.org/IntelROSProject shows the relationship among those packages, which also applies to ROS 2.
NVIDIA ROS 2 Projects

NVIDIA Jetson is working towards developing ROS 2 packages to ease the development of AI applications for robotics.

ROS Projects

- **Isaac ROS Nvblox**: Hardware-accelerated 3D scene reconstruction and Nav2 local costmap provider using nvblox.
- **Isaac ROS Object Detection**: Deep learning model support for object detection including DetectNet.
- **Isaac ROS DNN Inference**: This repository provides two NVIDIA GPU-accelerated ROS 2 nodes that perform deep learning inference using custom models. One node uses the TensorRT SDK, while the other uses the Triton SDK.
- **Isaac ROS Visual SLAM**: This repository provides a ROS 2 package that estimates stereo visual inertial odometry using the Isaac Elbrus GPU-accelerated library.
- **Isaac ROS Argus Camera**: This repository provides monocular and stereo nodes that enable ROS developers to use cameras connected to Jetson platforms over a CSI interface.
- **Isaac ROS image_pipeline**: This metapackage offers similar functionality as the standard, CPU-based image_pipeline metapackage, but does so by leveraging the Jetson platform’s specialized computer vision hardware.
- **Isaac ROS Common**: Isaac ROS common utilities for use in conjunction with the Isaac ROS suite of packages.
- **Isaac ROS AprilTags**: ROS 2 node uses the NVIDIA GPU-accelerated AprilTags library to detect AprilTags in images and publish their poses, ids, and additional metadata.
- **ROS and ROS 2 Docker Images**: Docker images for easy deployment on the NVIDIA Jetson platform, consisting of ROS 2, PyTorch, and other important machine learning libraries.
- **ROS and ROS 2 DockerFiles**: Dockerfiles for ROS 2 based on l4t which all you to build your own Docker image.
- **ROS 2 Packages for PyTorch and TensorRT**: ROS 2 packages for classification and object detection tasks using PyTorch and NVIDIA TensorRT. This tutorial is a good starting point AI integration with ROS 2 on NVIDIA Jetson.
- **ROS / ROS 2 Packages for Accelerated Deep Learning Nodes**: Deep learning image recognition, object detection, and semantic segmentation inference nodes and camera/video streaming nodes for ROS/ROS 2 using the jetson-inference library and NVIDIA Hello AI World tutorial.
- **ROS 2 Package for Human Pose Estimation**: A ROS 2 package for human pose estimation.
- **ROS 2 Package for Hand Pose Estimation and Gesture Classification**: A ROS 2 package for real-time hand pose estimation and gesture classification using TensorRT.
- **GPU accelerated ROS 2 Packages for Monocular Depth Estimation**: ROS 2 package for NVIDIA GPU-accelerated torch2trtxb examples such as monocular depth estimation and text detection.
- **ROS 2 Package for Jetson stats**: ROS 2 package for monitoring and controlling your NVIDIA Jetson [Xavier NX, Nano, AGX Xavier, TX1, TX2].
- **ROS 2 Packages for DeepStream SDK**: ROS 2 package for NVIDIA DeepStream SDK.
Simulation Projects

- **Isaac Sim Nav2**: In this ROS 2 sample, we are demonstrating Omniverse Isaac Sim integrated with the ROS 2 Nav2 project.

- **Isaac Sim Multiple Robot ROS 2 Navigation**: In this ROS 2 sample, we are demonstrating Omniverse Isaac Sim integrated with the ROS 2 Nav2 stack to perform simultaneous multiple robot navigation.

References

More updates on NVIDIA Jetson ROS 2 can be found here.

4.8.10 Glossary

Glossary of terms used throughout this documentation:

**API** An API, or Application Programming Interface, is an interface that is provided by an “application”, which in this case is usually a shared library or other language appropriate shared resource. APIs are made up of files that define a contract between the software using the interface and the software providing the interface. These files typically manifest as header files in C and C++ and as Python files in Python. In either case it is important that APIs are grouped and described in documentation and that they are declared as either public or private. Public interfaces are subject to change rules and changes to the public interfaces prompt a new version number of the software that provides them.

**client_library** A client library is an API that provides access to the ROS graph using primitive middleware concepts like Topics, Services, and Actions.

**package** A single unit of software, including source code, build system files, documentation, tests, and other associated resources.

**REP** ROS Enhancement Proposal. A document that describes an enhancement, standardization, or convention for the ROS community. The associated REP approval process allows the community to iterate on a proposal until some consensus has been made, at which point it can be ratified and implemented, which then becomes documentation. All REPs are viewable from the REP index.

**VCS** Version Control System, such as CVS, SVN, git, mercurial, etc...

**rclcpp** The C++ specific Client Library for ROS. This includes any middleware related APIs as well as the related message generation of C++ data structures based on interface definitions like Messages, Services, and Actions.

**repository** A collection of packages usually managed using a VCS like git or mercurial and usually hosted on a site like GitHub or BitBucket. In the context of this document, repositories usually contain one or more packages of one type or another.

4.8.11 Citations

If you use ROS 2 in your work please cite the 2022 Science Robotics paper *Robot Operating System 2: Design, architecture, and uses in the wild*.


```latex
@article{
  doi:10.1126/scirobotics.abm6074,
  author = {Steven Macenski and Tully Foote and Brian Gerkey and Chris Lalancette and William Woodall },
  (continues on next page)
```
If you use ROS2Composition in your work, please cite the 2023 IEEE RA-L paper Impact of ROS2 Node Composition in Robotic Systems.


The Robot Operating System (ROS) is a set of software libraries and tools for building robot applications. From drivers and state-of-the-art algorithms to powerful developer tools, ROS has the open source tools you need for your next robotics project.

Since ROS was started in 2007, a lot has changed in the robotics and ROS community. The goal of the ROS 2 project is to adapt to these changes, leveraging what is great about ROS 1 and improving what isn’t.

This site contains the documentation for ROS 2. If you are looking for ROS 1 documentation, check out the ROS wiki. If you use ROS 2 in your work, please see Citations to cite ROS 2.

4.8.12 Getting started

- **Installation**
  - Instructions to set up ROS 2 for the first time
- **Tutorials**
  - The best place to start for new users!
  - Hands-on sample projects that help you build a progression of necessary skills
- **How-to Guides**
  - Quick answers to your “How do I...?” questions without working through the Tutorials
- **Concepts**
  - High-level explanations of core ROS 2 concepts covered in the Tutorials
- **Contact**
  - Answers to your questions or a forum to start a discussion
4.8.13 The ROS 2 project

If you’re interested in the advancement of the ROS 2 project:

• **Contributing**
  – Best practices and methodology for contributing to ROS 2, as well as instructions for migrating existing ROS 1 content to ROS 2

• **Distributions**
  – Past, present and future ROS 2 distributions

• **Features Status**
  – Features in the current release

• **Feature Ideas**
  – Ideas for nice-to-have features that are not under active development

• **Roadmap**
  – Planned work for ROS 2 development

• **ROSCon Talks**
  – Presentations by the community on ROS 2

• **Project Governance**
  – Information about the ROS Technical Steering Committee, Working Groups, and upcoming events

• **Marketing**
  – Downloadable marketing materials

4.8.14 Other ROS resources

• **ROS Answers** (ROS 1, ROS 2)
  – Q&A community website, similar to Stack Exchange
  – See Contact Page for more information

• **ROS Enhancement Proposals (REPs)** (ROS 1, ROS 2)
  – Proposals for new designs and conventions

• **ROS Discourse** (ROS 1, ROS 2)
  – Forum for general discussions and announcements for the ROS community
  – See the Contact Page for more information

• **ROS Index** (ROS 1, ROS 2)
  – Indexed list of all packages (i.e. Python Package Index (PyPI) for ROS packages)
  – See which ROS distributions a package supports
  – Link to a package’s repository, API documentation, or website
  – Inspect a package’s license, build type, maintainers, status, and dependencies
  – Get more info for a package on ROS Answers

• **ROS Prerelease** (ROS 1)
- Generates commands to emulate the ROS Buildfarm on your local machine
- Currently only shows ROS 1 distributions

- **ROS Robots (ROS 1, ROS 2)**
  - Showcases robots projects from the community
  - Instructions on how to contribute a robot

- **ROS Wiki (ROS 1)**
  - ROS 1 documentation and user modifiable content
  - Active until at least the last ROS 1 distribution is EOL

- **ROS.org (ROS 1, ROS 2)**
  - ROS 1 and ROS 2 product landing page, with high-level description of ROS and links to other ROS sites

**Deprecated**

- **ROS 2 Docs**
  - API documentation up to and including Galactic

- **ROS 2 Design**
  - Early design decisions behind ROS 2 development
  - New design proposals should be submitted via ROS Enhancement Proposals (REPs)

---

### 4.9 micro-ROS Documentation

**micro-ROS** is the default embedded toolkit for ROS 2 and Vulcanexus. It provides a complete set of tools, frameworks and APIs for deploying applications in *microcontrollers (MCU)* with full support for ROS 2 communications paradigms: Nodes, Publication/Subscription, Services, etc.

micro-ROS targets MCUs with 32 bit architectures and at least 32 kB of RAM, providing support for most popular *RTOSes* and *bare metal* systems. With this in mind, micro-ROS aims to be as *lightweight* and *versatile* as possible, with a wide set of configurable parameters to tune the application memory usage and timing requirements.

micro-ROS also targets *hard real-time* environments, giving the user control over:

- Memory usage, achieving *zero dynamic memory usage* at runtime.
- *Timing* of each operation to achieve *deterministic* behavior.
- *Execution order* of the operations.
micro-ROS is **transport agnostic**. With its default middleware (*eProsima Micro XRCE-DDS*) it is possible to run micro-ROS on top of almost any transport layer that supports a package or stream communication paradigm: TCP/IP or UDP/IP stacks, UART devices, CAN/FD buses, SPI, radio links, etc.

This documentation is structured as follows:

### 4.9.1 Getting started micro-ROS

**Table of Contents**

- **Overview**
- **Prerequisites**
- **Create a micro-ROS workspace**
  - Build micro-ROS library
  - Create a package
- **Add micro-ROS apps to the workspace**
  - Write the micro-ROS publisher
  - Write the micro-ROS subscriber
  - Add dependencies
- **Build**
- **Run**
  - Start the Agent
  - Run the apps

**Overview**

This tutorial provides step-by-step instructions to use Vulcanexus micro tools to build and execute a micro-ROS publisher/subscriber demo.

**Prerequisites**

Ensure that the Vulcanexus installation includes Vulcanexus micro (either *vulcanexus-iron-desktop*, *vulcanexus-iron-micro*, or *vulcanexus-iron-base*). Also, remember to source the environment in every terminal in this tutorial.

```
source /opt/vulcanexus/iron/setup.bash
```
Create a micro-ROS workspace

A workspace is a directory containing ROS 2 packages. Before using ROS 2, it’s necessary to source your ROS 2 installation workspace in the terminal you plan to work in. This makes ROS 2’s packages available for you to use in that terminal.

```bash
source /opt/vulcanexus/iron/setup.bash

# Best practice is to create a new directory for every new workspace.
mkdir -p ~/microros_ws/src
# Another best practice is to put any packages in your workspace into the src directory.
```

```bash
cd ~/microros_ws/src
```

Build micro-ROS library

The first step is to download and build micro-ROS sources into the created workspace. Vulcanexus micro includes the `micro_ros_setup` tool, which will handle all the needed steps.

Let’s begin using the `create_firmware_ws.sh` command with `host` as target:

```bash
# Create firmware step.
ros2 run micro_ros_setup create_firmware_ws.sh host
```

This command will download all needed sources and tools to build micro-ROS as a library. A folder named `firmware` must be present in your workspace, which will old platform specific tools and source code:

```
~/microros_ws/src/
|-- firmware
|  |-- dev_ws
|   |   |-- src
|   |   |   |-- eProsima
|   |   |       |-- Micro-CDR
|   |   |       |   |-- Micro-XRCE-DDS-Client
|   |   |   |   |-- ros2
|   |   |       |-- common_interfaces
|   |   |       |-- example_interfaces
|   |   |       |-- rcl_interfaces
|   |   |       |-- unique_identifier_msgs
|   |   |   |-- uros
|   |   |       |-- micro-ROS-demos
|   |   |       |-- micro_ros_msgs
|   |   |       |-- micro_ros_utilities
|   |   |       |-- rclc
|   |   |       |-- rmw_microxrcedds
|   |   |       |-- rosidl_typesupport_microxrcedds
```

As there is no need for extra configuration steps usually needed for cross-compilation or RTOS configuration, let’s build:

```bash
# Build step.
ros2 run micro_ros_setup build_firmware.sh
```

This completes the micro-ROS library build. Remember sourcing this workspace before building or running a micro-ROS app.
# Source micro-ROS installation.
source install/local_setup.bash

Note: A set of examples apps for Linux is also present and build under the src/uros/micro-ROS-demos/rclc directory.

Create a package

Recall that packages should be created in the src directory, not the root of the workspace. So, navigate into micro-ros_ws/src, and run the package creation command:

```bash
ros2 pkg create --build-type ament_cmake micro_pubsub
```

```
micro_pubsub/
  ├── CMakeLists.txt
  ├── package.xml
  ├── include
  │   └── micro_pubsub
  └── src
```

Add micro-ROS apps to the workspace

After the workspace is ready and sourced, micro-ROS applications can be added, build and run.

Note: Certain topics as message memory handling or micro-ROS API details are not covered on this tutorial. Check the micro-ROS User API tutorial for more details.

Write the micro-ROS publisher

On this example app, a publisher will send a periodic string message using a configurable timer.

Add the publisher source code on micro_pubsub/src/micro_publisher.c:

```c
// C standard includes.
#include <stdio.h>
#include <unistd.h>

// micro-ROS general includes with general functionality.
#include <rcl/rcl.h>
#include <rcl/error_handling.h>
#include <rclc/rclc.h>
#include <rclc/executor.h>

// Include subscriber message type.
#include <std_msgs/msg/string.h>
```
// Macros to handle micro-ROS return codes, error handling should be customized for the
// target system.
#define RCCHECK(fn) { rcl_ret_t temp_rc = fn; if((temp_rc != RCL_RET_OK)){ printf("Failed status on line %d: %d. Aborting.\n", __LINE__, (int) temp_rc); return 1; }}
#define RCSOFTCHECK(fn) { rcl_ret_t temp_rc = fn; if((temp_rc != RCL_RET_OK)){ printf("Failed status on line %d: %d. Continuing.\n", __LINE__, (int) temp_rc); }}

// Set maximum publisher string length.
#define STRING_LEN 200

// The publisher and string message objects are declared as global so they are available
// on `timer_callback`
DECLARE_RCL_PUBLISHER(publisher);
DECLARE_STD_MSGS_STRING(msg);
int counter = 0;

// The `timer_callback` function will be in charge of publishing and increment the
// message data on each timer period.
void timer_callback(rcl_timer_t * timer, int64_t last_call_time)
{
    (void) last_call_time;

    if (timer != NULL)
    {
        // Update string message with new value and size before its published.
        sprintf(msg.data.data, "Hello from micro-ROS #%d", counter++);
        msg.data.size = strlen(msg.data.data);

        RCSOFTCHECK(rcl_publish(&publisher, &msg, NULL));
        printf("Publishing: \"%s\"\n", msg.data.data);
    }
}

int main()
{
    // Get configured allocator.
    rcl_allocator_t allocator = rcl_get_default_allocator();

    // Initialize support object.
    rclc_support_t support;
    RCCHECK(rclc_support_init(&support, 0, NULL, &allocator));

    // Node name and namespace.
    const char * node_name = "minimal_micro_publisher";
    const char * node_namespace = ";"

    // Create node with default configuration.
    rcl_node_t node;
    RCCHECK(rclc_node_init_default(&node, node_name, node_namespace, &support));

    // Topic name and type support.
    const char * topic_name = "topic";
```c
const rosidl_message_type_support_t * type_support =
    ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, String);

// Create publisher.
RCHECK(rclc_publisher_init_default(
    &publisher,
    &node,
    type_support,
    topic_name));

// Initialize publisher message memory.
char string_memory[STRING_LEN];
msg.data.data = &string_memory[0];
msg.data.size = 0;
msg.data.capacity = STRING_LEN;

// Create timer with its callback and its trigger period.
rcl_timer_t timer;
const unsigned int timer_period = 500;
RCHECK(rclc_timer_init_default(
    &timer,
    &support,
    RCL_MS_TO_NS(timer_period),
    timer_callback));

// Initialize executor with one handle.
rclc_executor_t executor;
const size_t number_of_handles = 1;
RCHECK(rclc_executor_init(&executor, &support.context, number_of_handles, &
                           allocator));

// Add timer callback to the executor.
RCHECK(rclc_executor_add_timer(&executor, &timer));

// Spin forever
RCHECK(rclc_executor_spin(&executor));

// Free used resources on spin error and exit.
  // This methods will free used memory even if connection with the Agent is lost.
RCHECK(rclc_executor_fini(&executor));
RCHECK(rcl_timer_fini(&timer));
RCHECK(rcl_publisher_fini(&publisher, &node));
RCHECK(rcl_node_fini(&node));
RCHECK(rclc_support_fini(&support));

return 1;
```
Write the micro-ROS subscriber

This subscriber example is very similar to the publisher, but here the message reception will be handled by the subscriber callback if there is enough memory available.

Add the subscriber source code on micro_pubsub/src/micro_subscriber.c:

```c
// C standard includes.
#include <stdio.h>
#include <unistd.h>

// micro-ROS general includes with general functionality.
#include <rcl/rcl.h>
#include <rcl/error_handling.h>
#include <rclc/rclc.h>
#include <rclc/executor.h>

// Include subscriber message type.
#include <std_msgs/msg/string.h>

// Macros to handle micro-ROS return codes, error handling should be customized for the target system.
#define RCCHECK(fn) { rcl_ret_t temp_rc = fn; if((temp_rc != RCL_RET_OK)){ printf("Failed status on line %d: %d. Aborting.\n", __LINE__, (int) temp_rc); return 1; }}
#define RCSOFTCHECK(fn) { rcl_ret_t temp_rc = fn; if((temp_rc != RCL_RET_OK)){ printf("Failed status on line %d: %d. Continuing.\n", __LINE__, (int) temp_rc); }}

// Set maximum received string length. Received publications with strings larger than this size will be discarded.
#define STRING_LEN 200

// Callback to handle incoming subscriber messages.
void subscription_callback(const void * msgin)
{
    // Message type shall be casted to expected type from void pointer.
    const std_msgs__msg__String * msg = (const std_msgs__msg__String *)msgin;
    printf("I heard: "%s
", msg->data.data);
}

int main()
{
    // Get configured allocator.
    rcl_allocator_t allocator = rcl_get_default_allocator();

    // Initialize support object.
    rclc_support_t support;
    RCCHECK(rclc_support_init(&support, 0, NULL, &allocator));

    // Node name and namespace.
    const char * node_name = "minimalMicroSubscriber";
    const char * node_namespace = "";

    // Create node.
```
rcl_node_t node;
RCCHECK(rclc_node_init_default(&node, node_name, node_namespace, &support));

// Topic name and message type support.
const char * topic_name = "topic";
const rosidl_message_type_support_t * type_support =
ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, String);

// Create subscriber.
rcl_subscription_t subscriber;
RCCHECK(rclc_subscription_init_default(
   &subscriber,
   &node,
   type_support,
   topic_name));

// Initialize executor.
rclc_executor_t executor;
const size_t number_of_handles = 1;
RCCHECK(rclc_executor_init(&executor, &support.context, number_of_handles, &allocator));

// Add subscriber callback.
std_msgs__msg__String msg;
RCCHECK(rclc_executor_add_subscription(&executor, &subscriber, &msg, &subscription_callback, ON_NEW_DATA));

// Initialize subscriber message memory.
char string_memory[STRING_LEN];
msg.data.data = &string_memory[0];
msg.data.size = 0;
msg.data.capacity = STRING_LEN;

// Spin forever
RCSOFTCHECK(rclc_executor_spin(&executor));

// Free used resources on spin error and exit.
// This methods will free used memory even if connection with the Agent is lost.
RCSOFTCHECK(rclc_executor_fini(&executor));
RCSOFTCHECK(rcl_subscription_fini(&subscriber, &node));
RCSOFTCHECK(rcl_node_fini(&node));
RCSOFTCHECK(rclc_support_fini(&support));

return 1;
}
Add dependencies

Before building this examples, micro-ROS used headers and tools need to be included on the created ROS2 package:

1. Open package.xml on microros_ws/src/micro_pubsub directory
2. make sure to fill in the <description>, <maintainer> and <license> tags:

   ```xml
   <description>Examples of minimal publisher/subscriber using rclcpp</description>
   <maintainer email="you@email.com">Your Name</maintainer>
   <license>Apache License 2.0</license>
   ``

3. Add a new line after the ament_cmake buildtool dependency and paste the following dependencies corresponding to your node’s include statements:

   ```xml
   <depend>rcl</depend>
   <depend>rclc</depend>
   <depend>std_msgs</depend>
   <depend>rmw_microxrcedds</depend>
   ``

4. Add the dependencies to the CMakeLists.txt file, the following example can be used:

   ```
cmake_minimum_required(VERSION 3.5)
project(micro_pubsub)

find_package(ament_cmake REQUIRED)
find_package(rcl REQUIRED)
find_package(rclc REQUIRED)
find_package(std_msgs REQUIRED)
find_package(rmw_microxrcedds REQUIRED)

add_executable(publisher src/micro_publisher.c)
ament_target_dependencies(publisher
  rcl
  rclc
  std_msgs
  rmw_microxrcedds)

add_executable(subscriber src/micro_subscriber.c)
ament_target_dependencies(subscriber
  rcl
  rclc
  std_msgs
  rmw_microxrcedds)

install(TARGETS
  publisher
  subscriber
  DESTINATION lib/${PROJECT_NAME})

ament_package()
```
Build

Once the source code is ready, build your new package:

```
# ~/microros_ws
colcon build --packages-select micro_pubsub
```

Run

Start the Agent

The agent is included on Vulcanexus micro tool set. The default transport configuration for micro-ROS is UDPv4 with the Agent on `127.0.0.1:8888`:

```
# Start the agent
ros2 run micro_ros_agent micro_ros_agent udp4 --port 8888 -v5
```

This command will start the Agent, allowing communication between our micro-ROS apps and ROS2 environment.

Note: At the top of the apps main function, micro-ROS is initialized using the `rclc_support_init` method. It is important that the Agent is reachable on this step, as the method will fail if a connection cannot be established.

Run the apps

Before running the apps, set Micro XRCE-DDS as RMW implementation for each terminal:

```
# Set Micro XRCE-DDS as RMW implementation
export RMW_IMPLEMENTATION=rmw_microxrcedds
```

Run the publisher in one terminal:

```
# ~/microros_ws
ros2 run micro_pubsub publisher
```

Then run listener in another terminal:

```
# ~/microros_ws
ros2 run micro_pubsub subscriber
```

The listener will start printing messages to the console. Notice how the creation of entities and each publication appears on the Agent log.
4.9.2 Build System Components

- Renesas e² Studio
- Espressif ESP-IDF
- Zephyr RTOS
- ARM MBed RTOS
- NuttX RTOS
- Microsoft Azure RTOS
- TI Tiva C Series
- ST Micro ST Cube IDE/MX
- Platform.IO
- Arduino
- Raspberry Pi Pico

Renesas e² Studio

Renesas RA Family is the official supported hardware of micro-ROS. This family of MCUs provides a wide range of features based on Arm® Cortex®-M33, M23, and M4 processor cores.

Renesas e² Studio provides a whole integrated development environment that allows for professional development, test and profiling of embedded applications. The micro-ROS component for Renesas provides a set of tools and instructions to integrate the build and configuration of micro-ROS within a Renesas e² Studio project.

This package supports multiple RTOSes (FreeRTOS and Azure RTOS), as well as a bare-metal approach. Also, a wide range of transport layers are available: UART, UDP/IP, TCP/IP, USB-CDC and CAN/FD.

Note: For detailed instructions on how to use micro-ROS with this platform, please refer to micro-ROS for Renesas e² Studio Github repository or use your Vulcanexus distribution to download this module:

```bash
source /opt/vulcanexus/iron/setup.bash
ros2 run micro_ros_setup component renesas_e2_studio
```
**Espressif ESP-IDF**

micro-ROS component for ESP-IDF allows the integration of the micro-ROS stack in one of the most famous WiFi-enabled MCUs: the ESP32 family.

By means of this component, it is easy to integrate, configure and deploy a micro-ROS application that can communicate with the ROS 2 dataspace over WiFi, UART or USB-CDC. By default, this toolchain integrates FreeRTOS as framework and allows the user to run micro-ROS tasks simultaneously along with other user process.

Most of the ESP32 versions are supported: ESP32, ESP32-S2, ESP32-S3 and even the RISC-V based MCU of Espressif, the ESP32-C3.

**Note:** For detailed instructions on how to use micro-ROS with this platform visit micro-ROS component for ESP-IDF Github repository or use your Vulcanexus distribution to download this module:

```
source /opt/vulcanexus/iron/setup.bash
ros2 run micro_ros_setup component esp_idf
```

---

**Zephyr RTOS**

Zephyr RTOS is an RTOS for embedded systems supported by the Linux Foundation. It provides a full featured and layered architecture RTOS ready for deploy production-ready applications. Thanks to the huge amount of ready-to-use drivers, it will be easy to integrate connectivity solutions such as Bluetooth BLE, WiFi, USB or CAN.

With the micro-ROS module for Zephyr RTOS, ROS 2 users will find easy to integrate an embedded micro-ROS application in the Zephyr ecosystem. This module support a considerable part of the list of supported boards by Zephyr.

**Note:** For detailed instructions on how to use micro-ROS with this platform visit micro-ROS module for Zephyr Github repository or use your Vulcanexus distribution to download this module:
ARM MBed RTOS

micro-ROS provides a module for integration ROS 2 embedded application in ARM MBed RTOS. A basic UART transport is provided for version v6.8 and beyond.

Note: For detailed instructions on how to use micro-ROS with this platform visit micro-ROS module for Mbed RTOS Github repository or use your Vulcanexus distribution to download this module:

```
source /opt/vulcanexus/iron/setup.bash
ros2 run micro_ros_setup component zephyr_rtos
```

NuttX RTOS

NuttX is one of the most complete and mature RTOS available. It is supported by Apache Foundation and provides a piece of software with emphasis on standard compliance and small footprint. It is a powerful and flexible RTOS that can be used for embedded applications. Also, it is compliant with POSIX and ANSI standards.

micro-ROS component for NuttX provides an example application environment where the micro-ROS stack is integrated in the NuttX 10 build system and can be ran as a NuttX application with an UART transport.

Note: For detailed instructions on how to use micro-ROS with this platform visit micro-ROS app for Nuttx RTOS Github repository or use your Vulcanexus distribution to download this module:
Microsoft Azure RTOS

Azure RTOS is the embedded IoT development environment of Microsoft. It provides a huge amount of solutions and ready-to-use libraries with focus in cloud applications and connectivity.

By means of this module, micro-ROS is integrated in ThreadX (the Azure RTOS scheduler), using NetX (the Azure RTOS network stack) to provide a UDP/IP transport.

**Note:** For detailed instructions on how to use micro-ROS with this platform visit micro-ROS app for Microsoft Azure RTOS Github repository or use your Vulcanexus distribution to download this module:

```bash
source /opt/vulcanexus/iron/setup.bash
ros2 run micro_ros_setup component azure_rtos
```

TI Tiva C Series

Texas Instruments Tiva C Series is a family of MCU based on ARM Cortex-M4F. micro-ROS provides support for this platform and a basic USB-CDC transport.

**Note:** For detailed instructions on how to use micro-ROS with this platform visit micro-ROS app for TI Tiva™ C Series TM4C123G Github repository or use your Vulcanexus distribution to download this module:

```bash
source /opt/vulcanexus/iron/setup.bash
ros2 run micro_ros_setup component tiva_c_series
```
STM32Cube is one of the preferred options for using the ST Micro STM32 family. By means of this module, the micro-ROS user will find easy to integrate the micro-ROS stack in a STM32Cube project.

All the STM32 MCUs are supported by this IDE and most of them should be able to run the micro-ROS stack. By now, multiple Nucleo boards featuring STM32F4 and STM42F7 devices has been tested a proven to work. By using this module the micro-ROS user will find it easy to port micro-ROS to its own STM32 with FreeRTOS and the provided serial based transport.

Note: For detailed instructions on how to use micro-ROS with this platform visit micro-ROS for STM32CubeMX/IDE Github repository or use your Vulcanexus distribution to download this module:

```
source /opt/vulcanexus/iron/setup.bash
ros2 run micro_ros_setup component stm32cube
```

Platform.IO

Platform.IO is one of the most popular collaborative platform for embedded development. It provides a large set of supported platforms, frameworks and libraries for embedded development. All of this based with an intuitive configuration procedure and an automated toolchain installation.

By using this module, micro-ROS user will be able to integrate micro-ROS in their Platform.IO projects just by including a couple of configuration lines. Some of the out-of-the-box supported transport are: UART, WiFi and Ethernet. Also, the micro-ROS user will find easy to port micro-ROS to the large list of supported platforms.

Note: For detailed instructions on how to use micro-ROS with this platform visit micro-ROS for PlatformIO Github repository or use your Vulcanexus distribution to download this module:
source /opt/vulcanexus/iron/setup.bash
ros2 run micro_ros_setup component platformio

Arduino

Arduino is the reference development framework to introduce new users to the embedded world. Using their user-friendly IDE and tools, Arduino users can easily go from their first blinky LED to their own micro-ROS applications.

micro-ROS for Arduino provides a set of prebuilt libraries for reference platforms so the user do not have to handle the micro-ROS build procedure. Also, for advanced users, instructions for customizing the installation and recompiling the library are provided.

Note: For detailed instructions on how to use micro-ROS with this platform visit micro-ROS for Arduino Github repository or use your Vulcanexus distribution to download this module:

source /opt/vulcanexus/iron/setup.bash
ros2 run micro_ros_setup component arduino
Raspberry Pi Pico

Raspberry Pi Pico is the versatile embedded and low cost solution of **Raspberry Pi for microcontroller environments**. It features a dual core Cortex-M0+ based silicon named RP2040, which include a wide variety of peripherals such as I2C, SPI, UART or GPIO, and even a hardware programmable module named PIO. Also a full featured API for hardware abstraction are provided within the SDK.

micro-ROS module for Raspberry Pi Pico SDK provides a set of prebuilt libraries so the user do not have to handle the micro-ROS build procedure. Also, for advanced users, instructions for customizing the installation and recompiling the library are provided.

**Note:** For detailed instructions on how to use micro-ROS with this platform visit [micro-ROS for Raspberry Pi Pico Github repository](https://github.com/micro-ros/rpi_pico), or use your Vulcanexus distribution to download this module:

```bash
source /opt/vulcanexus/iron/setup.bash
ros2 run micro_ros_setup component raspberry_pi_pico
```

### 4.9.3 micro-ROS User API

In this section, you’ll learn the basics of the micro-ROS C API: `rclc`. This API implementation is based on the ROS 2 client support library (RCL), enriched with a set of convenience functions by the package RCLC. That is, RCLC does not add a new layer of types on top of rcl (like RCLCPP and RCLPY do) but only provides functions that ease the programming with the RCL types. New types are introduced only for concepts that are missing in RCL, such as the concept of an executor.

This API implements an lightweight embedded version of the complete ROS 2 communications paradigm:

#### Nodes

ROS 2 nodes are the main participants on ROS 2 ecosystem. They will communicate between each other using publishers, subscriptions, services, etc. Further information about ROS 2 nodes can be found on [Understanding nodes](#).

#### Initialization

Before a node is created, a `rclc_support_t` object needs to be created and initialized. The `rclc_support_init` method will handle micro-ROS initial configuration (memory initialization, transport configuration, ...).

**Note:** The `rclc_support_init` function will try to establish communication with the Agent and will fail if its not reachable on a configurable time. A good practice is to the ping the Agent to check the connection before calling this method.

- Initialize micro-ROS and create a node with default configuration:
```
// Get configured allocator
rcl_allocator_t allocator = rcl_get_default_allocator();

// Initialize support object
rclc_support_t support;
rclc_support_init(&support, argc, argv, &allocator);

// Node name
const char * node_name = "test_node";

// Node namespace (Can remain empty: "")
const char * namespace = "test_namespace";

// Init default node
rcl_node_t node;
rclc_node_init_default(&node, node_name, namespace, &support);
```

- **Configure the node Domain ID:**

ROS2 allows isolation of nodes through the ROS_DOMAIN_ID concept. Nodes can only discover and communicate nodes placed on the same domain. Further information about ROS 2 Domain ID can be found on [About Domain ID](#).

A node can be configured using the rclc_support_t internal rcl_init_options_t options structure. rcl provides the rcl_init_options_set_domain_id method to set a node domain. Example:

```
// Get configured allocator
rcl_allocator_t allocator = rcl_get_default_allocator();

// Initialize and modify options (Set DOMAIN ID to 10)
rcl_init_options_t init_options = rcl_get_zero_initialized_init_options();
rcl_init_options_init(&init_options, allocator);
rcl_init_options_set_domain_id(&init_options, 10);

// Initialize rclc support object with custom options
rclc_support_t support;
rclc_support_init_with_options(&support, 0, NULL, &init_options, &allocator);

// Node name
const char * node_name = "test_node";

// Node namespace (Can remain empty "")
const char * namespace = "test_namespace";

// Init node with configured support object
rcl_node_t node;
rclc_node_init_default(&node, node_name, namespace, &support);
```

**Note:** The configuration of the node will also be applied to its future elements (publishers, subscribers, services, ...).
Cleaning Up

All entities owned by a node (publishers, subscribers, services, ...) have to be destroyed before the node itself:

```c
// Destroy created entities (Example)
rcl_publisher_fini(&publisher, &node);

// Destroy the node
rcl_node_fini(&node);
```

This will delete the node from ROS2 graph, including any generated infrastructure on the agent (if possible) and used memory on the client.

Publishers and subscribers

Table of Contents

- Publisher
  - Initialization
  - Publish a message
- Subscription
  - Initialization
  - Callback
- Message initialization
- Cleaning Up

ROS 2 publishers and subscribers are the basic communication mechanism between nodes using topics. Further information about ROS 2 publish-subscribe pattern can be found on Understanding topics.

Ready to use code related to this concepts can be found on micro-ROS demos repository int32_publisher and int32_subscriber examples.

Note: micro-ROS publishers and subscribers can be configured using quality-of-service. For a better understanding of DDS quality-of-service, check About Quality of Service settings.

Publisher

Initialization

Starting from a code where `rclc_support_t` is initialized and a micro-ROS node is created, there are three ways to initialize a publisher depending on the desired quality-of-service configuration:

```c
// Topic name
const char * topic_name = "test_topic";
```

(continues on next page)
// Get message type support
const rosidl_message_type_support_t * type_support =
  ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Int32);

// Create a reliable publisher ('node' is available and already initialized)
const rcl_publisher_t reliable_publisher;
rclc_publisher_init_default(
  &reliable_publisher, &node,
  &type_support, &topic_name);

// Create a best effort publisher
const rcl_publisher_t best_effort_publisher;
rclc_publisher_init_best_effort(
  &best_effort_publisher, &node,
  &type_support, &topic_name);

// Create a custom QoS publisher
const rcl_publisher_t custom_publisher;
rclc_publisher_init(
  &custom_publisher, &node,
  &type_support, &topic_name, &rmw_qos_profile_default);

Publish a message

To publish a message on a successfully initialized publisher:

// Int32 message object
std_msgs__msg__Int32 msg;

// Set message value
msg.data = 0;

// Publish message
rcl_publish(&publisher, &msg, NULL);

For periodic publications, rcl_publish can be placed inside a timer callback. Check the Executor and timers section for details.

Subscription

Initialization

The subscription initialization is almost identical to the publisher one:

// Topic name
const char * topic_name = "test_topic";

// Get message type support
const rosidl_message_type_support_t * type_support =
ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Int32);

// Create a reliable subscriber ('node' is available and already initialized)
rcf_subscription_t reliable_subscriber;
rclc_subscription_init_default(
    &reliable_subscriber, &node,
    &type_support, &topic_name);

// Create a best effort subscriber
rcf_subscription_t best_effort_subscriber;
rclc_subscription_init_best_effort(
    &best_effort_subscriber, &node,
    &type_support, &topic_name);

// Create a custom QoS subscriber
rcf_subscription_t custom_subscriber;
rclc_subscription_init(
    &custom_subscriber, &node,
    &type_support, &topic_name, qos_profile);

Callback

The executor is responsible to call the configured callback when a publication is received. The callback will have a pointer to the received message as its only argument, containing the values received by the subscriber:

// Function prototype:
void (* rclc_subscription_callback_t)(const void *);

// Implementation example:
void subscription_callback(const void *msgin)
{
    // Cast message pointer to expected type
    const std_msgs__msg__Int32 * msg = (const std_msgs__msg__Int32 *) msgin;

    // Process message
    printf("Received: %d\n", msg->data);
}

Once the subscriber and the executor are initialized, the subscriber and its callback must be added to the executor to receive incoming publications once its spinning:

// Message object to receive publisher data
std_msgs__msg__Int32 msg;

// Add subscriber to the executor ('executor' and 'subscriber' are available and already initialized)
rclc_executor_add_subscription(
    &executor, &subscriber, &msg,
    subscription_callback, ON_NEW_DATA);

// Spin executor to receive messages
A subscription can also be removed from the executor after it is not longer needed:

```c
// Remove a subscription from an executor
rclc_executor_remove_subscription(
    &executor, &subscription);
```

**Message initialization**

Before publishing or receiving a message, it may be necessary to initialize its memory for types with strings or sequences.

**Note:** Check the *Handling messages memory tutorial* for details.

**Cleaning Up**

After finishing the publisher/subscriber, the node will no longer advertise that it is publishing/listening on the topic. To destroy an initialized publisher or subscriber:

```c
// Destroy publisher
rcl_publisher_fini(&publisher, &node);

// Destroy subscriber
rcl_subscription_fini(&subscriber, &node);
```

This will delete any automatically created infrastructure on the agent (if possible) and free used memory on the client side.

**Services**

Table of Contents

- **Server**
  - Initialization
  - Callback
- **Client**
  - Initialization
  - Callback
  - Send a request
- **Message initialization**
- **Cleaning Up**
ROS 2 services are another communication mechanism between nodes. Services implement a client-server paradigm based on ROS 2 messages and types. Further information about ROS 2 services can be found on Understanding services.

Ready to use code related to this concepts can be found on micro-ROS demos repository addtwoints_server and addtwoints_client examples.

**Server**

### Initialization

Services initialization follow the same pattern as publishers and subscribers:

```c
// Service name
const char * service_name = "test_service";

// Get message type support
const rosidl_message_type_support_t * type_support =
   ROSIDL_GET_SRV_TYPE_SUPPORT(example_interfaces, srv, AddTwoInts);

// Create a reliable service server ('node' is available and already initialized)
rc1_service_t reliable_service;
rcl_service_init_default(
   &reliable_service, &node,
   type_support, service_name);

// Create a best effort service server
rc1_service_t best_effort_service;
rcl_service_init_best_effort(
   &best_effort_service, &node,
   type_support, service_name);

// Create a custom QoS service server
rc1_service_t custom_service;
rcl_service_init(
   &custom_service, &node, type_support,
   service_name, &rmw_qos_profile_services_default);
```

### Callback

Once a request arrives, the executor will call the configured callback with the request and response messages as arguments. The request message `request_msg` contains the request sent by the client, while the `response_msg` argument should be modified with the server response. The response message will be sent to the client after the callback returns.

Using `AddTwoInts.srv` type definition as an example:

```c
int64 a
int64 b
---
int64 sum
```

The client request message will contain two integers a and b, and expects the `sum` of those values as the response:
// Function prototype:
void (* rclc_service_callback_t)(const void *, void *);

// Implementation example:
void service_callback(const void * request_msg, void * response_msg)
{
    // Cast messages to expected types
    example_interfaces__srv__AddTwoInts_Request * req_in =
        (example_interfaces__srv__AddTwoInts_Request *) request_msg;
    example_interfaces__srv__AddTwoInts_Response * res_in =
        (example_interfaces__srv__AddTwoInts_Response *) response_msg;

    // Handle request message and set the response message values
    printf("Client requested sum of %d and %d.\n",
        (int) req_in->a, (int) req_in->b);
    res_in->sum = req_in->a + req_in->b;
}

As in the subscriber, the service and its callback must be added to the executor in order to process incoming requests on a the executor spin:

// Service message objects
example_interfaces__srv__AddTwoInts_Request request_msg;
example_interfaces__srv__AddTwoInts_Response response_msg;

// Add service server to the executor (executor' and 'service' are available and already initialized)
rclc_executor_add_service(
    &executor, &service, &request_msg,
    &response_msg, service_callback);

// Spin executor to receive requests
rclc_executor_spin(&executor);

A service server can also be removed from the executor after it is not longer needed:

// Remove a service server from an executor
rclc_executor_remove_service(
    &executor, &service);

Client

Initialization

// Service name
const char * service_name = "test_service";

// Get message type support
const rosidl_message_type_support_t * type_support =
    ROSIDL_GET_SRV_TYPE_SUPPORT(example_interfaces, srv, AddTwoInts);

// Create a reliable service client (node' is available and already initialized)
```c
rcl_client_t reliable_client;
rcl_client_init_default(
    &reliable_client, &node,
    type_support, service_name);

// Create a best effort service client
rcl_client_t best_effort_client;
rcl_client_init_best_effort(
    &best_effort_client, &node,
    type_support, service_name);

// Create a custom QoS service client
rcl_client_t custom_client;
rcl_client_init(
    &custom_client, &node, type_support,
    service_name, &rmw_qos_profile_services_default);
```

### Callback

The function will have the response message as its only argument, containing the response sent by the server:

```c
// Function prototype:
void (* rclc_client_callback_t)(const void *);

// Implementation example:
void client_callback(const void * response_msg)
{
    // Cast response message to expected type
    example_interfaces__srv__AddTwoInts_Response * msgin =
        (example_interfaces__srv__AddTwoInts_Response *) response_msg;

    // Handle response message
    printf("Received service response %ld + %ld = %ld\n", req.a, req.b, msgin->sum);
}
```

Adding the service client to the executor:

```c
// Response message object
example_interfaces__srv__AddTwoInts_Response res;

// Add service client to the executor ('executor' and 'client' are available and already initialized)
rclc_executor_add_client(&executor, &client, &res, client_callback);

// Spin executor to receive requests
rclc_executor_spin(&executor);
```

A service client can also be removed from the executor after it is not longer needed:

```c
// Remove a service client from an executor
rclc_executor_remove_client( (continues on next page)```
Send a request

Once the service client and server are configured, the service client can send a request and spin the executor to get the reply. Following the example on `AddTwoInts.srv`:

```c
// Request message object (Must match initialized client type support)
example_interfaces__srv__AddTwoInts_Request request_msg;

// Initialize request message memory and set its values
example_interfaces__srv__AddTwoInts_Request__init(&request_msg);
request_msg.a = 24;
request_msg.b = 42;

// Sequence number of the request (Populated in 'rcl_send_request')
int64_t sequence_number;

// Send request
rcl_send_request(&client, &request_msg, &sequence_number);

// Spin the executor to get the response
rclc_executor_spin(&executor);
```

Message initialization

Before publishing or receiving a message, it may be necessary to initialize its memory for types with strings or sequences.

**Note:** Check the *Handling messages memory tutorial* for details.

Cleaning Up

To destroy an initialized service or client:

```c
// Destroy service server and client
rcl_service_fini(&service, &node);
rcl_client_fini(&client, &node);
```

This will delete any automatically created infrastructure on the agent (if possible) and free used memory on the client side.
Executor and timers

Table of Contents

- **Timers**
  - Initialization
  - Callback
  - Cleaning Up
- **Executor**
  - Initialization
  - Add/Remove entities
  - Spinning
  - Cleaning Up

Timers

Timers can be created and added to the executor, which will periodically execute the timer callback once it is spinning. Timers are usually used to handle periodic publications or events.

Initialization

Starting from a code where `rclc_support_t` is initialized, a timer can be initialized given a callback and its trigger period in nanoseconds:

```c
// Timer period on nanoseconds
const unsigned int timer_period = RCL_MS_TO_NS(1000);

// Create and initialize timer
rcl_timer_t timer;
rclc_timer_init_default(&timer, &support, timer_period, timer_callback);

// Add to the executor
rclc_executor_add_timer(&executor, &timer);
```

Callback

The callback gives a pointer to the associated timer and the time elapsed since the previous call or since the timer was created on the first iteration.

```c
// Function prototype
void (* rcl_timer_callback_t)(rcl_timer_t *, int64_t);

// Implementation example
void timer_callback(rcl_timer_t * timer, int64_t last_call_time)
```

(continues on next page)
{  
    // Check last triggered timestamp  
    printf("Last callback time: %ld\n", last_call_time);  

    if (timer != NULL)  
    {  
        // Handle periodic event  
        // Handle periodic event  
    }  
}

The timer can be canceled or have its period and/or callback modified using the passed pointer. Check rcl/timer.h for details.

**Cleaning Up**

To destroy an initialized timer:

```c
// Destroy timer  
rcl_timer_fini(&timer);
```

This will free used memory and make the timer invalid

**Executor**

The rclc executor allows to handle the execution of callbacks of entities such as subscriptions, timers, services and client taking into account the required priorities.

**Initialization**

The maximum number of handles of an executor is configured on its initialization. A handle can be defined as a single event or callback: subscriptions, timers, services, clients and guard conditions. Nodes and publishers are excluded as they do not trigger input events.

```c
// Get configured allocator  
rcl_allocator_t allocator = rcl_get_default_allocator();

// Initialize support object  
rclc_support_t support;  
rclc_support_init(&support, argc, argv, &allocator);

// Initialize executor  
rclc_executor_t executor;  
const size_t number_of_handles = 1;  
rclc_executor_init(&executor, &support.context, number_of_handles, &allocator);
```

**Note:** Complex entities with underlying services or subscriptions define their expected number of handles, as for example the parameter server.
Add/Remove entities

As explained on their respective Callback sections, entities can be added and removed from an executor. The executor will trigger the callbacks of the added entities, so periodic events or incoming messages can be handled. Removed entities will free the used executor handles and their callback will be removed from the executor events.

Note: Complex entities as the parameter server or action client/servers cannot be removed from an executor.

Spinning

The executor implements the ROS2 spin mechanism. During a spin, periodic events and incoming messages will be handle, invoking the respective callbacks for each event. The following spin methods are available:

- Spin: Endless spin, will block and check for events using a busy wait approach.

```c
// Spin endlessly
rclc_executor_spin(&executor);
```

- Spin period: Endless periodic spin. This method will block and check for events using periodically. If an event is triggered, the method will sleep for the remaining period time.

```c
const unsigned int spin_period = RCL_MS_TO_NS(1000);

// Spin endlessly
rclc_executor_spin_period(&executor, spin_period);
```

- Spin some: Spin with timeout. Will check for events with a maximum timeout. If an event is triggered within the specified timeout, the method will exit.

```c
const unsigned int spin_timeout = RCL_MS_TO_NS(100);

// Spin with timeout
rclc_executor_spin_some(&executor, spin_timeout);
```

Cleaning Up

To destroy an initialized executor:

```c
// Destroy executor
rclc_executor_fini(&executor);
```

This will free used memory and make the executor invalid.
Actions

**Table of Contents**
- *Action Server*
  - Initialization
  - Callbacks
- *Action Client*
  - Initialization
  - Send a request
  - Callbacks
- Message initialization
- Cleaning Up

ROS 2 actions are a communication mechanism between two nodes. Actions implement a client-server paradigm with feedback based on ROS 2 messages and types. Further information about ROS 2 actions can be found on *Understanding actions.*

Ready to use code related to this concepts can be found on micro-ROS demos repository [fibonacci_action_client](#) and [fibonacci_action_server](#) examples.

**Action Server**

micro-ROS action servers allows the user to accept or reject ROS 2 action clients goals, send feedback on their execution, and a final result.

**Initialization**

Action server initialization can be done using the following API:

```c
// Action name
const char * action_name = "fibonacci";

// Get action type support
const rosidl_action_type_support_t * type_support =
  ROSIDL_GET_ACTION_TYPE_SUPPORT(example_interfaces, Fibonacci);

// Create a reliable action server ('node' and 'support' is available and already initialized)
rclc_action_server_t action_server;
rcic_action_server_init_default(
  &action_server,
  &node,
  &support,
  type_support,
  action_name
);
```
Callbacks

Once an action goal request arrives, the executor will execute the configured handle_goal callback with the received goal request and the configured context as arguments. Using the received goal_handle, the action server can accept or reject the request returning RCL_RET_ACTION_GOAL_REJECTED or RCL_RET_ACTION_GOAL_ACCEPTED.

Using Fibonacci.action type definition as an example:

```c
# Goal
int32 order
---
# Result
int32[] sequence
---
# Feedback
int32[] sequence
```

The client request message will contain an integers order:

```c
// Function prototype:
void (* handle_goal_callback_t)(rclc_action_goal_handle_t *, void *);

// Implementation example:
int rcl_ret_t handle_goal(rclc_action_goal_handle_t * goal_handle, void * context)
{
    (void) context;

    example_interfaces__action__Fibonacci_SendGoal_Request * req =
        (example_interfaces__action__Fibonacci_SendGoal_Request *) goal_handle->ros_goal_request;

    // Too big, rejecting
    if (req->goal.order > 200) {
        return RCL_RET_ACTION_GOAL_REJECTED;
    }

    // Activate here the goal processing task
    return RCL_RET_ACTION_GOAL_ACCEPTED;
}
```

If the goal has been accepted, during the processing task the action server can send feedback to the client using the goal handle:

```c
// Generate and fill feedback
eexample_interfaces__action__Fibonacci_FeedbackMessage feedback = {};
// Publish feedback message
rclc_action_publish_feedback(goal_handle, &feedback);
```

When the requested goal is completed, the action server can send the result to the client using the goal handle and its finished state: GOAL_STATE_SUCCEEDED, GOAL_STATE_CANCELED or GOAL_STATE_ABORTED.

```c
// Generate and fill result message
eexample_interfaces__action__Fibonacci_GetResult_Response response = {0};
```
rcl_action_goal_state_t goal_state = GOAL_STATE_SUCCEEDED;
// Send result
rclc_action_send_result(goal_handle, goal_state, &response);

Note: The executor related to the action server shall spin to complete a successful \texttt{rclc\_action\_send\_result} call.

Also, during the goal handling, the action client can request the cancellation of the goal. If the client requests the cancellation, the action server can decide if the goal can be cancelled using a callback:

```c
// Function prototype:
void (* cancel_goal_callback_t)(rclc_action_goal_handle_t *, void *);

bool handle_cancel(rclc_action_goal_handle_t * goal_handle, void * context)
{
  if (/* goal can be cancelled */) {
    return true;
  }
  return false;
}
```

Finally, all the callbacks can be added to the executor when the action server is added:

```c
#define NUMBER_OF_SIMULTANEOUS_HANDLES 10

// Goal request storage
element_interfaces__action__Fibonacci_SendGoal_Request ros_goal_request[NUMBER_OF_SIMULTANEOUS_HANDLES];

// Add action server to the executor (executor and action_server are available and already initialized)
rclc_executor_add_action_server(
  &executor,
  &action_server,
  NUMBER_OF_SIMULTANEOUS_HANDLES,
  ros_goal_request,
  sizeof(example_interfaces__action__Fibonacci_SendGoal_Request),
  handle_goal,    // Goal request callback
  handle_cancel,  // Goal cancel callback
  NULL            // Context
);

// Spin executor to receive requests
rclc_executor_spin(&executor);
```

Note: An action server cannot be removed from the executor.
Action Client

micro-ROS action client allows the user to send ROS 2 action goal requests to an action server, receive feedback and results or request a goal cancellation.

Initialization

Action client initialization can be done using the following API:

```c
// Action name
const char * action_name = "fibonacci";

// Get action type support
const rosidl_action_type_support_t * type_support = 
    ROSIDL_GET_ACTION_TYPE_SUPPORT(example_interfaces, Fibonacci);

// Create a reliable action client ('node' is available and already initialized)
rclc_action_client_t action_client;
rclc_action_client_init_default(
    &action_client, 
    &node, 
    type_support, 
    action_name 
);
```

Send a request

An action client can send a goal request to the server and spin the executor to get the response.

Using `Fibonacci.action` type definition as an example:

```c
# Goal
int32 order
---
# Result
int32[] sequence
---
# Feedback
int32[] sequence
```

The client request message will contain an integers order:

```c
// Fill goal request message
e.exampleInterfaces__action__Fibonacci_SendGoal_Request ros_goal_request;
ros_goal_request.goal.order = 10;

// Send goal request
rclc_action_send_goal_request(&action_client, &ros_goal_request, NULL);

// Spin the executor to get the response
rclc_executor_spin(&executor);
```
Callbacks

In order to receive the goal request response, the client needs to implement a callback:

```c
// Function prototype:
void (* goal_request_callback_t)(rclc_action_goal_handle_t *, bool, void *);

void goal_request_callback(rclc_action_goal_handle_t * goal_handle, bool accepted, void * context)
{
    example_interfaces__action__Fibonacci_SendGoal_Request * request =
        (example_interfaces__action__Fibonacci_SendGoal_Request *) goal_handle->ros_goal_request;
    printf("Goal request (order: %d): %s\n", request->goal.order, accepted ? "Accepted" : "Rejected");
}
```

Also, the client needs to implement a callback to receive feedback:

```c
// Function prototype:
void (* feedback_callback_t)(rclc_action_goal_handle_t *, void *, void *);

void feedback_callback(rclc_action_goal_handle_t * goal_handle, void * ros_feedback, void * context)
{
    example_interfaces__action__Fibonacci_SendGoal_Request * request =
        (example_interfaces__action__Fibonacci_SendGoal_Request *) goal_handle->ros_goal_request;

    example_interfaces__action__Fibonacci_FeedbackMessage * feedback =
        (example_interfaces__action__Fibonacci_FeedbackMessage *) ros_feedback;

    printf("Goal Feedback (order: %d) [", request->goal.order);
    for (size_t i = 0; i < feedback->feedback.sequence.size; i++) {
        printf("%d ", feedback->feedback.sequence.data[i]);
    }
    printf("]\n");
}
```

And finally, the client needs to implement a callback to receive the result:

```c
// Function prototype:
void (* result_request_callback_t)(rclc_action_goal_handle_t *, void *, void *);

void result_request_callback()
{
}
```
rclc_action_goal_handle_t * goal_handle,
void * ros_result_response,
void * context)
{
    (void) context;

    example_interfaces__action__Fibonacci_SendGoal_Request * request =
    (example_interfaces__action__Fibonacci_SendGoal_Request *) goal_handle->ros_goal_˓
    →request;

    example_interfaces__action__Fibonacci_GetResult_Response * result =
    (example_interfaces__action__Fibonacci_GetResult_Response *) ros_result_response;

    printf(
        "Goal Result (order: %d) [ ",
        request->goal.order);

    if (result->status == GOAL_STATE_SUCCEEDED) {
        for (size_t i = 0; i < result->result.sequence.size; i++) {
            printf("%d ", result->result.sequence.data[i]);
        }
    } else if (result->status == GOAL_STATE_CANCELED) {
        printf("CANCELED ");
    } else {
        printf("ABORTED ");
    }

    printf("]\\n");
}

During the goal execution, the action client can request a goal cancellation:

// Send goal cancel request
rclc_action_send_cancel_request(goal_handle);

And define a callback for the cancel request response:

// Function prototype:
void (* cancel_request_callback_t)(rclc_action_goal_handle_t *, bool, void *);

void cancel_request_callback(
    rclc_action_goal_handle_t * goal_handle,
    bool cancelled,
    void * context)
{
    example_interfaces__action__Fibonacci_SendGoal_Request * request =
    (example_interfaces__action__Fibonacci_SendGoal_Request *) goal_handle->ros_goal_˓
    →request;

    printf(
        "Goal cancel request (order: %d): %s\\n",}
Finally, all the callbacks shall be exposed to the executor when the action client is added:

```c
#define MAX_FIBONACCI_ORDER 50

// Action message objects
example_interfaces__action__Fibonacci_FeedbackMessage ros_feedback;
example_interfaces__action__Fibonacci_GetResult_Response ros_result_response;

// Init message memory with expected sizes
ros_feedback.feedback.sequence.capacity = MAX_FIBONACCI_ORDER;
ros_feedback.feedback.sequence.size = 0;
ros_feedback.feedback.sequence.data = (int32_t *) malloc(
    ros_feedback.feedback.sequence.capacity * sizeof(int32_t));

ros_result_response.result.sequence.capacity = MAX_FIBONACCI_ORDER;
ros_result_response.result.sequence.size = 0;
ros_result_response.result.sequence.data = (int32_t *) malloc(
    ros_result_response.result.sequence.capacity * sizeof(int32_t));

// Add action client to the executor ('executor' and 'action_client' are available and already initialized)
rc1c_executor_add_action_client(    &executor,
    &action_client,
    NUMBER_OF_SIMULTANEOUS_HANDLES,
    &ros_result_response,
    &ros_feedback,
    goal_request_callback,
    feedback_callback,
    result_request_callback,
    cancel_request_callback,
    NULL    );

// Spin executor to handle client
rc1c_executor_spin(&executor);
```

**Note:** An action client cannot be removed from the executor.
**Message initialization**

Before using any of the types involved in goal request, cancel or feedback, it may be necessary to initialize its memory for types with strings or sequences.

*Note:* Check the *Handling messages memory tutorial* for details.

**Cleaning Up**

To destroy an initialized action service or client:

```c
// Destroy service server and client
rclc_action_server_fini(&action_server, &node);
rclc_action_client_fini(&action_client, &node);
```

This will delete any automatically created infrastructure on the agent (if possible) and free used memory on the client side.

**Parameter Server**

ROS 2 parameters allow the user to create variables on a node and manipulate/read them with different ROS 2 commands. Further information about ROS 2 parameters can be found on *Understanding parameters*.

Ready to use code related to this concepts can be found on micro-ROS demos repository parameter_server example.
Initialization

A micro-ROS parameter server can be initiated using the RCLC parameter server API:

```c
// Parameter server object
rclc_parameter_server_t param_server;

// Initialize parameter server with default configuration
rcl_ret_t rc = rclc_parameter_server_init_default(&param_server, &node);
if (RCL_RET_OK != rc)
{
    ... // Handle error
    return -1;
}
```

Options

A parameter server can be configured at configuration time, the following options can be adjusted:

- `notify_changed_over_dds`: Publish parameter events to other ROS 2 nodes as well.
- `max_params`: Maximum number of parameters allowed on the `rclc_parameter_server_t` object.
- `allow_undeclared_parameters`: Allows creation of parameters from external parameter clients. A new parameter will be created if a `set` operation is requested on a non-existing parameter.
- `low_mem_mode`: Reduces the memory used by the parameter server, functionality constrains are applied.

```c
// Parameter server object
rclc_parameter_server_t param_server;

// Initialze parameter server options
const rclc_parameter_options_t options = {
    .notify_changed_over_dds = true,
    .max_params = 4,
    .allow_undeclared_parameters = true,
    .low_mem_mode = false; 
};

// Initialize parameter server with configured options
rcl_ret_t rc = rclc_parameter_server_init_with_option(&param_server, &node, &options);
if (RCL_RET_OK != rc)
{
    ... // Handle error
    return -1;
}
```
Low memory mode

There is a low memory mode that ports the parameter functionality to memory constrained devices. The following constrains are applied:

- Request size limited to one parameter on Set, Get, Get types and Describe operations.
- List parameter request has no prefixes enabled nor depth.
- Parameter description strings not allowed, `rclc_add_parameter_description` is disabled.

Note: Using low memory mode in a STM32F4 with 7 parameters with `RCLC_PARAMETER_MAX_STRING_LENGTH = 50` and `notify_changed_over_dds = true` the memory usage drops from 11.7 kB to 4.1 kB.

Callback

When adding the parameter server to the executor, a callback could to be configured. This callback would then be executed on the following events:

- Parameter value change: Internal and external parameter set on existing parameters.
- Parameter creation: External parameter set on unexisting parameter if `allow_undeclared_parameters` is set.
- Parameter delete: External parameter delete on existing parameter.
- The user can allow or reject this operations using the bool return value.

Callback parameters:

- `old_param`: Parameter actual value, NULL for new parameter request.
- `new_param`: Parameter new value, NULL for parameter removal request.
- `context`: User context, configured on `rclc_executor_add_parameter_server_with_context`.

```c
bool on_parameter_changed(
    const Parameter* old_param,
    const Parameter* new_param,
    void* context)
{
    (void) context;

    if (old_param == NULL && new_param == NULL)
    {
        printf("Callback error, both parameters are NULL\n");
        return false;
    }

    if (old_param == NULL)
    {
        printf("Creating new parameter %s\n", new_param->name.data);
    }
    else if (new_param == NULL)
    {
        printf("Deleting parameter %s\n", old_param->name.data);
    }

    (continues on next page)
```
else
{
    printf("Parameter %s modified.", old_param->name.data);
    switch (old_param->value.type){
    case RCLC_PARAMETER_BOOL:
        printf(" Old value: %d, New value: %d (bool)", old_param->value.bool_value,
            new_param->value.bool_value);
        break;
    case RCLC_PARAMETER_INT:
        printf(" Old value: %ld, New value: %ld (int)", old_param->value.integer_value,
            new_param->value.integer_value);
        break;
    case RCLC_PARAMETER_DOUBLE:
        printf(" Old value: %f, New value: %f (double)", old_param->value.double_value,
            new_param->value.double_value);
        break;
    default:
        break;
    }
    printf("\n");
}
    return true;
}

Parameters modifications are disabled while the callback on_parameter_changed is executed, causing the following methods to return RCLC_PARAMETER_DISABLED_ON_CALLBACK if they are invoked:

- rclc_add_parameter
- rclc_delete_parameter
- rclc_parameter_set_bool
- rclc_parameter_set_int
- rclc_parameter_set_double
- rclc_set_parameter_read_only
- rclc_add_parameter_constraint_double
- rclc_add_parameter_constraint_integer

Once the parameter server and the executor are initialized, the parameter server must be added to the executor in order to accept parameter commands from ROS 2:

```c
// Add parameter server to the executor including defined callback
rc = rclc_executor_add_parameter_server(&executor, &param_server, on_parameter_changed);
```

Note that this callback is optional as its just an event information for the user. To use the parameter server without a callback:
// Add parameter server to the executor without a callback
rc = rclc_executor_add_parameter_server(&executor, &param_server, NULL);

Configuration of the callback context:

// Add parameter server to the executor including defined callback and a context
rc = rclc_executor_add_parameter_server_with_context(&executor, &param_server, on_parameter_changed, &context);

Add a parameter

The micro-ROS parameter server supports the following parameter types:

- **Boolean**:

```c
const char* parameter_name = "parameter_bool";
bool param_value = true;

// Add parameter to the server
rcl_ret_t rc = rclc_add_parameter(&param_server, parameter_name, RCLC_PARAMETER_BOOL);

// Set parameter value (Triggers `on_parameter_changed` callback, if defined)
rc = rclc_parameter_set_bool(&param_server, parameter_name, param_value);

// Get parameter value and store it in "param_value"
rc = rclc_parameter_get_bool(&param_server, parameter_name, &param_value);

if (RCL_RET_OK != rc)
{
    ... // Handle error
    return -1;
}
```

- **Integer**:

```c
const char* parameter_name = "parameter_int";
int param_value = 100;

// Add parameter to the server
rcl_ret_t rc = rclc_add_parameter(&param_server, parameter_name, RCLC_PARAMETER_INT);

// Set parameter value
rc = rclc_parameter_set_int(&param_server, parameter_name, param_value);

// Get parameter value on param_value
rc = rclc_parameter_get_int(&param_server, parameter_name, &param_value);
```

- **Double**:
\begin{verbatim}
const char* parameter_name = "parameter_double";
double param_value = 0.15;

// Add parameter to the server
rcl_ret_t rc = rclc_add_parameter(&param_server, parameter_name, RCLC_PARAMETER_DOUBLE);

// Set parameter value
rc = rclc_parameter_set_double(&param_server, parameter_name, param_value);

// Get parameter value on param_value
rc = rclc_parameter_get_double(&param_server, parameter_name, &param_value);
\end{verbatim}

The parameter string name size is controlled by the compile-time option RCLC_PARAMETER_MAX_STRING_LENGTH, the default value is 50.

**Note:** Parameters can also be created by external clients if the allow_undeclared_parameters flag is set. The client just needs to set a value on a non-existing parameter. Then this parameter will be created if the server has still capacity and the callback allows the operation.

---

**Delete a parameter**

Parameters can be deleted by both, the parameter server and external clients:

\begin{verbatim}
rclc_delete_parameter(&param_server, "param2");
\end{verbatim}

For external delete requests, the server callback will be executed, allowing the node to reject the operation.

**Parameters description**

- Parameter description: Adds a description of a parameter and its constraints, which will be returned on a describe parameter request:

\begin{verbatim}
rclc_add_parameter_description(&param_server, "param2", "Second parameter", "Only even numbers");
\end{verbatim}

The maximum string size is controlled by the compilation time option RCLC_PARAMETER_MAX_STRING_LENGTH, default value is 50.

- Parameter constraints: Informative numeric constraints that can be added to int and double parameters, returning these values on describe parameter requests:
  - \textbf{from value}: Start value for valid values, inclusive.
  - \textbf{to value}: End value for valid values, inclusive.
  - \textbf{step}: Size of valid steps between the from and to bound.

\begin{verbatim}
int64_t int_from = 0;
int64_t int_to = 20;
uint64_t int_step = 2;
rclc_add_parameter_constraint_integer(&param_server, "param2", int_from, int_to, int_step);
\end{verbatim}
double double_from = -0.5;
double double_to = 0.5;
double double_step = 0.01;
rclec_add_parameter_constraint_double(&param_server, "param3", double_from, double_to, double_step);

**Note:** This constraint will not be applied by the parameter server, leaving values filtering to the user callback.

- Read-only parameters: This flag blocks parameter changes from external clients, but allows changes on the server side:

  ```
  bool read_only = true;
  rclc_set_parameter_read_only(&param_server, "param3", read_only);
  ```

### Memory requirements

The parameter server uses five services and an optional publisher. These need to be taken into account on the `rmw_microxrcedds` package memory configuration:

**Note:** Check micro-ROS memory management for entity creating tutorial for more information.

```
# colcon.meta example with memory requirements to use a parameter server
{
    "names": {
        "rmw_microxrcedds": {
            "cmake-args": [
                "-DRMW_UXRCE_MAX_NODES=1",
                "-DRMW_UXRCE_MAX_PUBLISHERS=1",
                "-DRMW_UXRCE_MAX_SUBSCRIPTIONS=0",
                "-DRMW_UXRCE_MAX_SERVICES=5",
                "-DRMW_UXRCE_MAX_CLIENTS=0"
            ]
        }
    }
}
```

At runtime, the variable RCLC_EXECUTOR_PARAMETER_SERVER_HANDLES defines the necessary number of handles required by a parameter server for the rclc Executor:

```
// Executor init example with the minimum RCLC executor handles required
rclc_executor_t executor = rclc_executor_get_zero_initialized_executor();
rc = rclc_executor_init(
    &executor, &support.context,
    RCLC_EXECUTOR_PARAMETER_SERVER_HANDLES, &allocator);
```
Cleaning up

To destroy an initialized parameter server:

```c
// Delete parameter server
rclc_parameter_server_fini(&param_server, &node);
```

This will delete any automatically created infrastructure on the agent (if possible) and deallocate used memory on the parameter server side.

Middleware API

Table of Contents

- Transport configuration
- Client Key configuration
- Time configuration
  - Timeouts
  - Time synchronization
- Ping agent
- Error handling

micro-ROS default rmw layer `rmw_microxrcedds_c` offers a set of utilities to directly configure and interact with the underlying middleware. The exposed API detailed on this documentation can be used with the `rmw_microxrcedds_c` public header: `#include <rmw_microros/rmw_microros.h>`.

Transport configuration

Middleware provided transports can be configured at run-time, allowing the user to set the Agent address or the serial port identifier:

```c
// Init RCL options and context
rcl_init_options_t init_options = rcl_get_zero_initialized_init_options();
rcl_context_t context = rcl_get_zero_initialized_context();
rcl_init_options_init(&init_options, rcl_get_default_allocator());

rcl_context_t context = rcl_get_zero_initialized_context();
rcl_init_options_init(&init_options, rcl_get_default_allocator());

// Take RMW options from RCL options
rmw_init_options_t* rmw_options = rcl_init_options_get_rmw_init_options(&init_options);

// UDP transport configuration
char* agent_ip = "127.0.0.1";
char* agent_port = "8888";
rmw_uros_options_set_udp_address(agent_ip, agent_port, rmw_options);

// Serial transport configuration
char* serial_device = "/dev/ttyAMA0";
rmw_uros_options_set_serial_device(serial_device, rmw_options)
```
Note: Transport must be enabled on build time using RMW_UXRCE_TRANSPORT_UDP or RMW_UXRCE_TRANSPORT_SERIAL options.

Client Key configuration

The client key will identify a micro-ROS client on the Agent side, implying that all the connected clients must have a different key. This value is set randomly on micro-ROS initialization, but it is also possible to set it manually:

```c
// Init RCL options and context
rcl_init_options_t init_options = rcl_get_zero_initialized_init_options();
rcl_context_t context = rcl_get_zero_initialized_context();
rcl_init_options_init(&init_options, rcl_get_default_allocator());

// Take RMW options from RCL options
rmw_init_options_t* rmw_options = rcl_init_options_get_rmw_init_options(&init_options);

// Set RMW client key
uint32_t client_key = 0xBA5EBA11;
rmw_uros_options_set_client_key(0xBA5EBA11, rmw_options);
```

This feature can be useful for reusing DDS entities already created on the agent side, as explained on this Micro XRCE-DDS Deployment example.

Time configuration

Timeouts

• Confirmation timeout: Reliable entities can increase blocking time on rcl_publish or executor spin calls as they will wait for the acknowledgement of each sent message.

  The default timeout value can be modified individually at run-time:

```c
// Confirmation timeout in milliseconds
int ack_timeout = 1000;

// Get RMW publisher handle and set reliable timeout
rmw_publisher_t* rmw_publisher_handle = rcl_publisher_get_rmw_handle(&publisher);
rmw_uros_set_publisher_session_timeout(&rmw_publisher_handle, ack_timeout);

// Get RMW service handle and set reliable timeout
rmw_service_t* rmw_service_handle = rcl_service_get_rmw_handle(&service);
rmw_uros_set_service_session_timeout(rmw_service_handle, ack_timeout);

// Get RMW service handle and set reliable timeout
rmw_client_t* rmw_client_handle = rcl_client_get_rmw_handle(&client);
rmw_uros_set_client_session_timeout(rmw_client_handle, ack_timeout);
```

• Entity timeouts: Creation and destruction of entities also include a timeout, as they will wait for the Agent confirmation on the operation. This timeout will affect all init and fini methods such as rclc_node_init_default, rcl_publisher_fini, etc.
The default value can be modified at run-time for all entities:

```c
rclc_support_t support;
support.context

rmw_context_t * rmw_context = rcl_context_get_rmw_context(&support.context);

if (NULL != rmw_context)
{
    // Timeout in milliseconds
    int timeout = 1000;
    rmw_uros_set_context_entity_creation_session_timeout(rmw_context, timeout);
    rmw_uros_set_context_entity_destroy_session_timeout(rmw_context, timeout);
}
```

**Note:** To avoid waiting for agent confirmation, `timeout = 0` can be used, allowing the release of local resources if the agent is not present.

---

### Time synchronization

micro-ROS clients can synchronize their epoch time with the connected Agent, this can be very useful when working in embedded environments that do not provide any time synchronization mechanism. This utility is based on the NTP protocol, taking into account delays caused by the transport layer.

An usage example can be found on `epoch_synchronization` micro-ROS demo:

```c
// Sync timeout
const int timeout_ms = 1000;

// Synchronize time with the agent
rmw_uros_sync_session(timeout_ms);
```

After the session is synchronized, the adjusted timestamp can be retrieved with the following API:

```c
// Check if session has been synchronized
if (rmw_uros_epoch_synchronized())
{
    // Get time in milliseconds or nanoseconds
    int64_t time_ms = rmw_uros_epoch_millis();
    int64_t time_ns = rmw_uros_epoch_nanos();
}
```

**Note:** micro-ROS shall be already initialized and connected to the agent to use this functionality.
Ping agent

The client can test the connection with the Agent with the ping utility. This functionality can be used even when the micro-ROS context has not yet been initialized, which is useful to test the connection before trying to connect to the Agent. An example can be found on ping_uros_agent micro-ROS demo.

```c
// Timeout for each attempt
const int timeout_ms = 1000;

// Number of attempts
const uint8_t attempts = 5;

// Ping the agent
rmw_ret_t ping_result = rmw_uros_ping_agent(timeout_ms, attempts);

if (RMW_RET_OK == ping_result)
{
    // micro-ROS Agent is reachable
    ...
}
else
{
    // micro-ROS Agent is not available
    ...
}
```

A secondary API is provided to ping the Agent with a specific rmw configuration. This API allows the user to ping with a specific custom transport without interfering with the actual micro-ROS configuration:

```c
// Initialize rcl options and retrieve the internal rmw options
rcl_init_options_t init_options = rcl_get_zero_initialized_init_options();
rcl_init_options_init(&init_options, allocator);
rmw_init_options_t * rmw_options = rcl_init_options_get_rmw_init_options(&init_options);

// Set custom transport
rmw_uros_options_set_custom_transport(
    false,
    NULL,
    custom_transport_open,
    custom_transport_close,
    custom_transport_write,
    custom_transport_read,
    rmw_options);

// Ping the agent with custom transport
const int timeout_ms = 1000;
const uint8_t attempts = 5;
rmw_ret_t ping_result = rmw_uros_ping_agent_options(timeout_ms, attempts, rmw_options);

if (RMW_RET_OK == ping_result)
{
    // micro-ROS Agent is reachable
    ...
}
```

(continues on next page)
Error handling

micro-ROS RMW can be configured to report middleware errors to user space using custom callbacks. This option is disabled by default and needs to be enabled at compile time via `RMW_UROS_ERROR_HANDLING` CMake argument.

The behavior of this flag is:

- **`RMW_UROS_ERROR_HANDLING=OFF`**: Error handling is disabled. This is the default behavior. **Errors are not reported to user space.**
- **`RMW_UROS_ERROR_HANDLING=ON`** and **callback not set**: Error handling is enabled. Default ROS 2 RMW macros are used to report errors.
- **`RMW_UROS_ERROR_HANDLING=ON`** and **callback set**: Error handling is enabled. User callback and default ROS 2 RMW macros are used to report errors.

An example `colcon.meta` is:

```json
{
    "names": {
        "rmw_microxrceddss": {
            "cmake-args": [
                "-DRMW_UROS_ERROR_HANDLING=ON"
            ]
        }
    }
}
```

Once enabled, the user can register a callback to be called when an error is detected, using the following API:

```c
void rmw_uros_set_error_handling_callback(
    rmw_uros_error_handling error_cb);
```

An example callback of type `rmw_uros_error_handling` is:

```c
void error_handler(
    const rmw_uros_error_entity_type_t entity,
    const rmw_uros_error_source_t source,
    const rmw_uros_error_context_t context,
    const char * file,
    const int line)
{
    // Do something with the error
    ...
}
```

`rmw_uros_error_entity_type_t` represent with entity is triggering the error. It can be one of the following:
- `RMW_UROS_ERROR_ON_UNKNOWN`: Generic entity.
- `RMW_UROS_ERROR_ON_NODE`: Node entity.
- `RMW_UROS_ERROR_ON_SERVICE`: Service server entity.
- `RMW_UROS_ERROR_ON_CLIENT`: Service client entity.
- `RMW_UROS_ERROR_ON_SUBSCRIPTION`: Subscription entity.
- `RMW_UROS_ERROR_ON_PUBLISHER`: Publisher entity.
- `RMW_UROS_ERROR_ON_GRAPH`: Graph manager.
- `RMW_UROS_ERROR_ON_GUARD_CONDITION`: Guard condition entity.
- `RMW_UROS_ERROR_ON_TOPIC`: Topic memory.

`rmw_uros_error_source_t` represent the source of the error. It can be one of the following:

- `RMW_UROS_ERROR_ENTITY_CREATION`: Error on entity creation.
- `RMW_UROS_ERROR_ENTITY_DESTRUCTION`: Error on entity destruction.
- `RMW_UROS_ERROR_CHECK`: Error on a check.
- `RMW_UROS_ERROR_NOT_IMPLEMENTED`: Feature not implemented.
- `RMW_UROS_ERROR_MIDDLEWARE_ALLOCATION`: Memory error.

`rmw_uros_error_context_t` represent the context of the error and contains, one or more of the following members:

- `node`: Name of the node of type `const char *`.
- `node_namespace`: Namespace of the node of type `const char *`.
- `topic_name`: Name of the topic of type `const char *`.
- `ucdr`: Pointer to the `ucdrBuffer` of type `const ucdrBuffer *`.
- `size`: Size of the buffer of type `size_t`.
- `type_support`: Pointer to the type support of type `const message_type_support_callbacks_t *` or `const service_type_support_callbacks_t *`.
- `description`: Description of the error of type `const char *`.

Also, this callback function gets the file name and line number where the error was detected. This information can be accessed using:

- `file`: File name of type `const char *`.
- `line`: Line number of type `int`.

### micro-ROS Utilities

<table>
<thead>
<tr>
<th>Table of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>micro-ROS String Utilities</em></td>
</tr>
<tr>
<td><em>micro-ROS Types Utilities</em></td>
</tr>
</tbody>
</table>

`micro_ros_utilities` is a package that provides a set of tools for easing the use of micro-ROS on different platforms. Currently, it has two APIs:
micro-ROS type utils.

- micro-ROS `rosidl_runtime_c__String` wrapper that reduces dynamic memory operations.

**Note:** A full example can be found on how to use this API can be found [here](#).

### micro-ROS String Utilities

This API helps developers to manage strings in micro-ROS by means of providing a set of methods that allow initialization, destruction, set, and other common operations.

**Warning:** It is important to note that, unlike in ROS 2 rclcpp, in micro-ROS rclc memory in types is not initialized by default. It is required that the user initializes the memory for each type that is used in any micro-ROS procedure.

#### micro_ros_string_utilities_init

Create a `rosidl_runtime_c__String` from a `char` pointer.

```c
const char * str = "Hello World";
rosidl_runtime_c__String ros_str = micro_ros_string_utilities_init(str);
```

<table>
<thead>
<tr>
<th>Operation</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocate</td>
<td>Yes, it does not take ownership of <code>char*</code></td>
</tr>
<tr>
<td>Reallocate</td>
<td>No</td>
</tr>
<tr>
<td>Free</td>
<td>No</td>
</tr>
</tbody>
</table>

#### micro_ros_string_utilities_init_with_size

Create a `rosidl_runtime_c__String` from a size.

```c
size_t size = 10;
rosidl_runtime_c__String ros_str = micro_ros_string_utilities_init_with_size(size);
```

<table>
<thead>
<tr>
<th>Operation</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocate</td>
<td>Yes</td>
</tr>
<tr>
<td>Reallocate</td>
<td>No</td>
</tr>
<tr>
<td>Free</td>
<td>No</td>
</tr>
</tbody>
</table>
**micro_ros_string_utilities_set**

Create a `rosidl_runtime_c__String` from a char pointer reallocating an actual `rosidl_runtime_c__String`.

```c
const char * str = "Hello World";
size_t size = strlen(str) + 1;  // Add null terminator
rosidl_runtime_c__String ros_str = micro_ros_string_utilities_init_with_size(size);
ros_str = micro_ros_string_utilities_set(ros_str, str);
```

<table>
<thead>
<tr>
<th>Operation</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocate</td>
<td>No</td>
</tr>
<tr>
<td>Reallocate</td>
<td>Reallocates input <code>rosidl_runtime_c__String</code>, does not take ownership of char*</td>
</tr>
<tr>
<td>Free</td>
<td>No</td>
</tr>
</tbody>
</table>

**micro_ros_string_utilities_get_c_str**

Returns the char pointer to the `rosidl_runtime_c__String` data.

```c
const char * str = "Hello World";
rosidl_runtime_c__String ros_str = micro_ros_string_utilities_init(str);
char * ptr_str = micro_ros_string_utilities_get_c_str(ros_str);
```

**micro_ros_string_utilities_append**

Appends a char pointer to the end of a `rosidl_runtime_c__String`.

```c
const char * str = "Hello World";
rosidl_runtime_c__String ros_str = micro_ros_string_utilities_init(str);
ros_str = micro_ros_string_utilities_append(ros_str, "!");
```

<table>
<thead>
<tr>
<th>Operation</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocate</td>
<td>No</td>
</tr>
<tr>
<td>Reallocate</td>
<td>Yes if input <code>rosidl_runtime_c__String</code> is not big enough</td>
</tr>
<tr>
<td>Free</td>
<td>No</td>
</tr>
</tbody>
</table>

**micro_ros_string_utilities_remove_tail_chars**

Removes characters from the end of a `rosidl_runtime_c__String`.

```c
const char * str = "Hello World";
rosidl_runtime_c__String ros_str = micro_ros_string_utilities_init(str);
ros_str = micro_ros_string_utilities_remove_tail_chars(ros_str, 5);
```
<table>
<thead>
<tr>
<th>Operation</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocate</td>
<td>No</td>
</tr>
<tr>
<td>Reallocate</td>
<td>No</td>
</tr>
<tr>
<td>Free</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**micro_ros_string_utilities_destroy**

Destroys a rosidl_runtime_c__String.

```c
const char * str = "Hello World";
rosidl_runtime_c__String ros_str = micro_ros_string_utilities_init(str);
micro_ros_string_utilities_destroy(ros_str);
```

**micro-ROS Types Utilities**

This API helps developers to manage ROS types in micro-ROS. It handles the types structures recursively in order to initialize each member with the required memory size.

**micro_ros_utilities_memory_conf_t**

`micro_ros_utilities` provides a functionality to instantiate sequences and strings of fixed sizes.

Memory can be allocated in two ways:

- *statically*: in an user provided buffer.
- *dynamically*: using ROS 2 allocators

Memory allocation can be configured using configuration structure that has the following members:

- **max_string_capacity**: Maximum string capacity to use for message fields.
- **max_ros2_type_sequence_capacity**: Maximum capacity to use for sequence type msg fields (ie: unbounded arrays and lists) which contain ROS 2 msg types.
- **max_basic_type_sequence_capacity**: Maximum capacity to use for sequence type msg fields (ie: unbounded arrays and lists) which contain basic types (ie: primitive field types).

```c
static micro_ros_utilities_memory_conf_t conf = {0};
// OPTIONALLY this struct can configure the default size of strings, basic sequences and composed sequences
conf.max_string_capacity = 50;
conf.max_ros2_type_sequence_capacity = 5;
conf.max_basic_type_sequence_capacity = 5;
```

All message members will follow this configuration, unless they have a custom rule assigned to them.
**micro_ros_utilities_type_info**

Returns a `rosidl_runtime_c__String` with the type introspection data.

```c
#include <control_msgs/msg/joint_jog.h>

rosidl_runtime_c__String ros_str = micro_ros_utilities_type_info(ROSIDL_GET_MSG_TYPE_SUPPORT(control_msgs, msg, JointJog));
```

**micro_ros_utilities_get_static_size**

Returns the static memory size that will be used for a type with a given memory configuration.

```c
#include <control_msgs/msg/joint_jog.h>

static micro_ros_utilities_memory_conf_t conf = {0};

// OPTIONALLY this struct can configure the default size of strings, basic sequences and composed sequences
conf.max_string_capacity = 50;
conf.max_ros2_type_sequence_capacity = 5;
conf.max_basic_type_sequence_capacity = 5;

control_msgs__msg__JointJog msg;
rosidl_runtime_c__String ros_str = micro_ros_utilities_get_static_size(ROSIDL_GET_MSG_TYPE_SUPPORT(control_msgs, msg, JointJog) conf);
```

**micro_ros_utilities_create_message_memory**

Allocates dynamic memory for a message.

```c
#include <control_msgs/msg/joint_jog.h>

static micro_ros_utilities_memory_conf_t conf = {0};

// OPTIONALLY this struct can configure the default size of strings, basic sequences and composed sequences
conf.max_string_capacity = 50;
conf.max_ros2_type_sequence_capacity = 5;
conf.max_basic_type_sequence_capacity = 5;

control_msgs__msg__JointJog msg;
bool success = micro_ros_utilities_create_message_memory(ROSIDL_GET_MSG_TYPE_SUPPORT(control_msgs, msg, JointJog), &msg, conf);
```
**micro_ros_utilities_create_static_message_memory**

Allocates memory for a message in a user-provided buffer.

```c
#include <control_msgs/msg/joint_jog.h>

uint8_t my_buffer[1000];
static micro_ros_utilities_memory_conf_t conf = {0};

// OPTIONALLY this struct can configure the default size of strings, basic sequences and composed sequences
conf.max_string_capacity = 50;
conf.max_ros2_type_sequence_capacity = 5;
conf.max_basic_type_sequence_capacity = 5;

control_msgs__msg__JointJog msg;
bool success = micro_ros_utilities_create_static_message_memory(
    ROSIDL_GET_MSG_TYPE_SUPPORT(control_msgs, msg, JointJog),
    &msg_static,
    conf,
    my_buffer,
    sizeof(my_buffer)
);
```

**micro_ros_utilities_destroy_message_memory**

Deallocates the dynamic memory of a message.

```c
#include <control_msgs/msg/joint_jog.h>

static micro_ros_utilities_memory_conf_t conf = {0};

// OPTIONALLY this struct can configure the default size of strings, basic sequences and composed sequences
conf.max_string_capacity = 50;
conf.max_ros2_type_sequence_capacity = 5;
conf.max_basic_type_sequence_capacity = 5;

control_msgs__msg__JointJog msg;
bool success = micro_ros_utilities_create_message_memory(
    ROSIDL_GET_MSG_TYPE_SUPPORT(control_msgs, msg, JointJog),
    &msg,
    conf
);

success &= micro_ros_utilities_destroy_message_memory(
    ROSIDL_GET_MSG_TYPE_SUPPORT(control_msgs, msg, JointJog),
    &msg,
    conf
);
```
QoS configuration

Table of Contents

- Initialization
- History behaviour

QoS (Quality of Services) configuration related to DDS communication can also applied to micro-ROS entities, as its default middleware eProsima Micro XRCE-DDS allows for complete customization using the provided RMW API. Further information about ROS 2 available QoS can be found on About Quality of Service settings.

Initialization

As shown in the initialization API of each entity, a rmw_qos_profile_t can be used to configure the QoS of a micro-ROS entity:

```c
// Create a publisher with depth 100.
rcl_publisher_t custom_publisher;
rmw_qos_profile_t custom_qos = rmw_qos_profile_default;
custom_qos.depth = 100;
rclc_publisher_init(
    &custom_publisher, &node,
    &type_support, &topic_name, &custom_qos);
```

This rmw_qos_profile_t structure is provided by ROS 2 RMW headers: rmw_qos_profile_t. A list with predefined qos profiles is also available: qos_profiles.h.

Note: By default, the following qos configuration is applied on each entity:

- Default Publisher/subscriber: rmw_qos_profile_default.
- Best effort Publisher/subscriber: rmw_qos_profile_sensor_data.
- Default Service/client: rmw_qos_profile_services_default.
- Best effort Service/client: rmw_qos_profile_services_default with reliability = RMW_QOS_POLICY_RELIABILITY_BEST_EFFORT

History behaviour

When micro-ROS history slots are complete and new data arrives, the behaviour is determined by the configured history kind and depth. More details about memory configuration can be found on the Middleware related memory tutorial.

- RMW_QOS_POLICY_HISTORY_KEEP_LAST: New data will be stored on free history slots up to the configured depth value. If the entity already owns up to depth slots, a the oldest message will be freed and use for the received data.
- RMW_QOS_POLICY_HISTORY_KEEP_ALL: New data will always be stored on free history slots up to the configured depth value. If the entity already owns up to depth slots, the new message will be discarded.
Vulcanexus Documentation, Release 1.0.0

- There is a special case for depth = 0, where the history kind will be ignored and history slots won't be reused before they are released, following the keep all approach.

Note: History kind types not listed on this section are treated as RMW_QOS_POLICY_HISTORY_KEEP_LAST.

Multithreading

Table of Contents

- Configuration
- API usage

Multithreading support can be enabled at compilation time, which enables the micro-ROS default middleware (Micro XRCE-DDS) to be thread safe under specific circumstances.

With this feature, micro-ROS entities such as publishers, subscribers or services can run on different RTOS execution threads, with the improvements on performance, usability and resource optimization that this implies.

Ready to use code related to this concept can be found on micro-ROS demos repository multi-thread_publisher_subscriber example.

Configuration

This option is disabled by default and needs to be enabled at compile time via UCLIENT_PROFILE_MULTITHREAD CMake argument on the middleware package (microxrcedds_client).

An example colcon.meta is:

```json
{
   "names": {
      "microxrcedds_client": {
         "cmake-args": ["-DUCLIENT_PROFILE_MULTITHREAD=ON"
         ]
      }
   }
}
```

Supported platforms

This functionality is not available on all platforms, as specific Mutex implementations are provided by each system. Support for the following platforms is provided:

- FreeRTOS
- Zephyr
- POSIX

For other platforms, a compilation error will be triggered with the following message: XRCE multithreading not supported for this platform.
In order to allow the micro-ROS library to be built against the locking APIs of these RTOSes, the required include folders shall be added in the *CMake Toolchain* used to build the library.

If support for other environment is required, a contribution to the *multithread* support in Micro XRCE-DDS client shall be done.

**API usage**

This section will explain the limitations of this approach, offering design tips to overcome them.

**Restrictions**

1. **Entity creation and destruction methods are not thread safe**: Any other micro-ROS related execution shall be halted until this steps are completed.

2. The *parameter server API* is not thread safe, this include modifications coming from a triggered callback.

3. The executor callbacks will run within the same thread where the executor is spinned.

**Architecture tips**

1. Initialization and destruction of entities:
   - Have a initial micro-ROS thread with initialization and destruction steps for all entities.
   - Use the approach shown on the *Ping API* tutorial to block other threads until the micro-ROS entities are ready.

2. Callback dispatching:
   - Executor callbacks can be distributed on multiple threads by using a unique executor instance per thread.
   - This means that its expected to have a executor instance for each thread where callbacks shall be processed.

3. Publishers’ publish methods are thread safe and can be called from multiple threads as long as the publisher object is not destroyed.

4. Parameter server local API shall be used within the same thread as its related executor is spinned.

5. Action servers goal execution is expected to run on its own thread, being *rclc_action_send_result* thread safe.

**4.9.4 Benchmarking**

- **Memory Profiling**
  - Profiling methodology
  - Pub/Sub applications
  - Service/Client applications
  - RMW History
- **Throughput**
  - Stream-oriented transports
This tutorial will give insight on micro-ROS benchmarking on different topics:

- Memory profiling: Static, dynamic and stack.
- Data throughput on different transports.
- Latency between micro-ROS Client and Agent.

This tests has been performed on a Renesas EK-RA6M5 board using micro-ROS for Renesas e2 Studio build system.

### Memory Profiling

This section will cover micro-ROS memory usage on the most basic entity types. The test on this section have been performed using the provided UDP transport with FreeRTOS + TCP as network stack.

#### Profiling methodology

The memory profile has been performed with the following configuration:

- Reliable entities with a fixed topic size.
- UDP transport (FreeRTOS + TCP).
- Transport MTU: 512 B.
- Micro XRCE-DDS Client history: 4 slots.
- RMW History: 4 slots (Except for RMW History section).

For more information on the middleware configuration, check the Memory management tutorial.

**Note:** There are no differences on memory usage between different topic sizes and the reliability kind used, as the topic size plus reliability and/or middleware overhead shall fit in the static buffers pre-allocated by the program at compile-time, defined by the history configuration.

In general, the topic size will only affect data throughput as it is directly related to the size of the messages exchanged by the middleware.

Meanwhile, to measure the different types of memory:

- **Static memory:** The static memory has been calculated as the difference between the memory occupied by the `.bss` and `.data` sections with a non-zero number of entities, and the memory occupied by the same sections when no micro-ROS application is running, that is, the memory occupied by the rest of components of the RTOS and libraries.

- **Stack memory:** The stack consumed during the program execution is taken into account by means of a FreeRTOS specific function involved in the memory management capabilities offered by this RTOS, the `uxTaskGetStackHighWaterMark()` function. This function returns the amount of stack that remains unused when the stack consumed by the program is at its greatest value. By subtracting this figure to the total stack available, which is known, one can obtain the stack peak used by the app.

- **Dynamic Memory:** This is the memory dynamically allocated by the program by calls to `calloc()` and `malloc()` functions in the C language. The call to dynamic memory have been override with custom memory allocators to measure the total requested memory.
Pub/Sub applications

Publishers and subscribers have been tested varying the `RMW_UXRCE_MAX_SUBSCRIPTIONS` and `DRMW_UXRCE_MAX_PUBLISHERS` configuration between 1, 5 and 10. The entities are then initialized and used as usual on a `std_msgs/msg/Int32` topic.

**Note**: Notice that each of these entities has its own associated topic, concluding that the number of topics used does not impact memory usage.

The total memory (static plus stack plus dynamic) occupied is summarized in the plots below:

![Graph showing memory usage for publishers and subscribers](image)

From this data, it’s concluded that a publisher takes a total of ~550 B meanwhile a subscriber uses ~600 B. There is virtually no difference between these two entities, as the memory pools of micro-ROS RMW are shared among all the entities participating in a given application.

To get a better understanding of the memory usage, the same is provided data but broken down into its different memory types used:

This shows that both the static and the dynamic memories change with the entity number, while the stack usage stays constant.

Service/Client applications

The same approach is used to measure service and clients applications for a `example_interfaces/srv/AddTwoInts` service kind. Notice that this time the total memory is shown along its individual types:

As concluded on the previous section, the memory used is almost identical for a ~500 B usage by both entity kinds. Note that it is also virtually identical to the memory used by a publisher or subscriber application.
Chapter 4. Contributing to the documentation
RMW History

As explained before, the topic memory comes from the RMW history, which is formed by static memory pools defined on compilation time. For a varying `RMW_UXRCE_MAX_HISTORY` between 1 and 10:

![Graph showing static memory usage vs RMW history](image)

As expected, the static memory used by each history slot equals the `MTU * RMW_UXRCE_STREAM_HISTORY` formula, which for this scenario: $512 \times 4 = 2048$ B. For more details on the middleware memory usage, check the Memory management tutorial.

Throughput

On this section data throughput will be measured for different transports and topic sizes. To perform this test, a simple best effort publisher micro-ROS application sends variable `std_msgs/msg/String` for 5 seconds.

The transport are divided based on their framing configuration. More details can be found on the Custom Transports tutorial.
Stream-oriented transports

The tested stream oriented transports and their configuration are:

- USB-CDC: 115200 bauds per second.
- Serial UART: 115200 bauds per second.
- TCP (AWS Secure Sockets) based on Wi-Fi-Pmod-Expansion-Board.
  - PMOD: 460800 bauds per second.

As expected, USB shows the higher throughput due to the fact that has the higher bandwidth, followed by TCP over WiFi and Serial. There is also a great improvement on the throughput as the payload is increased, caused by the overhead added by the HDLC framing protocol.

Packet-oriented transports

As for packet oriented transports, the following have been tested:

- CAN-FD using a PCAN-USB FD adapter.
  - Nominal rate: 0.5 Mbps
  - Data rate: 2 Mbps
- UDP (FreeRTOS + TCP) over cable.
- UDP (ThreadX + NetX) over cable.

This data shows how variable is micro-ROS data throughput regarding the used RTOS and network stack, as there is a clear difference between UDP using FreeRTOS + TCP and NetX. Its also clear that the throughput in this case is linear with the topic size, avoiding performance differences as in the previous section.
**Note:** As CAN-FD protocol has a maximum payload of 64 bytes, the topic size used has been adjusted to the available RMW History parameter.

### Latency

Latency and round trip time (RTT) has been measured with a pub/sub application were timestamps are exchanged using `std_msgs/msg/Int64` messages.

To calculate the results, the timestamp of the board is synchronized with the Agent using the `time synchronization API`.

<table>
<thead>
<tr>
<th>Transport</th>
<th>Client publish time</th>
<th>Agent publish time</th>
<th>RTT</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN-FD</td>
<td>1.29 ms</td>
<td>2.41 ms</td>
<td>3.70 ms</td>
</tr>
<tr>
<td>USB-CDC</td>
<td>1.09 ms</td>
<td>1.45 ms</td>
<td>2.55 ms</td>
</tr>
<tr>
<td>Serial UART</td>
<td>3.77 ms</td>
<td>5.04 ms</td>
<td>8.81 ms</td>
</tr>
<tr>
<td>UDP (ThreadX + NetX)</td>
<td>0.54 ms</td>
<td>0.72 ms</td>
<td>1.26 ms</td>
</tr>
<tr>
<td>UDP (FreeRTOS + TCP)</td>
<td>0.52 ms</td>
<td>0.70 ms</td>
<td>1.22 ms</td>
</tr>
<tr>
<td>TCP (PMOD WiFi)</td>
<td>1.55 ms</td>
<td>2.04 ms</td>
<td>3.59 ms</td>
</tr>
</tbody>
</table>

As expected, the latency and RTT is directly related to the transport latency and throughput.

Tutorials about micro-ROS can be found at [micro-ROS tutorials](#).

micro-ROS follows an architecture that is based on ROS 2, but also has significant differences that allows the embedded integration. In order to ease the use, it also provides tools for handling the integration in embedded platforms.

**micro-ROS layered architecture** abstracts the OS/RTOS with a middleware layer. On top of the middleware, micro-ROS uses the same ROS 2 core layers (RMW, RCL and RCUTILS). With respect to user APIs, micro-ROS provides a **C99 API named RCLC** that allows using all the concepts available on ROS 2 C++ or Python APIs (RCLCPP or RCLPY).

**Compatibility with ROS 2 type support** is also granted, meaning that any message definition used in ROS 2 can be used seamlessly in micro-ROS. By means of the micro-ROS build systems, any `.msg`, `.srv` or `.action` definition file can be integrated in the micro-ROS build in order to achieve a type interoperability with ROS 2.

In the case of micro-ROS, the default middleware layer is eProsima **Micro XRCE-DDS**, a **lightweight, portable**, with **minimal overhead** middleware based on the OMG DDS-XRCE standard.

The micro-ROS default middleware is compound of two parts: the **Micro XRCE-DDS Client** that runs in the MCU side, and the **Micro XRCE-DDS Agent** that runs in the ROS 2 side and communicates the embedded side with the ROS 2 dataspace. The eProsima **Micro XRCE-DDS** is the base package for the **micro-ROS Agent**, a package that extends the former with specific ROS 2 features such as graph management.

**Note:** **micro-ROS Agent** is distributed along with **Vulcanexus Micro**. Check the installation tutorial.

In the same way as ROS 2, micro-ROS has a **pluggable middleware interface** so other middleware implementations can be used at the bottom layer of the stack.

micro-ROS provides a set of tools for integrating it in many build systems, toolchains and embedded development frameworks. Those packages are heterogeneous due to the diversity of tools used by different vendors.

In general, micro-ROS provides standalone solutions for most used RTOSes and environments such as: **Zephyr RTOS**, **Espressif ESP-IDF**, **FreeRTOS**, **Microsoft Azure RTOS**, etc. It also provides packages for integrating it in major vendor tools: **Renesas e2 Studio** or **STM32 Cube IDE/MX**.
Finally micro_ros_setup is a package that provides simple scripts for navigating the micro-ROS support packages. micro_ros_setup is distributed along with Vulcanexus Micro. Check the installation tutorial.

Note: For the full understanding of this build system approach please refer to the Build System Components reference.

4.10 Vulcanexus Core Tutorials

A set of ROS 2 tutorials are provided in the ROS 2 Documentation. Vulcanexus documentation includes additional ROS 2 tutorials, along with middleware specific user features only available in Fast DDS.

This section will provide a collection of tutorials both on the use and application of the basic functionality, as well as on the exploitation of ROS 2 for more advanced users.

4.10.1 Vulcanexus Security Tutorials

Vulcanexus ensures maximum security for ROS 2 networks. Security in ROS 2 is provided by the underlying DDS implementation; with Fast DDS being the most secure open-source DDS implementation, fulfilling four of the five DDS Security standard plugins. Furthermore, Fast DDS meets Quality Level 1 requirements by ROS and offers PKCS#11 support to store and retrieve private keys from a hardware secure module (HSM).

Listed below are the tutorials for configuring security in ROS 2 environments.
Storing private keys in Hardware Secure Modules (HSM)

- Background
- Prerequisites
- ROS 2 Security infrastructure
- Generate HSM token
- Execute ROS 2 demo nodes with security
- Security breach tryout

Background

ROS 2 Security is implemented by means of the DDS Security specification. Specifically, when using Vulcanexus, security is implemented through Fast DDS security plugins. The security files required by the aforementioned specification are encapsulated within the ROS 2 security enclaves. This approach allows to set different security policies for different nodes, processes, users, or devices by defining different enclaves with their corresponding permissions. However, ROS 2 currently requires that the private key used to encrypt and decrypt the enclave information is also stored within the enclave as plain text, which might pose a security vulnerability. Such vulnerability can be mitigated through the use of Hardware Security Modules (HSM). These modules provide a query API compliant with the PKCS #11 standard for which Vulcanexus Core (through Fast DDS) provides the required support so the private keys, instead of being stored in the filesystem, could be kept inside HSM devices.

This tutorial provides step-by-step instructions to secure the communications within the ROS 2 talker/listener demo by the means of a HSM and leveraging Vulcanexus Core enhanced security capabilities.

Prerequisites

This tutorial does not require a physical HSM device connected to the test environment. Instead, a software HSM emulator is going to be installed and used. Run the following commands to install the required packages (administrative privileges may be required):

```
apt update
apt install --yes --no-install-recommends softhsm2 gnutls-bin
```

Note that `softhsm2` package creates a new group called `softhsm`. In order to grant access to the HSM module, non-root users must belong to this group (root users already have permission). To add a user to the group, run the following command (administrative privileges may be required):

```
usermod -a -G softhsm <user>
```

For this command to take effect, the user should log out and in again into its account.

Please, also remember to source the environment in every terminal in this tutorial.

```
source /opt/vulcanexus/iron/setup.bash
```
ROS 2 Security infrastructure

First of all, the required security enclaves to run this tutorial has to be generated using the ROS 2 CLI. Two security enclaves are going to be generated in this tutorial, one for the talker node and the other for the listener. ROS 2 enclaves are saved within a keystore generated running the following commands:

```
mkdir ~/vulcanexus_hsm_ws && cd ~/vulcanexus_hsm_ws
ros2 security create_keystore vulcanexus_demo_keystore
ros2 security create_enclave vulcanexus_demo_keystore /talker_listener/talker
ros2 security create_enclave vulcanexus_demo_keystore /talker_listener/listener
```

For more information about ROS 2 security infrastructure, please refer to this tutorial.

Generate HSM token

In this step, the private keys that have been generated by ROS 2 CLI in each enclave are going to be imported into virtual HSM tokens for later retrieval by Vulcanexus when necessary using PKCS #11. Therefore, the key.pem auto-generated files within each enclave are going to be replaced by a key.p11 file with the PKCS #11 URI in it. As already explained, this tutorial emulates the HSM using softHSM. Generate the HSM tokens running (administrative privileges may be required):

```
softsm2-util --init-token --free --label vulcanexus_demo_talker --pin...
  -VulcanexusDemoTalker --so-pin VulcanexusDemoTalker
softsm2-util --init-token --free --label vulcanexus_demo_listener --pin...
  -VulcanexusDemoListener --so-pin VulcanexusDemoListener
```

As the private keys have already been generated by ROS 2 CLI, this demo is going to directly import these keys into the HSM tokens (administrative privileges may be required).

```
softsm2-util --import vulcanexus_demo_keystore/enclaves/talker_listener/talker/key.pem...
  --token vulcanexus_demo_talker --label key --pin VulcanexusDemoTalker --id...
  123456789ABCDEF123456789ABCDEF
softsm2-util --import vulcanexus_demo_keystore/enclaves/talker_listener/listener/key.pem...
  --token vulcanexus_demo_listener --label key --pin VulcanexusDemoListener --id...
  123456789ABCDEF123456789ABCDEF
```

Once the tokens have been initialized, the auto-generated private keys can be safely removed from the filesystem.

```
rm -rf vulcanexus_demo_keystore/enclaves/talker_listener/talker/key.pem
rm -rf vulcanexus_demo_keystore/enclaves/talker_listener/listener/key.pem
```

The PKCS #11 URI with the HSM information has to be stored in the key.p11 file. When inquiring the HSM emulator for the stored tokens, the pin is going to be required. Either set the GNUTLS_PIN environment variable with the pin or enter it when asked by the tool. Run the following commands to extract the required URIs and save them automatically to the files (administrative privileges may be required):

```
GNUTLS_PIN=VulcanexusDemoTalker p11tool --provider /usr/lib/softsm/libsoftsm2.so --
  --list-tokens --login | grep "token=vulcanexus_demo_talker" | awk '{print $2}' >
  vulcanexus_demo_keystore/enclaves/talker_listener/talker/key.p11
GNUTLS_PIN=VulcanexusDemoListener p11tool --provider /usr/lib/softsm/libsoftsm2.so --
  --list-tokens --login | grep "token=vulcanexus_demo_listener" | awk '{print $2}' >
  vulcanexus_demo_keystore/enclaves/talker_listener/listener/key.p11
```
Even though the URIs have been saved, the corresponding pins have not been included into the URIs. Two options can be followed:

Environment variable

```
export FASTDDS_PKCS11_PIN=VulcanexusDemo<Talker|Listener>
```

PKCS #11 URI

Edit both key.p11 files and add at the end the corresponding pin: ?

```
pin-value=VulcanexusDemo<Talker|Listener>.
```

**Execute ROS 2 demo nodes with security**

Security in ROS 2 is enabled by means of environment variables as explained in ROS 2 documentation. Remember to correctly setup your environment in each terminal.

```
source /opt/vulcanexus/iron/setup.bash
export ROS_SECURITY_KEYSTORE=~/.vulcanexus_hsm_ws/vulcanexus_demo_keystore
export ROS_SECURITY_ENABLE=true
export ROS_SECURITY_STRATEGY=Enforce
```

The latest environment variable ensures that if the security files are not correct, ROS 2 will not be initialized.

If the pin is not included in the PKCS #11 URI within the key.p11 file, then remember to also export the following environment variable with the corresponding pin depending on the node being launched:

```
export FASTDDS_PKCS11_PIN=VulcanexusDemo<Talker|Listener>
```

Please, remember to also source Vulcanexus installation before launching the following commands.

In the first terminal launch the talker:

```
ros2 run demo_nodes_cpp talker --ros-args --enclave /talker_listener/talker
```

In the second terminal launch the listener:

```
ros2 run demo_nodes_cpp listener --ros-args --enclave /talker_listener/listener
```

Communication between both nodes is established.

**Security breach tryout**

If a second listener is launched in a third terminal without security (do not set the environment variables but remember to source Vulcanexus installation), this node will not be included into the communication.

```
source /opt/vulcanexus/iron/setup.bash
ros2 run demo_nodes_cpp listener
```
4.10.2 Vulcanexus QoS Tutorials

ROS 2 only allows for the configuration of certain middleware QoS (see ROS 2 QoS policies). However, Vulcanexus offers extended configuration capabilities to take full advantage of the features in Fast DDS. Vulcanexus ensures maximum customization possibilities over Quality of Service (QoS) policies for ROS 2 networks.

Listed below are the tutorials for configuring QoS in ROS 2 environments.

Configuring Fast-DDS QoS via XML profiles

- Configuring Fast DDS in Vulcanexus for ROS 2 applications
- Changing publication mode
- XML configuration
  - XML configuration file location
  - Applying different profiles to different entities
- Example

Vulcanexus offers the possibility of fully configuring Fast DDS’ QoS policies through XML profile definition. This can be done in several ways depending on whether the application used is a native Fast DDS application or a Vulcanexus application using Fast DDS’s RMW implementation. This guide will focus on the latter. For more information regarding XML profiles in Fast DDS please refer to the XML configuration page from Fast DDS’s documentation.

Configuring Fast DDS in Vulcanexus for ROS 2 applications

ROS 2 only allows for the configuration of certain middleware QoS (see ROS 2 QoS policies). However, rmw_fastrtps offers extended configuration capabilities to take full advantage of all the features available in Fast DDS. This section describes how to specify this extended configuration.

Changing publication mode

rmw_fastrtps in Vulcanexus uses asynchronous publication by default. This can be changed setting the environment variable RMW_FASTRTPS PUBLICATION_MODE to one of the following allowed values:

- **ASYNCHRONOUS**: asynchronous publication mode. Setting this mode implies that when the publisher invokes the write operation, the data is copied into a queue, a background thread (asynchronous thread) is notified about the addition to the queue, and control of the thread is returned to the user before the data is actually sent. The background thread is in charge of consuming the queue and sending the data to every matched reader.

- **Synchronous**: synchronous publication mode. Setting this mode implies that the data is sent directly within the context of the user thread. This entails that any blocking call occurring during the write operation would block the user thread, thus preventing the application from continuing its operation. It is important to note that this mode typically yields higher throughput rates at lower latencies, since there is no notification nor context switching between threads.

- **Auto**: let Fast DDS select the publication mode. This implies using the publication mode set in the XML file, or otherwise, the default value set in Fast DDS (see PublishModeQosPolicy).

rmw_fastrtps defines two configurable parameters in addition to ROS 2 QoS policies. Said parameters, and their default values under ROS 2, are:
XML configuration

To use specific Fast-DDS features within a Vulcanexus application, XML configuration files can be used to configure a wide set of QoS. Please refer to XML Profiles page for Fast DDS native applications in Fast DDS’s documentation to see the whole list of configuration options available in Fast DDS.

When configuring rmw_fastrtps using XML files, there are certain points that have to be taken into account:

- QoS contained in rmw_qos_profile_t are always honored, unless set to _SYSTEM_DEFAULT. In that case, XML values, or Fast DDS default values in the absences of XML ones, are applied. This means that if any QoS in rmw_qos_profile_t is set to something other than _SYSTEM_DEFAULT, the corresponding value in the XML is ignored.

- By default, rmw_fastrtps overrides the values for MemoryManagementPolicy and PublishModeQosPolicy. This means that the values configured in the XML for these two parameters will be ignored. Instead, PREALLOCATED_WITH_REALLOC_MEMORY_MODE and ASYNCHRONOUS_PUBLISH_MODE are used respectively.

- The overriding of MemoryManagementPolicy and PublishModeQosPolicy can be avoided by setting the environment variable RMW_FASTRTPS_USE_QOS_FROM_XML to 1 (its default value is 0). This will make rmw_fastrtps use the values defined in the XML for MemoryManagementPolicy and PublishModeQosPolicy. Bear in mind that setting this environment variable but not setting these policies in the XML results in using the default values in Fast DDS. These are different from the aforementioned rmw_fastrtps default values (see MemoryManagementPolicy and PublishModeQosPolicy). In general, setting RMW_FASTRTPS_USE_QOS_FROM_XML effectively overrides whatever configuration was set with RMW_FASTRTPS_PUBLICATION_MODE, setting the publication mode to the value specified in the XML, or to the Fast DDS default publication mode if none is set in the XML.

The following table summarizes which values are used or ignored according to the configured variables:

<table>
<thead>
<tr>
<th>RMW_FASTRTPS_USE_QOS_FROM_XML</th>
<th>Fast DDS XML QoS</th>
<th>Fast DDS XML history memory policy and publication mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (default)</td>
<td>Overridden by rmw_qos_profile_t</td>
<td>Overridden by rmw_fastrtps default value</td>
</tr>
<tr>
<td>0 (default)</td>
<td>overriden by rmw_qos_profile_t</td>
<td>Overridden by rmw_fastrtps default value</td>
</tr>
<tr>
<td>0 (default)</td>
<td>Used</td>
<td>Overridden by rmw_fastrtps default value</td>
</tr>
<tr>
<td>1</td>
<td>Overridden by rmw_qos_profile_t</td>
<td>Used</td>
</tr>
<tr>
<td>1</td>
<td>Overriden by rmw_qos_profile_t</td>
<td>Used</td>
</tr>
<tr>
<td>1</td>
<td>Used</td>
<td>Used</td>
</tr>
</tbody>
</table>
XML configuration file location

There are two possibilities for providing *Fast DDS* with XML configuration files:

- **Recommended**: Setting the location with environment variable `FASTRTPS_DEFAULT_PROFILES_FILE` to contain the path to the XML configuration file (see Environment Variables).

```bash
export FASTRTPS_DEFAULT_PROFILES_FILE=<path_to_xml_file>
```

- **Alternative**: Placing the XML file in the running application directory under the name `DEFault_FASTRTPS_PROFILES.xml`.

For example:

```bash
export FASTRTPS_DEFAULT_PROFILES_FILE=<path_to_xml_file>
export RMW_FASTRTPS_USE_QOS_FROM_XML=1
ros2 run <package> <application>
```

Applying different profiles to different entities

`rmw_fastrtps` allows for the configuration of different entities with different QoS using the same XML file. For doing so, `rmw_fastrtps` locates profiles in the XML based on topic names.

Creating publishers/subscribers with different profiles

- To configure a publisher, define a `<data_writer>` profile with attribute `profile_name=topic_name`, where `topic_name` is the name of the topic prepended by the node namespace (which defaults to `""` if not specified), i.e. the node’s namespace followed by topic name used to create the publisher. Mind that topic names always start with `/` (it is added when creating the topic if not present), and that namespace and topic name are always separated by one `/`. If such profile is not defined, `rmw_fastrtps` attempts to load the `<data_writer>` profile with attribute `is_default_profile="true"`.

- To configure a subscriber, define a `<data_reader>` profile with attribute `profile_name=topic_name`, where `topic_name` is the name of the topic prepended by the node namespace (which defaults to `""` if not specified), i.e. the node’s namespace followed by topic name used to create the subscriber. Mind that topic names always start with `/` (it is added when creating the topic if not present), and that namespace and topic name are always separated by one `/`. If such profile is not defined, `rmw_fastrtps` attempts to load the `<data_reader>` profile with attribute `is_default_profile="true"`.

The following table presents different combinations of node namespaces and user specified topic names, as well as the resulting topic names and the suitable profile names:

<table>
<thead>
<tr>
<th>User specified topic name</th>
<th>Node namespace</th>
<th>Final topic name</th>
<th>Profile name</th>
</tr>
</thead>
<tbody>
<tr>
<td>chatter</td>
<td>DEFAULT (&quot;&quot;&quot;)</td>
<td>/chatter</td>
<td>/chatter</td>
</tr>
<tr>
<td>chatter</td>
<td>test_namespace</td>
<td>/test_namespace/chatter</td>
<td>/test_namespace/chatter</td>
</tr>
<tr>
<td>/chatter</td>
<td>test_namespace</td>
<td>/chatter</td>
<td>/chatter</td>
</tr>
<tr>
<td>/chatter</td>
<td>/test_namespace</td>
<td>/chatter</td>
<td>/chatter</td>
</tr>
</tbody>
</table>

**Important**: Node namespaces are NOT prepended to user specified topic names starting with `/`, a.k.a Fully Qualified Names (FQN). For a complete description of topic name remapping please refer to Remapping Names.
Creating services with different profiles

ROS 2 services contain a subscriber for receiving requests, and a publisher to reply to them. rmw_fastrtps allows for configuring each of these endpoints separately in the following manner:

- To configure the request subscriber, define a `<data_reader>` profile with attribute `profile_name=topic_name`, where `topic_name` is the name of the service after mangling. For more information on name mangling, please refer to Topic and Service name mapping to DDS. If such profile is not defined, rmw_fastrtps attempts to load a `<data_reader>` profile with attribute `profile_name="service"`. If neither of the previous profiles exist, rmw_fastrtps attempts to load the `<data_reader>` profile with attribute `is_default_profile="true"`.

- To configure the reply publisher, define a `<data_writer>` profile with attribute `profile_name=topic_name`, where `topic_name` is the name of the service after mangling. If such profile is not defined, rmw_fastrtps attempts to load a `<data_writer>` profile with attribute `profile_name="service"`. If neither of the previous profiles exist, rmw_fastrtps attempts to load the `<data_writer>` profile with attribute `is_default_profile="true"`.

Creating clients with different profiles

ROS 2 clients contain a publisher to send requests, and a subscription to receive the service’s replies. rmw_fastrtps allows for configuring each of these endpoints separately in the following manner:

- To configure the requests publisher, define a `<data_writer>` profile with attribute `profile_name=topic_name`, where `topic_name` is the name of the service after mangling. If such profile is not defined, rmw_fastrtps attempts to load a `<data_writer>` profile with attribute `profile_name="client"`. If neither of the previous profiles exist, rmw_fastrtps attempts to load the `<data_writer>` profile with attribute `is_default_profile="true"`.

- To configure the reply subscription, define a `<data_reader>` profile with attribute `profile_name=topic_name`, where `topic_name` is the name of the service after mangling. If such profile is not defined, rmw_fastrtps attempts to load a `<data_reader>` profile with attribute `profile_name="client"`. If neither of the previous profiles exist, rmw_fastrtps attempts to load the `<data_reader>` profile with attribute `is_default_profile="true"`.

Creating ROS contexts and nodes

ROS `context` and `node` entities are mapped to Fast DDS Participant entity, according to the following table:

<table>
<thead>
<tr>
<th>ROS entity</th>
<th>Fast DDS entity since Foxy</th>
<th>Fast DDS entity in Eloquent &amp; below</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context</td>
<td>Participant</td>
<td>Not DDS direct mapping</td>
</tr>
<tr>
<td>Node</td>
<td>Not DDS direct mapping</td>
<td>Participant</td>
</tr>
</tbody>
</table>

This means that on Foxy and later releases, contexts can be configured using a `<Participant>` profile with attribute `is_default_profile="true"`. The same profile will be used in Eloquent and below to configure nodes.

For example, a profile for a ROS 2 context on Foxy and later releases would be specified as:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
  <participant profile_name="participant_profile_ros2" is_default_profile="true">
    <rtps>
      <name>profile_for_ros2_context</name>
    </rtps>
  </participant>
</profiles>
```

(continues on next page)
Example

The following example uses both the ROS 2 `demo_nodes_cpp` package’s talker/listener demo and the client/service demo, configuring Fast DDS to publish synchronously, and to have dynamically allocated publisher and subscriber histories.

Create a XML file `vulcanexus_example.xml` and save it in `path/to/xml/`

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">

    <participant profile_name="participant_profile_ros2" is_default_profile="true">
        <rtps>
            <name>profile_for_ros2_context</name>
        </rtps>
    </participant>

    <!-- Default publisher profile -->
    <data_writer profile_name="default publisher profile" is_default_profile="true">
        <qos>
            <publishMode>
                <kind>SYNCHRONOUS</kind>
            </publishMode>
        </qos>
        <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
    </data_writer>

    <!-- Publisher profile for topic helloworld -->
    <data_writer profile_name="helloworld">
        <qos>
            <publishMode>
                <kind>SYNCHRONOUS</kind>
            </publishMode>
        </qos>
    </data_writer>

    <!-- Request subscriber profile for services -->
    <data_reader profile_name="service">
        <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
    </data_reader>

    <!-- Request publisher profile for clients -->
    <data_writer profile_name="client">
        <qos>
            <publishMode>
                <kind>ASYNCHRONOUS</kind>
            </publishMode>
        </qos>
    </data_writer>

</profiles>
```

(continues on next page)
For the talker/listener demo application.

1. Open one terminal and run:

```bash
export FASTRTPS_DEFAULT_PROFILES_FILE=path/to/xml/vulcanexus_example.xml
export RMW_FASTRTPS_USE_QOS_FROM_XML=1
ros2 run demo_nodes_cpp talker
```

2. Open one terminal and run:

```bash
export FASTRTPS_DEFAULT_PROFILES_FILE=path/to/xml/vulcanexus_example.xml
export RMW_FASTRTPS_USE_QOS_FROM_XML=1
ros2 run demo_nodes_cpp listener
```

The same approach can be used when running the add_two_ints client/service demo application.

1. Open one terminal, run:

```bash
# Client
export FASTRTPS_DEFAULT_PROFILES_FILE=path/to/xml/vulcanexus_example.xml
export RMW_FASTRTPS_USE_QOS_FROM_XML=1
ros2 run demo_nodes_cpp add_two_ints_client
```

2. Open another terminal and run:

```bash
# Server
export FASTRTPS_DEFAULT_PROFILES_FILE=path/to/xml/vulcanexus_example.xml
export RMW_FASTRTPS_USE_QOS_FROM_XML=1
ros2 run demo_nodes_cpp add_two_ints_server
```
Modifying Ownership and Ownership Strength QoS Policy

- **Background**
- **Prerequisites**
- **XML Profile definition**
- **Execute ROS 2 demo nodes with modified QoS**

**Background**

Fast DDS over Vulcanexus offers the possibility of fully configuring QoS policy through XML profile definition. This tutorial provides step-by-step instructions to modify the Ownership QoS within the ROS 2 talker/listener demo (see Ownership QoS Policy).

This QoS Policy specifies whether it is allowed for multiple DataWriter to update the same instance of data, and if so, how these modifications should be arbitrated.

**Prerequisites**

The first prerequisite is to have Vulcanexus Iron installed (see Linux binary installation or Linux installation from sources).

Please, remember to source the environment in every terminal in this tutorial.

```
source /opt/vulcanexus/iron/setup.bash
```

In a terminal sourced with the previous line, run the following command to install the ROS 2 demo-nodes-cpp tutorial (administrative privileges may be required):

```
apt-get update && apt install -y ros-iron-demo-nodes-cpp
```

**XML Profile definition**

In order to specify the desired custom configuration for the Ownership QoS policy, an XML file is required (see Fast DDS XML profiles). In any directory, run the following commands to create two files named `small_strength.xml` and `large_strength.xml`:

```
touch small_strength.xml
touch large_strength.xml
```

Open those files with your preferred editor and write down the following XML code to the `small_strength.xml` file. For the `large_strength.xml` file, write down the same code, but changing the value 10 for any other larger number, for example, 58.

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
  <data_writer profile_name="/chatter">
    <qos>
      <ownership>
```

(continues on next page)
<kind>EXCLUSIVE</kind>
</ownership>
<ownershipStrength>
 <value>10</value>
</ownershipStrength>
</qos>
</data_writer>

<data_reader profile_name="/chatter">
 <qos>
  <ownership>
   <kind>EXCLUSIVE</kind>
  </ownership>
 </qos>
</data_reader>
</profiles>

For the next section of this tutorial, let us consider both created XML files are stored in the ~/ directory.

**Execute ROS 2 demo nodes with modified QoS**

Open one terminal and source Vulcanexus environment. To set `small_strength.xml` to define the profile configuration used on the creation of ROS 2 nodes, populating the `FASTRTPS_DEFAULT_PROFILES_FILE` environment variable to point out to the file is needed. Then, you can run `ros-demo-nodes-cpp` program to create a listener with `EXCLUSIVE_OWNERSHIP_POLICY` QoS:

```
source /opt/vulcanexus/iron/setup.bash
export FASTRTPS_DEFAULT_PROFILES_FILE=~/small_strength.xml
ros2 run demo_nodes_cpp listener
```

Open another terminal and source Vulcanexus environment. To create `ros-demo-nodes-cpp` talker, run the following commands:

```
source /opt/vulcanexus/iron/setup.bash
export FASTRTPS_DEFAULT_PROFILES_FILE=~/small_strength.xml
ros2 run demo_nodes_cpp talker
```

**Note:** Note that the profile used by the listener is the data_reader profile, and the one used by the talker is the data_writer one.

Now, both nodes should be communicating. It can be seen that, the Hellow World messages that the talker sends, are being received by the listener. The number of those messages coincides.

In a third terminal, source Vulcanexus environment. To create another `ros-demo-nodes-cpp` talker, but now with greater ownership strength (see Ownership Strength QoS Policy), the `FASTRTPS_DEFAULT_PROFILES_FILE` will point out to `large_strength.xml`:

```
source /opt/vulcanexus/iron/setup.bash
export FASTRTPS_DEFAULT_PROFILES_FILE=~/large_strength.xml
ros2 run demo_nodes_cpp talker
```
Now, it can be seen that, although the first talker keeps sending messages, the messages being read by listener are those of the newly created talker, i.e. the message index sent by second talker matches that of the arriving message in listener. This is happening due to the second talker setting a higher Ownership Strength value than the first one.

If now the second talker process is killed, the messages being received by the listener are the ones from the first talker.

**Modifying Partition QoS Policy**

- **Background**
- **Prerequisites**
- **XML Profile definition**
- **Execute ROS 2 demo nodes with modified QoS**

**Background**

Fast DDS over Vulcanexus offers the possibility of fully configuring QoS policy through XML profile definition. This tutorial provides step-by-step instructions to modify the Partition QoS Policy within the ROS 2 talker/listener demo (see **Partition QoS Policy**).

This QoS Policy allows the introduction of a logical partition inside the physical partition introduced by a domain. For a DataReader to see the changes made by a DataWriter, not only the Topic must match, but also they have to share at least one logical partition.

**Prerequisites**

The first prerequisite is to have Vulcanexus Iron installed (see **Linux binary installation** or **Linux installation from sources**).

Please, remember to source the environment in every terminal in this tutorial.

```
source /opt/vulcanexus/iron/setup.bash
```

In a terminal sourced with the previous line, run the following command to install the ROS 2 demo-nodes-cpp tutorial (administrative privileges may be required):

```
apt-get update && apt install -y ros-iron-demo-nodes-cpp
```

**XML Profile definition**

In order to specify the desired custom configuration for the Partition QoS policy, an XML file is required (see **Fast DDS XML profiles**). In any directory, run the following commands to create two files named `corresponding_partition.xml` and `another_partition.xml`:

```
touch corresponding_partition.xml
touch another_partition.xml
```

Open those files with your preferred editor, and write down the following XML code to the `corresponding_partition.xml` file.

```xml
<qos_policy>
  <partitions>
    <partition name="corresponding_partition">
      <node>
        <node_names>
          <node_names>
            <name>corresponding_node</name>
          </node_names>
        </node>
      </partition>
    </partitions>
  </qos_policy>
```

Write down the following XML code to the `another_partition.xml` file.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles"
  <data_writer profile_name="/chatter">
    <qos>
      <partition>
        <names>
          <name>part3</name>
        </names>
      </partition>
    </qos>
  </data_writer>
  
  <data_reader profile_name="/chatter">
    <qos>
      <partition>
        <names>
          <name>part1</name>
          <name>part4</name>
        </names>
      </partition>
    </qos>
  </data_reader>
</profiles>
```

For the next section of this tutorial, let us consider both created XML files are stored in the `~` directory.

**Execute ROS 2 demo nodes with modified QoS**

Open one terminal and source Vulcanexus environment. To set `corresponding_partition.xml` to define the profile configuration used on the creation of ROS 2 nodes, populating the `FASTRTPS_DEFAULT_PROFILES_FILE` environment variable to point out to the file is needed. Then, you can run `ros-demo-nodes-cpp` program to create a listener belonging to `part1` and `part4` partitions:

```
source /opt/vulcanexus/iron/setup.bash
export FASTRTPS_DEFAULT_PROFILES_FILE=/~/corresponding_partition.xml
ros2 run demo_nodes_cpp listener
```
Open another terminal and source Vulcanexus environment. To create `ros-demo-nodes-cpp` talker belonging to `part1` and `part2`, run the following commands:

```
source /opt/vulcanexus/iron/setup.bash
export FASTRTPS_DEFAULT_PROFILES_FILE=~/corresponding_partition.xml
ros2 run demo_nodes_cpp talker
```

**Note:** Note that the profile used by the listener is the data_reader profile, and the one used by the talker is the data_writer one.

Now, both nodes should be communicating, as they belong to at least one same partition, which is `part1` in this case. It can be seen that, the *Hellow World* messages that the talker sends, are being received by the listener.

Talker process can be killed using `Ctrl+C`. Then, in the same terminal, to create `ros-demo-nodes-cpp` talker belonging to `part3`, we will set `FASTRTPS_DEFAULT_PROFILES_FILE` to point out to `another_partition.xml`. Run the following commands:

```
export FASTRTPS_DEFAULT_PROFILES_FILE=~/another_partition.xml
ros2 run demo_nodes_cpp talker
```

Now talker and listener are not communicating, as they don’t belong to any same partition. Talker and listener are isolated from one another, as listener reads messages just from `part1` or `part4` partitions, while talker is publishing its messages for `part3` only.

### Persistent Data using Durability QoS

- **Background**
- **Prerequisites**
- **XML Profile definition**
- **Environment variables set up**
- **Testing Publisher Persistency**
- **Testing Subscriber Persistency**

**Background**

The DDS Durability QoS Policy specifies how much importance ROS 2 nodes give to the exchanged data. The possible options are briefly explained here:

- **Durability**
  - **Volatile:** Old data values are ignored. Any new node will not receive any previous data.
  - **Transient local:** Old data values are important. Any new node will receive data generated before its creation.
  - **Transient:** Data is so important that it is backed up or persisted into a database file. This will guarantee that if a node crashes it can be reenacted and keep operating without data losses.

Only **Volatile** and **Transient Local** durability options are ROS 2 builtins. In order to set up **Transient** nodes, xml configuration files are required as explained in the XML profiles tutorial. This tutorial provides guidelines on how to set up persistent nodes.
Prerequisites

Vulcanexus Iron should be installed *(follow the steps here)*. For testing sake, using a docker container is often a more convenient approach *(docker setup)*. In order to test this feature, a *Vulcanexus* lightweight docker image *(as core or micro)* is enough. The images are available for download on *Vulcanexus* website.

```
docker load -i ./ubuntu-vulcanexus-iron-micro.tar

# Terminal 1
docker run -ti --name persistence_testing ubuntu-vulcanexus:ironmicro

# Terminal 2
docker exec -ti persistence_testing /vulcanexus_entrypoint.sh /bin/bash
```

This tutorial requires two terminals: one for the data writer (1) and the other for the reader (2). Note that for user convenience, the terminal that launches the container (1) has the overlay loaded in the *entrypoint* but any other terminal that connects to it (2) must explicitly load the overlay calling the *vulcanexus_entripoint.sh* script.

The concepts introduced here are applied to any ROS 2 nodes and the tutorial will not use a specific package but rely on the ROS 2 CLI capabilities.

To create a publisher node from the ROS 2 command line that sends *TIMES* samples:

```
ros2 topic pub [-t TIMES] [--qos-durability {system_default,transient_local,volatile,unknown}] topic_name message_type [values]
```

To create a subscriber node from ROS 2 command line:

```
ros2 topic echo [--qos-durability {system_default,transient_local,volatile,unknown}] --topic_name [message_type]
```

**Note:** The *ros2 topic* command line interface provides convenient parameters for defining node QoS' as *--qos-durability*. Specifying those on ROS 2 packages requires modifying the sources to appropriately set *ROS 2 client library* QoS structures as *rclepp::QoS* or *rclpy_qos*.

**XML Profile definition**

In order to specify the desired custom configuration for the Durability QoS policy, an XML file is required *(see Fast DDS XML profiles)*.

Usually a single configuration file is enough but this tutorial requires two independent persistent databases (for writer and reader) in order to show the persistency advantages.

In the working directory of choice *(henceforth /home/tutorial/)* create two files:

- A publisher node configuration file: *writer_config.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<dds>
  <profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant is_default_profile="true" profile_name="persistence_service_participant">
      <rtps>
```

(continues on next page)
A subscriber node configuration file: reader_config.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<dds>
  <profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant is_default_profile="true" profile_name="persistence_service_participant">
      <rtps>
        <propertiesPolicy>
          <properties>
            <!-- Select persistence plugin -->
            <property>
              <name>dds.persistence.plugin</name>
              <value>builtin.SQLITE3</value>
            </property>
            <!-- Database file name -->
            <property>
              <name>dds.persistence.sqlite3.filename</name>
              <value>writer_database.db</value>
            </property>
          </properties>
          <propertiesPolicy>
            <!-- Persistence GUID -->
            <property>
              <name>dds.persistence.guid</name>
              <value>56.55.4C.43.41.4E.45.58.55.53.5F.50|53.5F.44.57</value>
            </property>
          </propertiesPolicy>
        </rtps>
      </participant>
    </profiles>
  </dds>
```
For each node is specified:

- A database filename as the `dds.persistence.sqlite3.filename` property in the default participant profile.

- An endpoint (data_writer or data_reader) profile which:

  - is associated to the node using the `topic name` as `profile_name` attribute. In this tutorial the `topic name` will be `persistency_test`.

  - The endpoint requires a `GUID` specified as the `dds.persistence.guid` property.

  - The endpoint Durability QoS must be set to `transient`.

**Environment variables set up**

ROS 2 nodes will locate their associated xml configuration files using the `FASTRTPS_DEFAULT_PROFILES_FILE` environment variable. For example:

```bash
export FASTRTPS_DEFAULT_PROFILES_FILE=/home/tutorial/writer_config.xml
```
Testing Publisher Persistency

The following steps will show how using persistent endpoints prevents subscribers from receiving duplicated data when a publisher crashes.

1. Launch persistent publisher and subscriber.

   # Terminal 1
   ```
   export FASTRTPS_DEFAULT_PROFILES_FILE=/home/tutorial/writer_config.xml
   ros2 topic pub --times 5 --qos-durability system_default /persistency_test
   →std_msgs/String "{'data': 'Hello'}"
   ```

   # Terminal 2
   ```
   export FASTRTPS_DEFAULT_PROFILES_FILE=/home/tutorial/reader_config.xml
   ros2 topic echo --qos-durability system_default /persistency_test std_msgs/
   →String
   ```

   Note that we must specify `system_default` as durability in order to enforce the use of the xml file provided value.

2. Relaunch the publisher again and check the subscriber is able to receive the samples.

   # Terminal 1
   ```
   ros2 topic pub --times 5 --qos-durability system_default /persistency_test
   →std_msgs/String "{'data': 'Hello'}"
   ```

3. Delete the publisher database. Now the publisher state is reset. Relaunch the publisher and check the first 10 samples are discarded by the subscriber because they were already received.

   # Terminal 1
   ```
   rm writer_database.db
   ros2 topic pub --times 11 --qos-durability system_default /persistency_test
   →std_msgs/String "{'data': 'Hello'}"
   ```

Testing Subscriber Persistency

The following steps will show how using persistent endpoints prevents subscribers from receiving duplicated data when the subscriber crashes and it’s relaunched.

1. Launch persistent publisher and subscriber.

   # Terminal 1
   ```
   export FASTRTPS_DEFAULT_PROFILES_FILE=/home/tutorial/writer_config.xml
   ros2 topic pub --times 5 --qos-durability system_default /persistency_test
   →std_msgs/String "{'data': 'Hello'}"
   ```

   # Terminal 2
   ```
   export FASTRTPS_DEFAULT_PROFILES_FILE=/home/tutorial/reader_config.xml
   ros2 topic echo --qos-durability system_default /persistency_test std_msgs/
   →String
   ```

2. Delete the publisher database. Relaunch the publisher and check the first 5 samples are discarded by the subscriber because they were already received.
3. Delete the publisher database. Restart the subscriber node. Relaunch the publisher and check the first 10 samples are discarded by the subscriber because they were already received. Note the new subscriber deduced it from its persistence database.

```
# Terminal 1
rm writer_database.db
ros2 topic pub --times 10 --qos-durability system_default /persistency_test
  std_msgs/String "\{data: 'Hello'\}"

# Terminal 2
<CTRL-C>
ros2 topic echo --qos-durability system_default /persistency_test std_msgs/
  String
```

Static discovery between ROS 2 nodes with Initial Peers

- **Background**
- **Prerequisites**
  - Adding initial peers QoS Policy
    - Obtain Vulcanexus docker container IP address
    - XML configuration file location
- **Example**
  - Unicast Example

**Background**

As most ROS 2 developers already know, Fast DDS is the default middleware of ROS 2, so this tutorial focuses on applying a direct configuration to the Fast DDS middleware so that you can modify its settings without the need to do it through the exposed API of ROS 2.

Vulcanexus offers the possibility of fully configuring Fast DDS’ QoS policies through XML profile definition (see ROS 2 QoS policies). For more information on how to configure ROS 2 through XML profiles please refer to Configuring Fast-DDS QoS via XML profiles tutorial.

This section describes how to specify this extended policies, in particular the initial peers configuration, which applies to the DDS DomainParticipant QoS. For this purpose, it is important to take into consideration the mapping between ROS 2 entities and DDS DomainParticipants (see Node to Participant mapping). In short, a ROS 2 Context is equivalent to a DDS DomainParticipant, so a DDS DomainParticipant can contain multiple ROS 2 Nodes (see Creating ROS contexts and nodes). Therefore, this tutorial only applies to deployments with more than one ROS 2 Context.

According to the DDS standard, each DomainParticipant must listen for incoming DomainParticipant discovery meta-traffic in two different ports, one linked with a multicast address, and another one linked to a unicast address. Vulcanexus, together with Fast DDS, allows configuring an initial peers list, which contains one or more such IP-port
address pairs corresponding to remote DomainParticipants discovery listening resources, so that the local Domain-
Participant will not only send its PDP traffic to the default multicast network address, but also to all the IP-port pairs
specified in the initial peers list.

A complete description of the initial peers list and its configuration can be found in Initial Peers discovery mechanism
in Fast DDS documentation.

Prerequisites

First of all, make sure that Vulcanexus Iron is installed. The docker installation is required for this tutorial (see Docker
installation).

Open two terminals, and run the Vulcanexus Iron image in each one with the following command:

- Terminal 1:
  
  docker run -it --name ros2_context_1 ubuntu-vulcanexus:iron-desktop

- Terminal 2:
  
  docker run -it --name ros2_context_2 ubuntu-vulcanexus:iron-desktop

Note: It is highly recommended to complete the Configuring Fast-DDS QoS via XML profiles tutorial to learn how to
configure ROS 2 via XML configuration files.

Adding initial peers QoS Policy

In the docker container named ros2_context_1, create a XML file, and complete it with the following example:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles"
  <participant profile_name="ros2_context1_participant" is_default_profile="true">
    <rtps>
      <builtin>
        <initialPeersList>
          <locator>
            <udpv4>
              <address>172.17.0.3</address> <!-- ros2_context_2 IP address-->
            </udpv4>
            <port>7412</port> <!-- ros2_context_2 default port -->
          </locator>
        </initialPeersList>
      </builtin>
    </rtps>
  </participant>
</profiles>
```

Note: (Advanced users) According to the RTPS standard (Section 9.6.1.1), the participant discovery traffic unicast
listening ports are defined in the Well Known Ports section of the Fast DDS documentation. Thus, in this example, the
Participant operates in Domain 0 (default domain) and its ID is 1, so its discovery traffic unicast listening port is 7412. By default *eProsima Fast DDS* uses as initial peers the metatraffic multicast locators (network addresses).

It is required to include the XML file in at least one of the Vulcanexus docker containers, as the participant with the initial peer set would let the other participant know the address and port through which they will communicate.

### Obtain Vulcanexus docker container IP address

The IP address included in the document must be updated with the Vulcanexus docker container assigned IP address. Open a terminal, and list the Vulcanexus docker containers.

```
docker container list
```

The output should look like the following table:

<table>
<thead>
<tr>
<th>CONTAINER ID</th>
<th>IMAGE</th>
<th>COMMAND</th>
<th>CREATED</th>
<th>STATUS</th>
<th>PORTS</th>
<th>NAMES</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;container ID1&gt;</td>
<td>ubuntu-vulcanexus:iron-desktop</td>
<td>“…vulcanexus...”</td>
<td>2 minutes ago</td>
<td>Up 2 minutes</td>
<td>ros2_context_1</td>
<td></td>
</tr>
<tr>
<td>&lt;container ID2&gt;</td>
<td>ubuntu-vulcanexus:iron-desktop</td>
<td>“…vulcanexus...”</td>
<td>2 minutes ago</td>
<td>Up 2 minutes</td>
<td>ros2_context_2</td>
<td></td>
</tr>
</tbody>
</table>

Using the container name, check its detailed information. The following command also filters the output to obtain only the container IP address:

```
docker inspect ros2_context_2 | grep IPAddress
```

Edit the previously XML file created in the `ros2_context_1` container, and include the IP address of the `ros2_context_2` container. Make sure the IP address set in the xml file is NOT the container’s own IP address.

### XML configuration file location

In order to use the profiles loaded from XML configuration files, it is required to let the application know it. There are two possibilities for providing ROS 2 with XML configuration files:

- **Recommended**: Setting the location in `FASTRTPS_DEFAULT_PROFILES_FILE` environment variable, which may contain the path to the XML configuration file (see Environment Variables).

  ```
  export FASTRTPS_DEFAULT_PROFILES_FILE=<path_to_xml_file>
  ```

- **Alternative**: Renaming the XML file to `DEFAULT_FASTRTPS_PROFILES.xml` and placing it in the running application directory. For example:

  ```
  cp <path_to_xml_file> DEFAULT_FASTRTPS_PROFILES.xml
  export RMW_FASTRTPS_USE_QOS_FROM_XML=1
  ```
Example

Once the ROS 2 context running in ros2_context_1 Vulcanexus container has been configured, run a demo application in each container.

- ros2_context_1 container (terminal 1):
  
  ```
  ros2 run demo_nodes_cpp talker
  ```

- ros2_context_2 container (terminal 2):
  
  ```
  ros2 run demo_nodes_cpp listener
  ```

The listener would receive the messages sent by the talker.

Unicast Example

In the previous example, both multicast and unicast communication were used for entity discovery. Next, it is shown how to fully disable multicast communication for discovery, so that the only means of discovery is through unicast communication. In this case, one of the participants will know in advance the listening address of the other participant and will send its discovery information via unicast communication. The second participant will then reply to the first participant using the network address contained in the information received. After the discovery, both will start the transmission of user data.

Replace the content of the existing XML configuration file in the ros2_context_1 container with the one shown below.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
  <participant profile_name="ros2_context1Participant" is_default_profile="true">
    <rtps>
      <builtin>
        <initialPeersList>
          <locator>
            <udpv4>
              <address>172.17.0.3</address> <!-- ros2_context_2 IP address -->
              <port>7777</port> <!-- ros2_context_2 port -->
            </udpv4>
          </locator>
        </initialPeersList>
        <metatrafficUnicastLocatorList>
          <locator>
            <udpv4>
              <address>172.17.0.2</address> <!-- ros2_context_1 IP address -->
              <port>7666</port> <!-- ros2_context_1 port -->
            </udpv4>
            </locator>
          </metatrafficUnicastLocatorList>
        </builtin>
      </rtps>
    </participant>
  </profiles>
```

(continues on next page)
In this case, by defining a meta-traffic unicast locator, the Participant creates a unicast meta-traffic receiving resource (communication status and discovery protocol data) for each address-port pair specified (see Disabling all Multicast Traffic). In that way, and by setting a custom port as initial peer, both ROS 2 context would communicate exclusively by each specified network address and ports (avoiding multicast and the usage of any default ports).

Then, create a new XML file and complete it with the following example in the `ros2_context_2` container:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="ros2_context2_participant" is_default_profile="true">
        <rtps>
            <builtin>
                <initialPeersList>
                    <locator>
                        <udpv4>
                            <address>172.17.0.2</address> <!-- ros2_context_1 IP address -->
                            <port>7666</port> <!-- ros2_context_1 port -->
                        </udpv4>
                    </locator>
                </initialPeersList>
                <metatrafficUnicastLocatorList>
                    <locator>
                        <udpv4>
                            <address>172.17.0.3</address> <!-- ros2_context_2 IP address -->
                            <port>7777</port> <!-- ros2_context_2 port -->
                        </udpv4>
                    </locator>
                </metatrafficUnicastLocatorList>
            </builtin>
        </rtps>
    </participant>
</profiles>
```

After comparing the two files, it is worth noting that the network address that the `ros2_context_1` container is trying to connect (as a initial peer) is now the listening network address in the `ros2_context_2` container (as a meta-traffic unicast), and vice versa.

Make sure the XML profile has been configured in both docker containers:

```
export FASTRTPS_DEFAULT_PROFILES_FILE=<path_to_xml_file>
```

Run the application example again in both Vulcanexus docker containers:

- `ros2_context_1` container (terminal 1):
  ```
  ros2 run demo_nodes_cpp talker
  ```

4.10. Vulcanexus Core Tutorials 1491
• ros2_context_2 container (terminal 2):

```
ros2 run demo_nodes_cpp listener
```

The listener would receive the messages sent by the talker.

**Change mutable QoS through get native API**

- Background
- Prerequisites
- Explaining the source code
- Configuration of initial QoS
- Build
- Run
- Change mutable QoS via command line

**Background**

Fast DDS over Vulcanexus offers the possibility of not only configuring the QoS policy when creating ROS nodes, but also to modify the mutable ones after the node/publisher/subscription creation. The QoS that allow their modification after entity creation, are called mutable QoS. Please refer to Fast DDS QoS Policy documentation to get a list of all supported QoS and whether they are mutable.

*Modifying Ownership and Ownership Strength QoS Policy* explained the use of the Ownership and Ownership Strength QoS (see Ownership QoS Policy) and how to configure them within the ROS 2 talker/listener demo.

This tutorial will show how to change Ownership Strength QoS in runtime, after all nodes have been already deployed. This will be done through the interaction of three nodes (one subscriber and two publishers): After creation, the subscriber will only be receiving data from the publisher with the largest ownership strength (corresponding to exclusive ownership QoS).

![Diagram](image)

After that, the ownership strength of the other publisher will be changed to become larger than that of the first one, thus making the subscriber to start showing the data of the latter publisher.
This will be done creating a custom package, following similar steps as in Monitoring for parameter changes (C++) to be able to change a node’s parameter, and respond to that change by changing the Partition QoS of the publisher.

The ROS 2 middleware layer (see Different ROS 2 DDS/RTPS vendors) provides APIs to get handles to the objects of the inner DDS implementation, which is needed to be able to change the mutable Qos. Thus, this tutorial will also show how to use that powerful tool. For another demo on how to access inner RMW entities, see demo_nodes_cpp_native.

Prerequisites

The first prerequisite is to have Vulcanexus Iron installed (see Linux binary installation or Linux installation from sources).

Also, before starting this tutorial, user should be familiar with creating a workspace and creating a package, as well as familiar with parameters and their function in a ROS 2 system. The recommendation is to first complete the following tutorials:

- Modifying Ownership and Ownership Strength QoS Policy
- Writing a simple publisher and subscriber (C++)
- Understanding parameters
- Using parameters in a class (C++)
- Monitoring for parameter changes (C++)

This tutorial focuses on the explanations regarding mutable QoS change in runtime and regarding access to inner objects of DDS middleware implementation, and for that reason, not all the code is going to be explained, as it is already explained in the aforementioned tutorials. Create a clean workspace and download the Vulcanexus - Change Mutable QoS Through get_native API project:

```
# Create directory structure
mkdir ~/vulcanexus_ws
mkdir ~/vulcanexus_ws/src
mkdir ~/vulcanexus_ws/src/vulcanexus_change_mutable_qos
mkdir ~/vulcanexus_ws/src/vulcanexus_change_mutable_qos/src

# Download project source code
cd ~/vulcanexus_ws/src/vulcanexus_change_mutable_qos
wget -O CMakeLists.txt https://raw.githubusercontent.com/eProsima/vulcanexus/iron/docs/resources/tutorials/core/qos/mutable/vulcanexus_change_mutable_qos/CMakeLists.txt
```

(continues on next page)
wget -O package.xml https://raw.githubusercontent.com/eProsima/vulcanexus/iron/docs/resources/tutorials/core/qos/mutable/vulcanexus_change_mutable_qos/package.xml

cd ~/vulcanexus_ws/src/vulcanexus_change_mutable_qos/src
wget -O change_mutable_qos_publisher.cpp https://raw.githubusercontent.com/eProsima/vulcanexus/iron/docs/resources/tutorials/core/qos/mutable/vulcanexus_change_mutable_qos/src/change_mutable_qos_publisher.cpp

# Download profile config files for Fast DDS participants
wget -O large_ownership_strength.xml https://raw.githubusercontent.com/eProsima/vulcanexus/iron/docs/resources/tutorials/core/qos/mutable/vulcanexus_change_mutable_qos/src/large_ownership_strength.xml
wget -O small_ownership_strength.xml https://raw.githubusercontent.com/eProsima/vulcanexus/iron/docs/resources/tutorials/core/qos/mutable/vulcanexus_change_mutable_qos/src/small_ownership_strength.xml
wget -O subscriber_exclusive_ownership.xml https://raw.githubusercontent.com/eProsima/vulcanexus/iron/docs/resources/tutorials/core/qos/mutable/vulcanexus_change_mutable_qos/src/subscriber_exclusive_ownership.xml

The resulting directory structure should be:

```
~/vulcanexus_ws/
  src
    vulcanexus_change_mutable_qos
      CMakeLists.txt
      package.xml
      src
        change_mutable_qos_publisher.cpp
        large_ownership_strength.xml
        small_ownership_strength.xml
        subscriber_exclusive_ownership.xml
```

**Explaining the source code**

In the case of the Subscriber, this tutorial only needs a minimal subscriber listening on the topic `/chatter`. For convenience, the `listener` node from `demo_nodes_cpp` package will be used, as it is just a minimal subscriber listening for the aforementioned topic. For more information on how to create a minimal subscriber, *Writing a simple publisher and subscriber (C++)* tutorial shows how to write one.

In the case of the Publishers, the package is using only one executable, which takes an argument to assign the name of the Node. Here not all the code is going to be explained, as the referred tutorials of the prerequisites section explain big part of it. For instance, the `/chatter` temporized publisher is explained in the *Writing a simple publisher and subscriber (C++)*, and the mechanism to respond by means of a user callback to a change in a node’s parameter is explained in *Monitoring for parameter changes (C++)*.

The `demo_nodes_cpp_native` shows how to access inner RMW and Fast DDS entities, although it is not actually explained. In this tutorial, that same mechanism is used. In the private section of the `Node_ChangeMutableQoS_Publisher` class, the pointers to the native handlers are declared:

```cpp
// Pointers to RMW and Fast DDS inner object handles
rcl_publisher_t * rcl_publisher_;
```
In the constructor, the pointers are populated by calling the APIs provided by the rmw and rmw_fastrtps_cpp, until obtaining the `eprosima::fastdds::dds::DataWriter` handle:

```c
// Access RMW and Fast DDS inner object handles
rcl_publisher_ = publisher_->get_publisher_handle().get();
rmw_publisher_ = rcl_publisher_get_rmw_handle(rcl_publisher_);
data_writer_ = rmw_fastrtps_cpp::get_datawriter(rmw_publisher_);
```

When the `Publisher_X_ownership_strength` is updated (for instance, via command line using `ros2 param set` command), the parameter callback is raised, and the `eprosima::fastdds::dds::DataWriter` handle is used to update its ownership strength. Below, a snippet of code from the constructor of the node, where the parameter is declared, the subscription to its changes is registered, and the callback to be run on the parameter change event is defined.

```c
// Declare parameter
std::string parameter_name = node_name_prefix + "_ownership_strength";
this->declare_parameter(parameter_name, 100); // This is the parameter initialization. 100 is only to state it is int type

// Create a parameter subscriber that can be used to monitor parameter changes
param_subscriber_ = std::make_shared<rclcpp::ParameterEventHandler>(this);

// Set a callback for this node's integer parameter, "Publisher_X_ownership_strength"
auto callback = [this](const rclcpp::Parameter & p) {
  RCLCPP_INFO(this->get_logger(), "Callback: Received an update to parameter \"%s\" of type %s: \n  \"%ld\"",
  p.get_name().c_str(),
  p.get_type_name().c_str(),
  p.as_int());

  eprosima::fastdds::dds::DataWriterQos dw_qos;
  data_writer_->get_qos(dw_qos);

  dw_qos.ownership_strength().value = p.as_int();
  data_writer_->set_qos(dw_qos);
};
```

In this case, as in the current version of Fast DDS the built-in statistics are enabled by default (see `DomainParticipantQos`), it is needed to retrieve the internal QoS by means of `::get_qos()`, then perform the modifications and update the QoS by means of `::set_qos()`: The value of the ownership strength is set from the value of the updated parameter.
Configuration of initial QoS

Ownership Strength Policy is mutable, but Ownership Policy is not, so configuring EXCLUSIVE_OWNERSHIP_POLICY to all participants before running the ROS nodes is needed. To do that, inside the package, there are three xml files. Each one of them defines a profile for a publisher with a “large” ownership strength, another with a “small” ownership strength and a subscriber (that does not need an ownership strength definition). For the three of them, exclusive ownership is defined.

large_ownership_strength.xml

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<dds>
  <profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <data_writer profile_name="/chatter">
      <qos>
        <ownership>
          <kind>EXCLUSIVE</kind>
        </ownership>
        <ownershipStrength>
          <value>10</value>
        </ownershipStrength>
      </qos>
    </data_writer>
  </profiles>
</dds>
```

small_ownership_strength.xml

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<dds>
  <profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <data_writer profile_name="/chatter">
      <qos>
        <ownership>
          <kind>EXCLUSIVE</kind>
        </ownership>
        <ownershipStrength>
          <value>1</value>
        </ownershipStrength>
      </qos>
    </data_writer>
  </profiles>
</dds>
```

subscriber_exclusive_ownership.xml

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<dds>
  <profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <data_reader profile_name="/chatter">
      <qos>
        <ownership>
          <kind>EXCLUSIVE</kind>
        </ownership>
      </qos>
    </data_reader>
  </profiles>
</dds>
```
Build

Now the package is ready to be built. Change the directory to the workspace folder and build using colcon:

```
source /opt/vulcanexus/iron/setup.bash
cd ~/vulcanexus_ws
colcon build
```

Run

Open three terminals in the workspace folder. On each of them, Vulcanexus installation, as well as the package installation is needed. Then, export the `FASTRTPS_DEFAULT_PROFILES_FILE` environment variable to point out to the corresponding profiles file and run the node.

- In the first terminal, run the `listener` node from the `demo_nodes_cpp`, configured with the `subscriber_exclusive_ownership.xml` file.

- Then, in another terminal, run the first publisher, configured also with the `large_ownership_strength.xml` file. This Publisher will then be configured with ownership strength value of 10. At this point both nodes should be communicating, and the messages from Publisher 1 should be shown in the Subscriber.

- In the third terminal, run the second publisher, configured with the `small_ownership_strength.xml` file. This Publisher will then be configured with ownership strength value of 2. This Publisher 2 starts sending messages (it can be seen that the number of the message starts from 1 while the messages from Publisher 1 are already in a higher number), and the Subscriber is still receiving messages from Publisher 1 and not from Publisher 2. This is because of the exclusive ownership.

The code to execute in each terminal can be found in the tabs below:

First terminal

```
source /opt/vulcanexus/iron/setup.bash
cd ~/vulcanexus_ws
# Using profile to set exclusive ownership
export FASTRTPS_DEFAULT_PROFILES_FILE=./install/vulcanexus_change_mutable_qos/profiles/...
->subscriber_exclusive_ownership.xml
ros2 run demo_nodes_cpp listener "# Run minimal subscriber"
```

Second terminal

```
source /opt/vulcanexus/iron/setup.bash
cd ~/vulcanexus_ws
source install/setup.bash
# Using profile to set large strenght value
export FASTRTPS_DEFAULT_PROFILES_FILE=./install/vulcanexus_change_mutable_qos/profiles/...
->large_ownership_strength.xml
ros2 run vulcanexus_change_mutable_qos change_mutable_qos_publisher Publisher_1 "# Run Publisher 1"
```

(continues on next page)
Third terminal

```bash
source /opt/vulcanexus/iron/setup.bash
cd ~/vulcanexus_ws
source install/setup.bash
`# Using profile to set small strenght value'
export FASTRTPS_DEFAULT_PROFILES_FILE=./install/vulcanexus_change_mutable_qos/profiles/
...small_ownership_strength.xml
ros2 run vulcanexus_change_mutable_qos change_mutable_qos_publisher Publisher_2  `# Run Publisher 2`
```

Publisher 1 has higher ownership strength than Publisher 2.

**Change mutable QoS via command line**

In this last section, the `param set` command will be used to change the value of the node’s parameter created earlier. The parameter change will cause the parameter-changed callback to be called, which then results in a change in the ownership strength. In another terminal, try the following code:

```bash
source /opt/vulcanexus/iron/setup.bash
cd ~/vulcanexus_ws
source install/setup.bash
ros2 param set /Publisher_2_change_mutable_qos Publisher_2_ownership_strength 50
```

With that execution, the ownership strength of the Publisher 2 has changed to become larger than that of the Publisher 1. **Now the Subscriber should be receiving the messages from the Publisher 2 and not from the Publisher 1.**

**4.10.3 Vulcanexus Deployment Tutorials**

Dealing with advanced network topologies can be challenging. Vulcanexus provides out-of-the-box functionality by fully customizing its underlying Fast-DDS middleware implementation features.

In this series of tutorials, different Vulcanexus approaches to address complex deployment scenarios are explained. For tutorials related to cloud or cloud-edge deployments, please refer [Vulcanexus Cloud Tutorials](#).

**Customizing Network Transports**

- **Background**
- **Prerequisites**
- **XML Configuration**
- **Run the example**
Background

Note: This documentation assumes basic knowledge of UDP/TCP/IP concepts. However, it is possible to follow it without this knowledge.

Vulcanexus applications, by default, enable two different transports: a UDPv4 transport for inter-host communication, and a shared memory transport (SHM) for inter-process communications with other applications running on the same host. Although the default-enabled transports may be suited for most use-cases, they do not intend to cover the entire spectrum of Vulcanexus deployments. Among the many use-cases that may benefit from a custom transport layer, there are:

1. Applications communicating over IPv6 networks.
2. NAT traversing communications over TCP (see Vulcanexus Cloud).
3. No-network deployments.
4. Communications enabling TLS.
5. Limiting Vulcanexus traffic to a subset of the host’s network interfaces.

For these reasons, Vulcanexus allows for the configuration of network and SHM transports leveraging Fast DDS capabilities to configure the middleware’s transport layer. This is achieved by defining the appropriate transport description in a Fast DDS XML configuration file.

This tutorial showcases a configuration in which the Vulcanexus communication is limited to a single network interface (localhost) while avoiding Vulcanexus-sent UDP datagrams to be fragmented at the IP level. This is achieved by setting an interface whitelist, and by limiting the size of the Vulcanexus RTPS (the underlying DDS wire-protocol) datagrams so that complete UDP packets fit into the system’s MTU (typically 1500 B).

The latter is specially beneficial to achieve reliable communication over DDS (through a UDPv4 network transport) when communicating over a lossy network. This is because, although the underlying RTPS protocol can be configured as reliable, UDPv4 is a best-effort protocol over IP (another best-effort protocol), and as such is susceptible to network losses. If the RTPS datagrams (which become payloads of UDPv4 datagrams) have the maximum allowed size for a UDP payload (~65 kiB), then the resulting packet would need to be fragmented into 44 fragments (assuming an 1500 B MTU). In this scenario, losing one of those 44 fragments entails losing the entire UDP datagram, and consequently the entire RTPS datagram. If the network’s packet drop rate is 1/44 or higher, no UDP datagram will ever be reconstructed, resulting in no RTPS datagram ever received, and therefore the RTPS reliability cannot succeed at all. By fragmenting the DDS data into several self-contained, smaller-than-the-MTU UDP datagrams, 43 out of every 44 RTPS fragments will be received at first try (in the considered scenario), and the RTPS reliability will be able to retransmit the missing ones successfully.

Prerequisites

Please, remember to source the environment in every terminal used during this tutorial.

```
source /opt/vulcanexus/iron/setup.bash
```

Install the ROS 2 image demo package (administrative privileges may be required):

```
apt update && apt install -y ros-iron-image-tools
```
XML Configuration

Save the following XML configuration file at the desired location, which will be referred as `<path_to_xml_config_file>` from here onwards.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<dds>
  <profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTTPS_Profiles" >
    <!-- Create a custom transport descriptor -->
    <transport_descriptor>
      <transport_id>custom_udpv4_transport</transport_id>
      <type>UDPv4</type>
      <!-- Limit the RTPS datagrams to 1400 B -->
      <maxMessageSize>1400</maxMessageSize>
      <!-- Limit communication to localhost -->
      <interfaceWhiteList>
        <address>127.0.0.1</address>
      </interfaceWhiteList>
    </transport_descriptor>
  </transport_descriptors>
  <participant profile_name="participant_profile" is_default_profile="true">
    <rtps>
      <!-- Disable builtin transports -->
      <useBuiltinTransports>false</useBuiltinTransports>
      <!-- Enable custom transport -->
      <userTransports>
        <transport_id>custom_udpv4_transport</transport_id>
      </userTransports>
    </rtps>
  </participant>
</profiles>
</dds>
```

Run the example

**Note:** To run this example using a Vulcanexus docker container, GUI capabilities are required (see Docker installation).

This tutorial leverages the ros-iron-image-tools package to demonstrate that the aforementioned XML configuration indeed achieves communication over localhost with UDPv4 datagrams smaller than the standard 1500 B MTU. To run the tutorial, two shells are required:

**Shell 1 (Subscription)**

```
source /opt/vulcanexus/iron/setup.bash
export FASTRTPS_DEFAULT_PROFILES_FILE=<path_to_xml_config_file>
ros2 run image_tools showimage
```

**Shell 2 (Publisher)**
source /opt/vulcanexus/iron/setup.bash
export FASTRTPS_DEFAULT_PROFILES_FILE=<path_to_xml_config_file>
ros2 run image_tools cam2image --ros-args -p burger_mode:=True

Optionally, as shown in the following video, it is possible to use Wireshark to sniff the RTPS traffic across the localhost interface to corroborate the size of the UDP datagrams containing RTPS fragments:

Connecting Nodes over an External Network

- Background
- Prerequisites
- Understanding External Locators
- Enable External Locators via XML configuration files
- Run the example

Background

Many robotic applications involve complex network topologies (mesh, nested, etc.) in which nodes, processes or entities from one host may need to interact with both, nodes from outside and inside its network.

Consider the following nested network scenario: one talker node inside one host’s private LAN sub-network needs to communicate with two listener nodes; the first one belonging to the same LAN as the talker, and the second one deployed within an external host with its own private sub-network. Also consider both hosts externally connected to the same network. The following diagram depicts the aforementioned scheme.

This example case consists on two hosts, the first one maintaining two docker containers running the well-known ROS 2 talker-listener example nodes, and a second one with a ROS 2 listener inside another container. In Vulcanexus, the communication among the different nodes can be achieved by means of Fast DDS’ External Locators feature defined in WireProtocolConfigQos.

Prerequisites

For accomplishing this tutorial, two available hosts with Docker and a Vulcanexus image are required. Please refer to the installation steps detailed in Docker installation. In addition, host’s ports 11200, 11201 need to be available (these ports have been selected as an example for this tutorial, but it is up to the user to make a different choice).

Fast DDS domain participants will require to announce themselves into their host’s external network(s). Net-tools, Network Manager or similar packages need to be installed in the system in order to retrieve the corresponding IPs addresses.
Understanding External Locators

External locators should be seen as an extra feature over the default communication mechanisms (unicast, multicast). In that sense, it is possible not just to discover peers via multicast within the same LAN, but also peers in remote sub-networks over a shared external network at some higher level (WAN, WLAN, etc.). It is for this reason that External Locators relies on the concept of levels of externality, which effectively map to the different nesting levels of the network setup, resulting in a sequence of external IP addresses exposed by the different nested sub-network interfaces.

As described in the next section, it is possible to configure external IP locators with an associated externality index, cost, and sub-network mask within the DomainParticipant configuration.

Enable External Locators via XML configuration files

In order to define the desired External Locators configuration, an XML profile needs to be provided (see Fast DDS XML profiles). External Locators announcement for the different Communication phases: participant and endpoint discovery phase (metatraffic, initial peers tags), and user data communication phase (user traffic tag), should be defined.

Following with the example above, two XML configuration profiles should be provided. The power of External Locators is the ability to connect to nodes within external networks while still being discovered in the local network. Due to this reasoning, multicast discovery will be used for the second container on the first host (consequently, there is no need to provide any additional XML profile). The two resultant XML configurations are detailed below:

Note: Note that the container network itself does not create another level of externality in this case, as it is bridged with the host network.

HOST 1
CONTAINER 1

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<dds>
  <profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="container0" is_default_profile="true">
      <rtps>
        <!-- External locators for user traffic -->
        <default_external_unicast_locators>
          <udpv4 externality="1" cost="0" mask="24">
            <!-- Host 1 external IP -->
            <address>192.168.1.40</address>
            <port>11201</port>
          </udpv4>
        </default_external_unicast_locators>
        <builtin>
          <!-- External locators for discovery traffic -->
          <metattraffic_external_unicast_locators>
            <udpv4 externality="1" cost="0" mask="24">
              <!-- Host 1 external IP -->
              <address>192.168.1.40</address>
              <port>11200</port>
            </udpv4>
          </metattraffic_external_unicast_locators>
        </builtin>
      </rtps>
    </participant>
  </profiles>
</dds>
```

(continues on next page)
HOST 2
CONTAINER 1

```xml
<initialPeersList>
  <!-- Container 2 peer-->
  <locator>
    <udpv4>
      <!-- Host 2 external IP -->
      <address>192.168.1.56</address>
      <port>11200</port>
    </udpv4>
  </locator>

  <!-- Local network DDS default multicast to discover other participants in the same LAN, using External Locators, or not-->
  <locator>
    <udpv4>
      <address>239.255.0.1</address>
      <port>7400</port>
    </udpv4>
  </locator>
</initialPeersList>
```

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<dds>
  <profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="container0" is_default_profile="true">
      <rtps>
        <!-- External locators for user traffic -->
        <default_external_unicast_locators>
          <udpv4 externality="1" cost="0" mask="24">
            <!-- Host 2 external IP -->
            <address>192.168.1.56</address>
            <port>11201</port>
          </udpv4>
        </default_external_unicast_locators>
        <builtin>
          <!-- External locators for discovery traffic -->
          <metatraffic_external_unicast_locators>
            <udpv4 externality="1" cost="0" mask="24">
              <!-- Host 2 external IP -->
              <address>192.168.1.56</address>
              <port>11200</port>
            </udpv4>
          </metatraffic_external_unicast_locators>
        </builtin>
      </rtps>
    </participant>
  </profiles>
</dds>
```
Run the example

This section provides with step-by-step instructions for setting up the example scenario described in previous sections. On both hosts, open a shell and run:

HOST 1
TERMINAL 1

docker run --rm -it \
    '# Cleanup, interactive terminal' \ 
     -p 11200-11201:7412-7413/udp \
     ubuntu-vulcanexus:iron-desktop \
     '# Image name'

TERMINAL 2

docker run --rm -it \
    '# Cleanup, interactive terminal' \ 
     ubuntu-vulcanexus:iron-desktop \
     '# Image name'

HOST 2
TERMINAL 1

docker run --rm -it \
    '# Cleanup, interactive terminal' \ 
     -p 11200-11201:7412-7413/udp \
     ubuntu-vulcanexus:iron-desktop \
     '# Image name'
Note: It is important to specify the port mapping argument so as to expose docker internal ports to the host. See Docker Networking for further information.

The next step is the creation of the XML profiles. Inside each one of the three containers, create a Profiles.xml file and paste the contents of the corresponding XML profile configuration, according to the previous section.

Finally, export the environment variable pointing to the Profiles.xml file, source Vulcanexus environment and run the ros2 example nodes.

HOST 1
CONTAINER 1

```bash
source /vulcanexus_entrypoint.sh
export FASTRTPS_DEFAULT_PROFILES_FILE=/Profiles.xml  # Or the Profiles.xml file location
ros2 run demo_nodes_cpp talker
```

CONTAINER 2

```bash
source /vulcanexus_entrypoint.sh
ros2 run demo_nodes_cpp listener
```

HOST 2
CONTAINER 1

```bash
source /vulcanexus_entrypoint.sh
export FASTRTPS_DEFAULT_PROFILES_FILE=/Profiles.xml  # Or the Profiles.xml file location
ros2 run demo_nodes_cpp listener
```

At this point, nodes should be communicating with each other as expected. A message *Hello World: [count]* should start printing in the talker’s container terminal while both listeners keep receiving it, in their respective container consoles, as follows:

**Fast DDS - Vulcanexus Topic Intercommunication**

This tutorial presents a step-by-step demonstration on how to intercommunicate Vulcanexus applications with native Fast DDS applications.

- Background
- Prerequisites
- IDL type definition
- Vulcanexus Application
  - Vulcanexus Application - Type generation
  - Vulcanexus Application - C++
- Fast DDS Application
  - Fast DDS Application - Type generation
  - Fast DDS Application - C++
- Run the demo
Background

Being Fast DDS the default Vulcanexus middleware enables the possibility of intercommunicating Vulcanexus applications with native Fast DDS ones. This is of special interest when integrating pre-existing systems with each other, such as interfacing with a third-party software which exposes a DDS API. Since both Fast DDS’ and Vulcanexus’ backbone is DDS, it is possible to intercommunicate full-blown systems running a Vulcanexus stack with smaller systems for which Vulcanexus is either unnecessary or unfit, such as more constrained environments or applications that would not require any Vulcanexus functionality other than the middleware.

Prerequisites

For convenience, this tutorial is built and run within a Docker environment, although Docker is not required. The tutorial focuses on the explanations regarding message type and topic name compatibilities rather than given an in depth explanation about the code used. Create a clean workspace and download the Vulcanexus - Fast DDS Topic Intercommunication project:

```
# Create directory structure
mkdir ~/vulcanexus_dds_ws
cd ~/vulcanexus_dds_ws
mkdir fastdds_app idl vulcanexus_app
# Download project source code
wget -O CMakeLists.txt https://raw.githubusercontent.com/eProsima/vulcanexus/iron/docs/resources/tutorials/core/deployment/dds2vulcanexus/topic/CMakeLists.txt
wget -O Dockerfile https://raw.githubusercontent.com/eProsima/vulcanexus/iron/docs/resources/tutorials/core/deployment/dds2vulcanexus/topic/Dockerfile
wget -O package.xml https://raw.githubusercontent.com/eProsima/vulcanexus/iron/docs/resources/tutorials/core/deployment/dds2vulcanexus/topic/package.xml
wget -O README.md https://raw.githubusercontent.com/eProsima/vulcanexus/iron/docs/resources/tutorials/core/deployment/dds2vulcanexus/topic/README.md
wget -O vulcanexus_app/publisher.cpp https://raw.githubusercontent.com/eProsima/vulcanexus/iron/docs/resources/tutorials/core/deployment/dds2vulcanexus/topic/vulcanexus_app/publisher.cpp
```

The resulting directory structure should be:

```
~/vulcanexus_dds_ws/
    ├── CMakeLists.txt
    └── Dockerfile
```
Finally, the Docker image can be built with:

```bash
cd ~/vulcanexus_dds_ws
docker build -f Dockerfile -t dds2vulcanexus .
```

**IDL type definition**

Although the `msg` format used to be the preferred way to describe topic types in ROS 2 (just to ease the migration from ROS types), they get converted into IDL under the hood before the actual topic type related code is generated on the CMake call to `rosidl_generate_interfaces`. This means that the topic type definitions can be written as IDL files directly, allowing for a straightforward type compatibility with native DDS applications, since the standardized type definition format in DDS is in fact IDL. For a complete correspondence matrix between `msg` and IDL types (referred as DDS Types in the table), please refer to Field types.

This tutorial leverages ROS 2 capabilities of describing types in IDL to define a `HelloWorld.idl` that will be used by both the Vulcanexus and native Fast DDS applications. The `HelloWorld.idl`, and its `msg` equivalent is as follows:

```idl
module dds2vulcanexus
{
    module idl
    {
        struct HelloWorld
        {
            unsigned long index;
            string message;
        };
    };
};
```

HelloWorld.msg

```plaintext
uint32 index
string message
```

It is important to note that `rosidl_generate_interfaces` converts the simple `HelloWorld.msg` into and IDL containing the structure (which is named after the `msg` file name) within 2 nested modules, the outermost being the package name (in this case `dds2vulcanexus`), and the innermost being the name of the directory in which the file is located. Mind that in the aforementioned directory structure, the IDL file is placed within an `idl` directory, hence the name of the innermost module.

The following sections detail how to incorporate the IDL message definition into both the Vulcanexus and native Fast DDS applications, covering both the C++ and CMake sides.
Vulcanexus Application

On this tutorial, the Vulcanexus application consists on a simple publisher node which will publish messages to the HelloWorld topic once a second.

Vulcanexus Application - Type generation

Inspecting the CMakeLists.txt file downloaded in Prerequisites, the following CMake code pertains the Vulcanexus publisher:

```cmake
# Vulcanexus application
message(STATUS "Configuring Vulcanexus application...")
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(rosidl_default_generators REQUIRED)

set(type_files
    "idl/HelloWorld.idl"
)
rosidl_generate_interfaces(${PROJECT_NAME}
    ${type_files}
)

ament_export_dependencies(rosidl_default_runtime)

add_executable(vulcanexus_publisher vulcanexus_app/publisher.cpp)
ament_target_dependencies(vulcanexus_publisher rclcpp)
rosidl_get_typesupport_target(cpp_typesupport_target ${PROJECT_NAME} "rosidl_typesupport_˓
    -->cpp")
target_link_libraries(vulcanexus_publisher "${cpp_typesupport_target}"
)

install(TARGETS
    vulcanexus_publisher
    DESTINATION lib/${PROJECT_NAME}
)

ament_package()
```

In particular, the type related code is generated in:

```cmake
set(type_files
    "idl/HelloWorld.idl"
)
rosidl_generate_interfaces(${PROJECT_NAME}
    ${type_files}
)
```
Vulcanexus Application - C++

The simple Vulcanexus publisher node is as follows:

```cpp
#include <chrono>
#include <memory>
#include <string>
#include "rclcpp/qos.hpp"
#include "rclcpp/rclcpp.hpp"

using namespace std::chrono_literals;

class HelloWorldPublisher : public rclcpp::Node {
public:
  HelloWorldPublisher()
    : Node("helloworld_publisher")
  {
    sample_.index = 0;
    sample_.message = "Hello from Vulcanexus";

    publisher_ = this->create_publisher<dds2vulcanexus::idl::HelloWorld>("HelloWorld", 10);

    auto timer_callback = [this]() -> void {
      sample_.index++;
      RCLCPP_INFO(this->get_logger(), "Publishing: '%s %u'", sample_.message.c_str(), sample_.index);
      this->publisher_->publish(sample_);
    };
    timer_ = this->create_wall_timer(1s, timer_callback);

private:
```

(continues on next page)
To use the type generated from the IDL, three things are done:

1. Include the generated type header:

   ```cpp
   #include "dds2vulcanexus/idl/hello_world.hpp"
   ```

2. Create a publisher in a HelloWorld topic which uses the generated type.
   1. First, the HelloWorldPublisher Node class stores a shared pointer to the publisher:

   ```cpp
   rclcpp::Publisher<dds2vulcanexus::idl::HelloWorld>::SharedPtr publisher_
   ```

   2. Then, upon construction, it instantiates the publisher, assigning it to the shared pointer class data member:

   ```cpp
   publisher_ = this->create_publisher<dds2vulcanexus::idl::HelloWorld>(
                     "HelloWorld", 10);
   ```

3. Publish data on the topic. In this case, the HelloWorldPublisher is using a wall timer to have periodic publications:
   1. HelloWorldPublisher has a data member for reusing the sample:

   ```cpp
   dds2vulcanexus::idl::HelloWorld sample_
   ```

   2. HelloWorldPublisher, upon construction, creates said wall timer, which is used to publish data:

   ```cpp
   auto timer_callback = [this]() -> void {
     sample_.index++;
     RCLCPP_INFO(
         this->get_logger(), "Publishing: '%s %u'",
         sample_.message.c_str(), sample_.index);
     this->publisher_->publish(sample_);
   };
   timer_ = this->create_wall_timer(1s, timer_callback);
   ```
Fast DDS Application

Much like the Vulcanexus application, the native Fast DDS one consists on two parts:

1. The generated type related code (a.k.a type support).
2. The application code

Fast DDS Application - Type generation

In the case of Fast DDS, the type support is generated from the HelloWorld.idl file using Fast DDS-Gen.

In this tutorial, the Fast DDS type support is generated within the CMakeLists.txt file for the sake of completion and simplicity, but it can be generated as a pre-build step instead. Inspecting the CMakeLists.txt file downloaded in Pre-requisites, the following CMake code pertains the native Fast DDS subscriber:

```cmake
### Fast DDS application
find_package(fastcdr REQUIRED)
find_package(fastrtps REQUIRED)
find_program(FASTDDSGEN fastddsgen)
set(GENERATED_TYPE_SUPPORT_FILES
  ${CMAKE_SOURCE_DIR}/fastdds_app/HelloWorld.h
  ${CMAKE_SOURCE_DIR}/fastdds_app/HelloWorld.cxx
  ${CMAKE_SOURCE_DIR}/fastdds_app/HelloWorldPubSubTypes.h
  ${CMAKE_SOURCE_DIR}/fastdds_app/HelloWorldPubSubTypes.cxx)
add_custom_command(
  OUTPUT ${GENERATED_TYPE_SUPPORT_FILES}
  COMMAND ${FASTDDSGEN}
  -replace
  -typeros2
  -d ${CMAKE_SOURCE_DIR}/fastdds_app
  idl/HelloWorld.idl
  DEPENDS ${CMAKE_SOURCE_DIR}/idl/HelloWorld.idl
  COMMENT "Fast DDS type support generation" VERBATIM )
add_executable(fastdds_subscriber
  ${CMAKE_SOURCE_DIR}/fastdds_app/subscriber.cpp
  ${GENERATED_TYPE_SUPPORT_FILES})
target_link_libraries(fastdds_subscriber fastrtps fastcdr)
isolate(TARGETS
  fastdds_subscriber
  DESTINATION lib/${PROJECT_NAME})
```

In particular, the type generation related code is:

4.10. Vulcanexus Core Tutorials
```cpp
find_program(FASTDDSGEN fastddsgen)
set(
   GENERATED_TYPE_SUPPORT_FILES
   ${CMAKE_SOURCE_DIR}/fastdds_app/HelloWorld.h
   ${CMAKE_SOURCE_DIR}/fastdds_app/HelloWorld.cxx
   ${CMAKE_SOURCE_DIR}/fastdds_app/HelloWorldPubSubTypes.h
   ${CMAKE_SOURCE_DIR}/fastdds_app/HelloWorldPubSubTypes.cxx
)
add_custom_command(
   OUTPUT ${GENERATED_TYPE_SUPPORT_FILES}
   COMMAND ${FASTDDSGEN}
   -replace
   -typeros2
   -d ${CMAKE_SOURCE_DIR}/fastdds_app
   ${CMAKE_SOURCE_DIR}/idl/HelloWorld.idl
   DEPENDS ${CMAKE_SOURCE_DIR}/idl/HelloWorld.idl
   COMMENT "Fast DDS type support generation" VERBATIM
)
```

The call to *Fast DDS-Gen* within `add_custom_command` will generate the type support in the `fastdds_app` directory, leaving the file names in a convenient `GENERATED_TYPE_SUPPORT_FILES` CMake variable that is later used to add the source files to the executable. It is important to note the *Fast DDS-Gen* is called with the `-typeros2` flag, so it generates ROS 2 compatible type names.

### Fast DDS Application - C++

The native Fast DDS subscriber is as follows:

```cpp
#include <fastdds/dds/domain/DomainParticipant.hpp>
#include <fastdds/dds/domain/DomainParticipantFactory.hpp>
#include <fastdds/dds/domain/qos/DomainParticipantQos.hpp>
#include <fastdds/dds/subscriber/DataReader.hpp>
#include <fastdds/dds/subscriber/DataReaderListener.hpp>
#include <fastdds/dds/subscriber/InstanceState.hpp>

#include <condition_variable>
#include <csignal>
#include <mutex>

// Copyright 2022 Proyectos y Sistemas de Mantenimiento SL (eProsima).
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
// http://www.apache.org/licenses/LICENSE-2.0
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.
```

(continues on next page)
#include <fastdds/dds/subscriber/qos/DataReaderQos.hpp>
#include <fastdds/dds/subscriber/qos/SubscriberQos.hpp>
#include <fastdds/dds/subscriber/Subscriber.hpp>
#include <fastrtps/subscriber/SampleInfo.h>
#include <fastrtps/types/TypesBase.h>

#include "HelloWorldPubSubTypes.h"

class HelloWorldSubscriber : public eprosima::fastdds::dds::DataReaderListener
{
public:

    HelloWorldSubscriber()
        : participant_(nullptr), subscriber_(nullptr), topic_(nullptr), reader_(nullptr), type_(new dds2vulcanexus::idl::HelloWorldPubSubType())
    {
    }

    virtual ~HelloWorldSubscriber()
    {
        if (reader_ != nullptr)
        {
            subscriber_->delete_datareader(reader_);
        }
        if (topic_ != nullptr)
        {
            participant_->delete_topic(topic_);
        }
        if (subscriber_ != nullptr)
        {
            participant_->delete_subscriber(subscriber_);
        }
        eprosima::fastdds::dds::DomainParticipantFactory::get_instance()->delete_participant(participant_);
    }

    bool init()
    {
        auto factory = eprosima::fastdds::dds::DomainParticipantFactory::get_instance();
        participant_ = factory->create_participant(0, eprosima::fastdds::dds::PARTICIPANT_QOS_DEFAULT);
        if (participant_ == nullptr)
        {
            return false;
        }
        participant_->register_type(type_);
        topic_ = participant_->create_topic("rt/HelloWorld",
    }

(continues on next page)
type_.get_type_name(), eprosima::fastdds::dds::TOPIC_QOS_DEFAULT);
    if (topic_ == nullptr)
    {
      return false;
    }

    subscriber_ = participant_->create_subscriber(eprosima::fastdds::dds::SUBSCRIBER_QOS_DEFAULT);
    if (subscriber_ == nullptr)
    {
      return false;
    }

    reader_ = subscriber_->create_datareader(topic_,
    eprosima::fastdds::dds::DATAREADER_QOS_DEFAULT, this);
    if (reader_ == nullptr)
    {
      return false;
    }

    return true;
  }

  void on_data_available(
    eprosima::fastdds::dds::DataReader* reader)
  override
  {
    eprosima::fastdds::dds::SampleInfo info;
    if (reader->take_next_sample(&sample_, &info) ==
    eprosima::fastrtps::types::ReturnCode_t::RETCODE_OK)
    {
      if (info.instance_state == eprosima::fastdds::dds::InstanceStateKind::ALIVE_INSTANCE_STATE)
      {
        std::cout << "Receiving: " << sample_.message() << " " << sample_.index() << "" << std::endl;
      }
    }
  }

private:

  eprosima::fastdds::dds::DomainParticipant* participant_;  
  eprosima::fastdds::dds::Subscriber* subscriber_; 
  eprosima::fastdds::dds::Topic* topic_; 
  eprosima::fastdds::dds::DataReader* reader_; 
  eprosima::fastdds::dds::TypeSupport type_; 
  dds2vulcanexus::idl::HelloWorld sample_; 
};

std::condition_variable cv;
std::mutex mtx;
std::atomic_bool running;

void signal_handler_callback(int signum)
{
    std::cout << std::endl << "Caught signal " << signum << "; closing down..." << std::endl;
    running.store(false);
    cv.notify_one();
}

int main()
{
    signal(SIGINT, signal_handler_callback);

    HelloWorldSubscriber subscriber;
    running.store(subscriber.init());

    std::unique_lock<std::mutex> lck(mtx);
    cv.wait(lck, [&]() {
        return !running.load();
    });
}

There are several things to unpack in this application:

1. HelloWorldSubscriber holds both a reference to the type support, to create the topic, and a HelloWorld sample instance for reusing it upon reception.

    eprosima::fastdds::dds::TypeSupport type_
    dds2vulcanexus::idl::HelloWorld sample_

   1. The type support is instantiated upon construction:

    HelloWorldSubscriber()
        : participant_(nullptr)
        , subscriber_(nullptr)
        , topic_(nullptr)
        , reader_(nullptr)
        , type_(new dds2vulcanexus::idl::HelloWorldPubSubType())

    2. Then, it is registered in the DomainParticipant for further use:

        participant_−>register_type(type_);

2. The topic is created with name rt/HelloWorld. Mind that this topic name is different from the one set in the Vulcanexus publisher (HelloWorld). This is because Vulcanexus appends rt/ to the topic name passed when creating a Publisher or Subscription, where rt stands for ROS Topic, as services and actions have different prefixes (please refer to ROS 2 design documentation regarding Topic and Service name mapping to DDS). Another important detail is the type name, which in this example is extracted from the type support directly, as the type is generated with ROS 2 naming compatibility (see Fast DDS Application - Type generation).
3. A **DataReader** is created in the topic, setting the very **HelloWorldSubscriber** as listener, since it inherits from **DataReaderListener**, overriding the **on_data_available** callback:

```cpp
subscriber_ = participant_->create_subscriber(eprosima::fastdds::dds::SUBSCRIBER_QOS_DEFAULT);
if (subscriber_ == nullptr)
{
    return false;
}
reader_ = subscriber_->create_datareader(topic_,
    eprosima::fastdds::dds::DATAREADER_QOS_DEFAULT, this);
if (reader_ == nullptr)
{
    return false;
}
```

4. Finally, when a new sample arrives, Fast DDS calls the implementation of **on_data_available**, which print the data to the **STDOUT**:

```cpp
void on_data_available(
    eprosima::fastdds::dds::DataReader* reader) override
{
    eprosima::fastdds::dds::SampleInfo info;
    if (reader->take_next_sample(&sample_, &info) ==
        eprosima::fastrtps::types::ReturnCode_t::RETCODE_OK)
    {
        if (info.instance_state ==
            eprosima::fastdds::dds::InstanceStateKind::ALIVE_INSTANCE_STATE)
        {
            std::cout << "Receiving: " << sample_.message() << " " <<
                sample_.index() << " " << std::endl;
        }
    }
}
```
Run the demo

Once the Docker image is built, running the demo simply require two terminals. The image can be run in each of them with:

```
docker run -it --rm dds2vulcanexus
```

Then, run the publisher on one of the containers and the subscriber on the other:

Publisher

```
vulcanexus_publisher
```

Subscriber

```
fastdds_subscriber
```

Static Discovery

- Background
- Prerequisites
- Overview
- Standard discovery
- Static EDP discovery
- Full-fledged Static EDP discovery

Background

OMG DDS Interoperability specification defines the basic discovery mechanism that allows distributed systems to automatically find and match Endpoints sharing the same Topic. This mechanism is performed in two phases as explained in Fast DDS documentation:

- **Participant Discovery Phase (PDP)**: DDS DomainParticipants discover and acknowledge the existence of remote DomainParticipants. Usually done sending periodic announcements to well-known multicast addresses and ports defined in the specification.

- **Endpoint Discovery Phase (EDP)**: each DDS DomainParticipant send information about its own endpoints (DataWriters and DataReaders) through the communication channels established in the previous phase.

Standard discovery has its caveats, like the required multicast traffic being sent periodically, that can saturate the network bandwidth in constraint network architectures with a large number of DDS DomainParticipants. Vulcanexus leverages latest eProsima Fast DDS release to improve ROS 2 middleware layer. Being aware of the standard discovery mechanism caveats, eProsima Fast DDS provides alternative discovery mechanisms:

- **Fast DDS Discovery Server** reduces the discovery metatrace using a centralized client-server discovery mechanism instead of a distributed one. More information can be found in Using Fast DDS Discovery Server as discovery protocol tutorial. This schema reduces discovery metatrace from both PDP and EDP phases because the information is sent exclusively to the DomainParticipants in a need-to-know basis.

- Setting an initial peers list and disabling multicast also decreases the discovery traffic for the PDP phase, sending each DomainParticipant the announcement to the locators where the announcement is expected.
Fast DDS Static Discovery removes completely any EDP discovery metatraffic by configuring the endpoints, topics and data types beforehand. This tutorial deals on how to use this discovery mechanism within Vulcanexus. More information about Static EDP discovery can be found in Fast DDS documentation.

Prerequisites

This tutorial uses ROS 2 demo_nodes_cpp package which is provided with Vulcanexus Desktop distribution. If another Vulcanexus distribution is used, please ensure that this package and its dependencies are installed.

Wireshark is also used to capture and analyze the network traffic.

Please, remember to source the environment in every terminal used during this tutorial.

```
source /opt/vulcanexus/iron/setup.bash
```

Overview

This tutorial will use ROS 2 demo_nodes_cpp talker and listener applications to showcase the different configurations:

1. Talker-listener communication using **Standard discovery**. Shared memory transport is explicitly disabled to ensure EDP discovery traffic can be captured.
2. Talker-listener communication using **Static EDP discovery**.
3. Talker-listener full-fledged communication.

Standard discovery

eProsimas Fast DDS enables by default a Shared Memory (SHM) transport and a UDP transport. With this configuration, PDP discovery metatraffic will always be performed through the UDP transport. However, if the SHM transport requirements are met, EDP discovery metatraffic and user data traffic will be sent through this transport. In order to capture the traffic and analyze it, the SHM transport is going to be disabled. Save the following XML configuration file at the desired location, which will be referred as <path_to_xml_config_file> from here onwards, with vulcanexus_disable_shm.xml name.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<dds>
   <profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
      <transport_descriptors>
         <transport_descriptor>
            <transport_id>udp_transport</transport_id>
            <type>UDPv4</type>
         </transport_descriptor>
      </transport_descriptors>
   </profiles>
   <participant profile_name="default_participant_profile" is_default_profile="true">
      <rtps>
         <userTransports>
            <transport_id>udp_transport</transport_id>
         </userTransports>
         <useBuiltInTransports>false</useBuiltInTransports>
      </rtps>
   </participant>
</dds>
```

(continues on next page)
This configuration disables the built-in transports and registers a user-defined UDPv4 transport.

After launching Wireshark (with administrative privileges to be able to capture network traffic), start capturing the traffic going through any interface. The traffic shown can be filtered using as parameter `rtps` to see only traffic of interest. Next, run the talker and the listener:

Shell 1 (Listener)

```
source /opt/vulcanexus/iron/setup.bash
FASTRTPS_DEFAULT_PROFILES_FILE=<path_to_xml_config_file>/vulcanexus_disable_shm.xml ros2␣
˓→run demo_nodes_cpp listener
```

Shell 2 (Talker)

```
source /opt/vulcanexus/iron/setup.bash
FASTRTPS_DEFAULT_PROFILES_FILE=<path_to_xml_config_file>/vulcanexus_disable_shm.xml ros2␣
˓→run demo_nodes_cpp talker
```

The traffic captured will show several `DATA(p)` messages which are the periodic DomainParticipant announcements sent to multicast (PDP) and to any DomainParticipant already matched (Liveliness). In this case, the EDP discovery traffic will also be captured as `DATA(w)`, DataWriter discovery metatraffic, and `DATA(r)`, DataReader discovery metatraffic. The screenshot below has been taken running this tutorial.
Static EDP discovery

Now, Static EDP discovery is going to be enabled in order to remove any EDP discovery metatraffic. Within Vulcanexus ecosystem, Static EDP can only be enabled through XML configuration. Whereas different Endpoint profiles can be defined within the same XML file using the Topic name, it is not possible to have different DomainParticipant profiles within the same file when using Vulcanexus. Consequently, a different configuration file is required for each DomainParticipant, because the Static EDP settings include the path to the static XML configuration file which defines the Endpoints, Topics, and Data types.

Save the following XML configuration files in the path_to_xml_config_file under the names talker_profile.xml and listener_profile.xml respectively:

talker_profile.xml

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<dds>
  <profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles" >
    <transport_descriptors>
      <transport_descriptor>
        <transport_id>udp_transport</transport_id>
        <type>UDPv4</type>
      </transport_descriptor>
    </transport_descriptors>

    <participant profile_name="talker_participant_profile" is_default_profile="true">
      <rtps>
        <name>talker</name>
        <userTransports>
          <transport_id>udp_transport</transport_id>
        </userTransports>
        <useBuiltinTransports>false</useBuiltinTransports>
        <builtin>
          <discovery_config>
            <EDP>STATIC</EDP>
            <static_edp_xml_config>file://static_edp_info.xml</static_edp_xml_config>
          </discovery_config>
        </builtin>
      </rtps>
    </participant>

    <data_writer profile_name="/chatter">
      <userDefinedID>101</userDefinedID>
    </data_writer>
  </profiles>
</dds>
```

listener_profile.xml

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<dds>
  <profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles" >
    <transport_descriptors>
      <transport_descriptor>
        <transport_id>udp_transport</transport_id>
        <type>UDPv4</type>
      </transport_descriptor>
    </transport_descriptors>
  </profiles>
</dds>
```
Besides setting the EDP discovery to STATIC and defining the path to the Static configuration XML file, the Endpoint userDefinedID must be set so the Endpoints can be identified in this Static configuration file. Every remote Endpoint must be defined within this configuration file, with the Topic and Data type name so matching can be performed by the local DomainParticipant statically without the need for the remote participant to send the Endpoint discovery information. The specific endpoint QoS has also to be defined or the default QoS will be assumed.

Save the static configuration XML file to the same location under the name static_edp_info.xml:

```xml
<staticdiscovery>
    <participant>
        <name>talker</name>
        <writer>
            <userId>101</userId>
            <topicName>rt/chatter</topicName>
            <topicDataType>std_msgs::msg::dds::String_<topicDataType>
            <reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
            <durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
        </writer>
    </participant>
    <participant>
        <name>listener</name>
        <reader>
            <userId>102</userId>
            <topicName>rt/chatter</topicName>
        </reader>
    </participant>
</staticdiscovery>
```
It is important to be aware of ROS 2 Topic mangling rules. Whereas the endpoint profile has to be set using the ROS 2 Topic name, the static discovery configuration must used the DDS Topic name.

Run again the talker-listener demo loading the corresponding XML configuration file using FASTRTPS_DEFAULT_PROFILES_FILE environment variable:

Shell 1 (Listener)

```bash
source /opt/vulcanexus/iron/setup.bash
FASTRTPS_DEFAULT_PROFILES_FILE=<path_to_xml_config_file>/listener_profile.xml ros2 run...demo_nodes_cpp listener
```

Shell 2 (Talker)

```bash
source /opt/vulcanexus/iron/setup.bash
FASTRTPS_DEFAULT_PROFILES_FILE=<path_to_xml_config_file>/talker_profile.xml ros2 run...demo_nodes_cpp talker
```

If the traffic is captured again using Wireshark, this time no DATA(w) or DATA(r) is exchanged, but communication is established.

**Full-fledged Static EDP discovery**

ROS 2 provides several introspection tools like ros2 topic list or ros2 node list. Running these commands with the previous example will show only the default ROS 2 topics (/parameter_events and /rosout) and no node. The /chatter topic and the /talker and /listener nodes would not appear even though they are up and running and communicating. The issue is related to the internal endpoints that ROS 2 ecosystem, and by extension Vulcanexus, launch within each node to ensure these tools work as expected. Also, the vast majority of these tools depend on another DomainParticipant known as the ROS 2 Daemon. This section will complete the previous XML files so a full-fledged communication within Vulcanexus ecosystem is achieved, with working introspection tools. Information about every internal ROS 2 endpoint has been included. These endpoints are related to topic and node discovery, ROS 2 logging module, and parameters services and events.

Below, the complete XML files are shown. Please, replaced the previous XML configuration files with these ones and the new one required to configure the ROS 2 daemon DomainParticipant.

`talker_profile.xml`

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<dds>
  <profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles"
<transport_descriptors>
<transport_descriptor>
<transport_id>udp_transport</transport_id>
<type>UDPv4</type>
</transport_descriptor>
</profiles>
</dds>
```
</transport_descriptors>

<participant profile_name="talker_participant_profile" is_default_profile="true">
<rtsp>
  <name>talker</name>
  <userTransports>
    <transport_id>udp_transport</transport_id>
  </userTransports>
  <useBuiltInTransports>false</useBuiltInTransports>
  <builtin>
    <discovery_config>
      <EDP>STATIC</EDP>
      <static_edp_xml_config>file://static_edp_info.xml</static_edp_xml_config>
    </discovery_config>
  </builtin>
</rtsp>
</participant>

<data_writer profile_name="/chatter">
  <userDefinedID>101</userDefinedID>
</data_writer>

<!-->ROS 2 Internal Endpoints<-->
<data_writer profile_name="/parameter_events">
  <userDefinedID>201</userDefinedID>
</data_writer>
<data_writer profile_name="/ros_discovery_info">
  <userDefinedID>202</userDefinedID>
</data_writer>
<data_writer profile_name="/rosout">
  <userDefinedID>203</userDefinedID>
</data_writer>
<data_writer profile_name="rr/talker/describe_parametersReply">
  <userDefinedID>204</userDefinedID>
</data_writer>
<data_writer profile_name="rr/talker/list_parametersReply">
  <userDefinedID>205</userDefinedID>
</data_writer>
<data_writer profile_name="rr/talker/get_parametersReply">
  <userDefinedID>206</userDefinedID>
</data_writer>
<data_writer profile_name="rr/talker/get_parameter_typesReply">
  <userDefinedID>207</userDefinedID>
</data_writer>
<data_writer profile_name="rr/talker/set_parametersReply">
  <userDefinedID>208</userDefinedID>
</data_writer>
<data_writer profile_name="rr/talker/set_parameters_atomicallyReply">
  <userDefinedID>209</userDefinedID>
</data_writer>
</data_writer>
<data_reader profile_name="/parameter_events">

(continued on next page)
<userDefinedID>210</userDefinedID>
</data_reader>
<data_reader profile_name="ros_discovery_info">
    <userDefinedID>211</userDefinedID>
</data_reader>
<data_reader profile_name="rq/talker/set_parametersRequest">
    <userDefinedID>212</userDefinedID>
</data_reader>
<data_reader profile_name="rq/talker/get_parametersRequest">
    <userDefinedID>213</userDefinedID>
</data_reader>
<data_reader profile_name="rq/talker/get_parameter_typesRequest">
    <userDefinedID>214</userDefinedID>
</data_reader>
<data_reader profile_name="rq/talker/set_parameters_atomicallyRequest">
    <userDefinedID>215</userDefinedID>
</data_reader>
<data_reader profile_name="rq/talker/describe_parametersRequest">
    <userDefinedID>216</userDefinedID>
</data_reader>
<data_reader profile_name="rq/talker/list_parametersRequest">
    <userDefinedID>217</userDefinedID>
</data_reader>
</data_reader>
</profiles>
</dds>

listener_profile.xml

<?xml version="1.0" encoding="UTF-8" ?>
<dds>
    <profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
        <transport_descriptors>
            <transport_descriptor>
                <transport_id>udp_transport</transport_id>
                <type>UDPv4</type>
            </transport_descriptor>
        </transport_descriptors>

        <participant profile_name="listener_participant_profile" is_default_profile="true">
            <rtps>
                <name>listener</name>
                <userTransports>
                    <transport_id>udp_transport</transport_id>
                </userTransports>
                <useBuiltInTransports>false</useBuiltInTransports>
                <builtin>
                    <discovery_config>
                        <EDP>STATIC</EDP>
                        <static_edp_xml_config>file://static_edp_info.xml</static_edp_xml_config>
                    </discovery_config>
                </builtin>
            </rtps>
        </participant>
    </profiles>
</dds>
(continued from previous page)

```xml
<participant>

<data_reader profile_name="/chatter">
  <userDefinedID>102</userDefinedID>
</data_reader>

<!-->ROS 2 Internal Endpoints<-->

<data_reader profile_name="/parameter_events">
  <userDefinedID>1</userDefinedID>
</data_reader>
<data_reader profile_name="/ros_discovery_info">
  <userDefinedID>2</userDefinedID>
</data_reader>
<data_reader profile_name="/parameter_events">
  <userDefinedID>3</userDefinedID>
</data_reader>
<data_reader profile_name="/parameter_events">
  <userDefinedID>4</userDefinedID>
</data_reader>
<data_reader profile_name="/parameter_events">
  <userDefinedID>5</userDefinedID>
</data_reader>
<data_reader profile_name="/parameter_events">
  <userDefinedID>6</userDefinedID>
</data_reader>
<data_reader profile_name="/parameter_events">
  <userDefinedID>7</userDefinedID>
</data_reader>
<data_reader profile_name="/parameter_events">
  <userDefinedID>8</userDefinedID>
</data_reader>
<data_writer profile_name="/parameter_events">
  <userDefinedID>9</userDefinedID>
</data_writer>
<data_writer profile_name="/ros_discovery_info">
  <userDefinedID>10</userDefinedID>
</data_writer>
<data_writer profile_name="/rosout">
  <userDefinedID>11</userDefinedID>
</data_writer>
<data_writer profile_name="/rr/listener/describe_parametersReply">
  <userDefinedID>12</userDefinedID>
</data_writer>
<data_writer profile_name="/rr/listener/get_parametersReply">
  <userDefinedID>13</userDefinedID>
</data_writer>
<data_writer profile_name="/rr/listener/get_parameter_typesReply">
  <userDefinedID>14</userDefinedID>
</data_writer>
<data_writer profile_name="/rr/listener/set_parametersReply">
(continues on next page)
```

(continues on next page)
Vulcanexus Documentation, Release 1.0.0

ros2_daemon_profile.xml

<?xml version="1.0" encoding="UTF-8"?>
<dds>
  <profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <transport_descriptors>
      <transport_descriptor>
        <transport_id>udp_transport</transport_id>
        <type>UDPv4</type>
      </transport_descriptor>
    </transport_descriptors>

    <participant profile_name="ros2_daemon_participant_profile" is_default_profile="true">
      <rtps>
        <name>ros2_daemon</name>
        <userTransports>
          <transport_id>udp_transport</transport_id>
        </userTransports>
        <useBuiltInTransports>false</useBuiltInTransports>
        <builtin>
          <discovery_config>
            <EDP>STATIC</EDP>
            <static_edp_xml_config>file:///static_edp_info.xml</static_edp_xml_config>
          </discovery_config>
        </builtin>
      </rtps>
      <data_writer profile_name="/rosout"/>
      <data_writer profile_name="ros_discovery_info"/>
      <data_writer profile_name="/parameter_events"/>
    </participant>
  </profiles>
</dds>

(continues on next page)
<data_reader profile_name="ros_discovery_info">
    <userDefinedID>54</userDefinedID>
</data_reader>
</profiles>
</dds>

static_edp_info.xml

<staticdiscovery>
    <participant>
        <name>talker</name>
        <writer>
            <userId>101</userId>
            <topicName>rt/chatter</topicName>
            <topicDataType>std_msgs::msg::dds__String_</topicDataType>
            <reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
            <durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
        </writer>
        <writer>
            <userId>201</userId>
            <topicName>rt/parameter_events</topicName>
            <topicDataType>rcl_interfaces::msg::dds__ParameterEvent_</topicDataType>
            <reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
            <durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
        </writer>
        <writer>
            <userId>202</userId>
            <topicName>ros_discovery_info</topicName>
            <topicDataType>rmw_dds_common::msg::dds__ParticipantEntitiesInfo_</topicDataType>
            <reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
            <durabilityQos>TRANSIENT_LOCAL_DURABILITY_QOS</durabilityQos>
        </writer>
        <writer>
            <userId>203</userId>
            <topicName>rt/rosout</topicName>
            <topicDataType>rcl_interfaces::msg::dds__Log_</topicDataType>
            <reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
            <durabilityQos>TRANSIENT_LOCAL_DURABILITY_QOS</durabilityQos>
        </writer>
        <writer>
            <userId>204</userId>
            <topicName>rr/talker/describe_parametersReply</topicName>
            <topicDataType>rcl_interfaces::srv::dds__DescribeParameters_Response_</topicDataType>
            <reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
            <durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
        </writer>
        <writer>
            <userId>205</userId>
            <topicName>rr/talker/list_parametersReply</topicName>
            <topicDataType>rcl_interfaces::srv::dds__ListParameters_Response_</topicDataType>
            <reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
            <durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
        </writer>
    </participant>
</staticdiscovery>
<reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
<durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
</writer>
<writer>
  <userId>206</userId>
  <topicName>rr/talker/get_parametersReply</topicName>
  <topicDataType>rcl_interfaces::srv::dds_::GetParameters_Response_</topicDataType>
  <reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
  <durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
</writer>
<writer>
  <userId>207</userId>
  <topicName>rr/talker/get_parameter_typesReply</topicName>
  <topicDataType>rcl_interfaces::srv::dds_::GetParametersTypes_Response_</topicDataType>
  <reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
  <durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
</writer>
<writer>
  <userId>208</userId>
  <topicName>rr/talker/set_parametersReply</topicName>
  <topicDataType>rcl_interfaces::srv::dds_::SetParameters_Response_</topicDataType>
  <reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
  <durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
</writer>
<writer>
  <userId>209</userId>
  <topicName>rr/talker/set_parameters_atomicallyReply</topicName>
  <topicDataType>rcl_interfaces::srv::dds_::SetParametersAtomically_Response_</topicDataType>
  <reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
  <durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
</writer>
<reader>
  <userId>210</userId>
  <topicName>rt/parameter_events</topicName>
  <topicDataType>rcl_interfaces::msg::dds_::ParameterEvent_</topicDataType>
  <reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
  <durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
</reader>
<reader>
  <userId>211</userId>
  <topicName>ros_discovery_info</topicName>
  <topicDataType>rmw_dds_common::msg::dds_::ParticipantEntitiesInfo_</topicDataType>
  <reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
  <durabilityQos>TRANSIENT_LOCAL_DURABILITY_QOS</durabilityQos>
</reader>
<reader>
  <userId>212</userId>
</reader>
<topicName>rq/talker/set_parametersRequest</topicName>
<topicDataType>rcl_interfaces::srv::dds_::SetParameters_Request_</topicDataType>
<reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
<durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
</reader>

<topicName>rq/talker/get_parametersRequest</topicName>
<topicDataType>rcl_interfaces::srv::dds_::GetParameters_Request_</topicDataType>
<reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
<durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
</reader>

<topicName>rq/talker/get_parameter_typesRequest</topicName>
<topicDataType>rcl_interfaces::srv::dds_::GetParameterTypes_Request_</topicDataType>
<reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
<durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
</reader>

<topicName>rq/talker/set_parameters_atomicallyRequest</topicName>
<topicDataType>rcl_interfaces::srv::dds_::SetParametersAtomically_Request_</topicDataType>
<reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
<durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
</reader>

<topicName>rq/talker/describe_parametersRequest</topicName>
<topicDataType>rcl_interfaces::srv::dds_::DescribeParameters_Request_</topicDataType>
<reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
<durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
</reader>

<topicName>rq/talker/list_parametersRequest</topicName>
<topicDataType>rcl_interfaces::srv::dds_::ListParameters_Request_</topicDataType>
<reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
<durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
</reader>

<name>listener</name>

<userId>102</userId>
<topicName>rt/chatter</topicName>
(continues on next page)
<topicDataType>std_msgs::msg::dds::String</topicDataType>
<reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
<durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
</reader>

<reader>
  <userId>1</userId>
  <topicName>rt/parameter_events</topicName>
  <topicDataType>rcl_interfaces::msg::ParameterEvent</topicDataType>
  <reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
  <durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
</reader>

<reader>
  <userId>2</userId>
  <topicName>ros_discovery_info</topicName>
  <topicDataType>rmw_dds_common::msg::ParticipantEntitiesInfo</topicDataType>
  <reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
  <durabilityQos>TRANSIENT_LOCAL_DURABILITY_QOS</durabilityQos>
</reader>

<reader>
  <userId>3</userId>
  <topicName>rq/listener/get_parametersRequest</topicName>
  <topicDataType>rcl_interfaces::srv::GetParameters_Request</topicDataType>
  <reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
  <durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
</reader>

<reader>
  <userId>4</userId>
  <topicName>rq/listener/get_parameter_typesRequest</topicName>
  <topicDataType>rcl_interfaces::srv::GetParameterTypes_Request</topicDataType>
  <reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
  <durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
</reader>

<reader>
  <userId>5</userId>
  <topicName>rq/listener/set_parametersRequest</topicName>
  <topicDataType>rcl_interfaces::srv::SetParameters_Request</topicDataType>
  <reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
  <durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
</reader>

<reader>
  <userId>6</userId>
  <topicName>rq/listener/set_parameters_atomicallyRequest</topicName>
  <topicDataType>rcl_interfaces::srv::SetParametersAtomically_Request</topicDataType>
  <reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
  <durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
</reader>
<userId>7</userId>
<topicName>rq/listener/describe_parametersRequest</topicName>
<topicDataType>rcl_interfaces::srv::dds_::DescribeParameters_Request_</topicDataType>
<reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
<durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
</reader>
<userId>8</userId>
<topicName>rq/listener/list_parametersRequest</topicName>
<topicDataType>rcl_interfaces::srv::dds_::ListParameters_Request_</topicDataType>
<reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
<durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
</reader>
<userId>9</userId>
<topicName>rt/parameter_events</topicName>
<topicDataType>rcl_interfaces::msg::dds_::ParameterEvent_</topicDataType>
<reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
<durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
</writer>
<userId>10</userId>
<topicName>ros_discovery_info</topicName>
<topicDataType>rmw_dds_common::msg::dds_::ParticipantEntitiesInfo_</topicDataType>
<reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
<durabilityQos>TRANSIENT_LOCAL_DURABILITY_QOS</durabilityQos>
</writer>
<userId>11</userId>
<topicName>rt/rosout</topicName>
<topicDataType>rcl_interfaces::msg::dds_::Log_</topicDataType>
<reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
<durabilityQos>TRANSIENT_LOCAL_DURABILITY_QOS</durabilityQos>
</writer>
<userId>12</userId>
<topicName>rr/listener/describe_parametersReply</topicName>
<topicDataType>rcl_interfaces::srv::dds_::DescribeParameters_Response_</topicDataType>
<reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
<durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
</writer>
<userId>13</userId>
<topicName>rr/listener/get_parametersReply</topicName>
<topicDataType>rcl_interfaces::srv::dds_::GetParameters_Response_</topicDataType>
<reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
<durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
<writer>
  <userId>14</userId>
  <topicName>rr/listener/get_parameter_typesReply</topicName>
  <topicDataType>rcl_interfaces::srv::dds_::GetParametersTypes_Response_</topicDataType>
  <reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
  <durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
</writer>

<writer>
  <userId>15</userId>
  <topicName>rr/listener/set_parametersReply</topicName>
  <topicDataType>rcl_interfaces::srv::dds_::SetParameters_Response_</topicDataType>
  <reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
  <durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
</writer>

<writer>
  <userId>16</userId>
  <topicName>rr/listener/set_parameters_atomicallyReply</topicName>
  <topicDataType>rcl_interfaces::srv::dds_::SetParametersAtomically_Response_</topicDataType>
  <reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
  <durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
</writer>

<writer>
  <userId>17</userId>
  <topicName>rr/listener/list_parametersReply</topicName>
  <topicDataType>rcl_interfaces::srv::dds_::ListParameters_Response_</topicDataType>
  <reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
  <durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
</writer>

<writer>
  <userId>51</userId>
  <topicName>rt/rosout</topicName>
  <topicDataType>rcl_interfaces::msg::dds_::Log_</topicDataType>
  <reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
  <durabilityQos>TRANSIENT_LOCAL_DURABILITY_QOS</durabilityQos>
</writer>

<writer>
  <userId>52</userId>
  <topicName>ros_discovery_info</topicName>
  <topicDataType>rmw_dds_common::msg::dds_::ParticipantEntitiesInfo_</topicDataType>
  <reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
  <durabilityQos>TRANSIENT_LOCAL_DURABILITY_QOS</durabilityQos>
</writer>
Run again the talker-listener demo loading the corresponding XML configuration file using FASTRTPS_DEFAULT_PROFILES_FILE environment variable. This time a third terminal is required to run the ROS 2 daemon.

Shell 1 (Listener)

```
source /opt/vulcanexus/iron/setup.bash
FASTRTPS_DEFAULT_PROFILES_FILE=<path_to_xml_config_file>/listener_profile.xml ros2 run...
...demo_nodes_cpp listener
```

Shell 2 (Talker)

```
source /opt/vulcanexus/iron/setup.bash
FASTRTPS_DEFAULT_PROFILES_FILE=<path_to_xml_config_file>/talker_profile.xml ros2 run...
...demo_nodes_cpp talker
```

Shell 3 (ROS 2 Daemon)

```
source /opt/vulcanexus/iron/setup.bash
ros2 daemon stop
FASTRTPS_DEFAULT_PROFILES_FILE=<path_to_xml_config_file>/ros2_daemon_profile.xml ros2...
...daemon start
ros2 topic list
ros2 node list
```

Now, besides the internal ROS 2 Topics, the /chatter topic will be listed as well as the ROS 2 nodes /talker and /listener.
Modify Discovery Server locators on run-time

- **Background**
- **Overview**
- **Prerequisites**
  - Set up the discovery server networks
    - Configure the discovery entities
    - `FASTDDS_ENVIRONMENT_FILE`
- **Run the example**
- **Discovery Server on run-time**

**Background**

The **Discovery Server** is a Fast DDS enabled feature that procures an alternative discovery mechanism to the default ROS 2 discovery mechanism, Simple Discovery Protocol (SDP), which is served by the DDS implementations according to the DDS specification. Whereas SDP (right figure) provides automatic out-of-the-box discovery by leveraging multicast, ROS 2 Discovery Server (left figure) provides a centralized hub for managing discovery which drastically reduces network bandwidth utilization when compared to SDP, since the nodes, publishers, and subscribers, only discovered those remote ROS 2 entities with which they need to communicate (as opposed to the SDP model where everyone knows about each other).

![Fig. 4: Comparison of Discovery Server and Simple Discovery Protocol mechanisms](image)

A **server** is a context to which the **clients** (and maybe other **servers**) send their discovery information. The role of the **server** is to re-distribute the **clients** (and **servers**) discovery information to their known **clients** and **servers**.

A **client** is a context that connects to one or more **servers** from which it receives only the discovery information they require to establish communication with matching endpoints.

**Overview**

This tutorial will use ROS 2 **demo_nodes_cpp talker** and **listener** applications to establish the communication between **clients** through the **servers**. Two different discovery server networks would be set. Each of them would have at least one discovery **server**, and one **client** context, which would be running a **talker** node. The aim of the tutorial is to add a **client** context running a **listener** node that receives the **talker** node publications coming from the same discovery server network, and on run-time, update the discovery server list to make allow the **listener** node to receive also the publications coming from the other discovery server network **talker** node.

To do so, the **Client listener** node discovery servers list would be updated on run-time to include the **Server B**, so the **Client listener** node would be able to receive the new publications from the **Client talker B** node.
Prerequisites

First of all, make sure that Vulcanexus Iron is installed. The docker installation is required for this tutorial (see Docker installation).

Note: It is highly recommended to complete the Configuring Fast-DDS QoS via XML profiles tutorial to learn how to configure ROS 2 via XML configuration files.

Set up the discovery server networks

Set a specific interface pool for this implementation by running the following command.

```
docker network create --subnet=113.11.3.0/16 vulcanexus_tutorial
```

Then, open five terminals, and run the Vulcanexus Iron image in each one with the following commands with the specific IP addresses:

**Discovery Server Network A**

**Server A**

```
docker run -it --rm --name server_A --net vulcanexus_tutorial --ip 113.11.3.0 ubuntu-vulcanexus:iron-desktop
```

**Client talker A**

```
docker run -it --rm --name talker_A --net vulcanexus_tutorial --ip 113.11.3.2 ubuntu-vulcanexus:iron-desktop
```

**Client listener**

```
docker run -it --rm --name listener --net vulcanexus_tutorial --ip 113.11.3.3 ubuntu-vulcanexus:iron-desktop
```
Discover Server Network B
Server B

```
docker run -it --rm --name server_B --net vulcanexus_tutorial --ip 113.11.3.1 ubuntu-vulcanexus:iron-desktop
```

Client talker B

```
docker run -it --rm --name talker_B --net vulcanexus_tutorial --ip 113.11.3.4 ubuntu-vulcanexus:iron-desktop
```

**Configure the discovery entities**

The *servers* configuration is really simple: by using the Fast DDS Discovery CLI, the *server* discovery id would be set as 0 in the Discovery Server Network A, and 1 in the Discovery Server Network B.

Discovery Server Network A
Server A

```
fastdds discovery --server-id 0
```

Discovery Server Network B
Server B

```
fastdds discovery --server-id 1
```

The outputs should look similar to the followings:

Discovery Server Network A
Server A

```
### Server is running ###
Participant Type: SERVER
Server ID: 0
Server GUID prefix: 44.53.00.5f.45.50.52.4f.53.49.4d.41
Server Addresses: UDPv4:[0.0.0.0]:11811
```

Discovery Server Network B
Server B

```
### Server is running ###
Participant Type: SERVER
Server ID: 1
Server GUID prefix: 44.53.01.5f.45.50.52.4f.53.49.4d.41
Server Addresses: UDPv4:[0.0.0.0]:11811
```

The output displays the *server ID* set, followed by the server GUID prefix. Server address 0.0.0.0 tells *Fast DDS* to listen on all available interfaces. Finally, the 11811 default port would be necessary for further configuration.
**FASTDDS_ENVIRONMENT_FILE**

Each *client* needs to know to which *server* it should connect. In order to update that information on run-time, the environment variable `FASTDDS_ENVIRONMENT_FILE` must be set to an existing json file (see Fast DDS Environment variables). In that way, the environment variable that sets the *server* information would be loaded from a file instead of from the environment. This allows to change the discovery servers information by simply modifying the json file.

Create a `discovery_servers.json` json file for each *client* and introduce the *server* IP address and port.

**Note:** As long as the default port has been used in the tutorial, it could be omitted in the json configuration.

Discovery Server Network A

Client talker A

```
{
   "ROS_DISCOVERY_SERVER": "113.11.3.0:11811"
}
```

Client listener

```
{
   "ROS_DISCOVERY_SERVER": "113.11.3.0:11811"
}
```

Discovery Server Network B

Client talker B

```
{
   "ROS_DISCOVERY_SERVER": ";113.11.3.1:11811"
}
```

The environment value can be either an absolute or relative path. Set up the environment file:

```
export FASTDDS_ENVIRONMENT_FILE="discovery_servers.json"
```

**Run the example**

After all the configuration has been set, run the discovery servers, and the talker and listener client nodes:

Discovery Server Network A

Client talker A

```
export FASTDDS_ENVIRONMENT_FILE="discovery_servers.json"
ros2 run demo_nodes_cpp talker
```

Client listener

```
export FASTDDS_ENVIRONMENT_FILE="discovery_servers.json"
ros2 run demo_nodes_cpp listener
```

Discovery Server Network B

Client talker B

4.10. Vulcanexus Core Tutorials 1537
The expected output is that the Client listener node would only receive the publications from the Client talker A node. Now, let’s add the Server B as a discovery server in the Client listener on run-time to receive the Client talker B publications.

**Discovery Server on run-time**

During execution, modify the Client listener node json file to include the Server B data, as follows:

```json
{
    "ROS_DISCOVERY_SERVER": "113.11.3.0:11811;113.11.3.1:11811"
}
```

After saving the file, the Client listener would discover the Client talker B through the discovery server just set. Then, the Client listener would start receiving Client talker B publications.

**TCP over WAN with Discovery Server**

- **Background**
  - TCP: Transmission Control Protocol
  - WAN: Wide Area Network
  - Discovery Server
- **Overview**
- **Prerequisites**
  - Set up the docker networks
  - XML configuration files
Background

In this tutorial, it is explained how to deploy a TCP communication over WAN using Discovery Server as discovery mechanism. The following diagram describes the main idea of the deployment tutorial.

Fig. 5: TCP communication over WAN with Discovery Server

TCP: Transmission Control Protocol

The Transmission Control Protocol is an Internet Protocol which provides mainly reliability in the communication process. As it is connection-oriented, this protocol has several features that ensure the delivery, order and error check of the packages. Despite the fact that the latency is higher than other Internet Protocols such as UDP, its use has several advantages in particular scenarios where reliability has greater importance than the latency cost.

WAN: Wide Area Network

A Wide Area Network is a telecommunication network extended over a large geographic area. It usually involves a large number of nodes and redundancy, to ensure the reliability of the network. The Internet could be considered as a WAN itself.

Discovery Server

The Discovery Server is a Fast DDS enabled feature that procures an alternative discovery mechanism to the default ROS 2 discovery mechanism, Simple Discovery Protocol (SDP), which is served by the DDS implementations according to the DDS specification. Whereas SDP (right figure) provides automatic out-of-the-box discovery by leveraging multicast, the ROS 2 Discovery Server (left figure) provides a centralized hub for managing discovery which drastically reduces network bandwidth utilization when compared to SDP, since the nodes, publishers, and subscribers, only discover those remote ROS 2 entities with which they need to communicate with, as opposed to the SDP model where everyone knows each other.

A server is a context to which the clients (and maybe other servers) send their discovery information. The role of the server is to re-distribute the clients (and servers) discovery information to their known clients and servers.

A client is a context that connects to one or more servers from which it receives only the discovery information they require to establish communication with matching endpoints.
Overview

This tutorial will use ROS 2 demo_nodes_cpp talker and listener applications to establish the communication between clients through the server. Each node would be deployed in a docker container, in different networks.

The discovery server, would be deployed also in its own docker container, but it will be part of two networks: the WAN and the same network as the talker node. This setup allows the discovery server to perform routing tasks as a regular router does in a LAN (having a private IP which would be in the talker LAN IP, and a public IP, which would be in the WAN IP). Additionally, the same routing element is required in the listener LAN. A router container is included as the intermediary between the WAN and the listener node network.

Within these defined scenario, the following diagram describes the network setup for deploying the simulation.

Fig. 6: Network setup example to simulate TCP communication over WAN with Discovery Server

The expected behavior is that both talker and listener are able to connect to the discovery server, discover each other, and send and receive (respectively) the HelloWorld example messages over the WAN.

Important: All the communication, including discovery phase, would be performed using TCP. See the Fast DDS discovery phases documentation for further information.

Note: Docker performs a network configuration to isolate each of the docker networks. This tutorial would update some of the iptables configuration set automatically by docker in order to simulate properly the WAN scenario.
Prerequisites

First of all, make sure that Vulcanexus Iron is installed. The docker installation is required for this tutorial (see Docker installation), together with the eProsima Vulcanexus docker image. The Vulcanexus Iron image can be downloaded by running:

```
docker pull eprosima/vulcanexus:iron
```

In addition, docker compose is used to simplify the example deployment, and iptables is required to update the network configuration. Follow docker compose installation guide, and install the iptables dependency by running:

```
sudo apt install iptables
```

**Note:** It is highly recommended to complete the Configuring Fast-DDS QoS via XML profiles tutorial to learn how to configure ROS 2 via XML configuration files.

Set up the docker networks

Set the specific interface pools by running the following commands.

```
docker network create --subnet=10.1.0.0/16 talker_net
docker network create --subnet=10.2.0.0/16 listener_net
docker network create --subnet=10.3.0.0/16 wan_net
```

The three networks would have been created, all of them isolated from the others. In order to enable the communication between networks, it is mandatory to update the system iptables.

```
sudo iptables -I DOCKER-ISOLATION-STAGE-2 -i talker_net -o listener_net -j ACCEPT
sudo iptables -I DOCKER-ISOLATION-STAGE-2 -i listener_net -o talker_net -j ACCEPT
sudo iptables -I DOCKER-ISOLATION-STAGE-2 -i talker_net -o wan_net -j ACCEPT
sudo iptables -I DOCKER-ISOLATION-STAGE-2 -i listener_net -o wan_net -j ACCEPT
sudo iptables -I DOCKER-ISOLATION-STAGE-2 -i wan_net -o talker_net -j ACCEPT
sudo iptables -I DOCKER-ISOLATION-STAGE-2 -i wan_net -o listener_net -j ACCEPT
```

This set of commands is enabling both `input (-i)` and `output (-o)` tables to allow both sides communication between the three networks.

XML configuration files

It is mandatory to set the discovery `server` and the `client` nodes with the following configuration options in order to enable the TCP communication, regardless of whether the deployment is over Docker networks or WAN.

Both XML configuration files will be later used in the docker compose instructions to perform the containers deployment.
Server side

The following XML configuration describes the discovery server TCP configuration required. Create a workspace to run the tutorial, and include an XML configuration file named `server_configuration.xml` with the below code.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
  <!-- TCP transport descriptor -->
  <transport_descriptors>
    <!-- Transport descriptor identifier -->
    <transport_id>TCP_ds_transport</transport_id>

    <!-- TCP transport -->
    <type>TCPv4</type>

    <!-- Discovery server listening (physical) port -->
    <listening_ports>
      <port>10111</port>
    </listening_ports>

    <!-- Discovery server WAN address -->
    <wan_addr>10.1.1.1</wan_addr>
  </transport_descriptor>
</transport_descriptors>

<!-- Participant profile -->
<participant profile_name="TCP_discovery_server_profile" is_default_profile="true">
  <rtps>
    <!-- Use declared TCP transport descriptor -->
    <userTransports>
      <transport_id>TCP_ds_transport</transport_id>
    </userTransports>

    <!-- Do not use default builtin transports -->
    <useBuiltinTransports>false</useBuiltinTransports>

    <!-- Set server's GUID prefix -->
    <prefix>44.53.00.5f.45.50.52.4f.53.49.4d.41</prefix>
  </rtps>

  <!-- Discovery server configuration -->
  <discovery_config>
    <!-- Node kind: SERVER -->
    <discoveryProtocol>SERVER</discoveryProtocol>
  </discovery_config>

  <!-- Set server's listening locator for discovery phase -->
  <metatrafficUnicastLocatorList>
    <locator>
      <!-- TCP Discovery server listening locator -->
      <tcpv4>
      </tcpv4>
    </locator>
  </metatrafficUnicastLocatorList>
</participant>
</profiles>
```

(continues on next page)
Note that in the discovery configuration section, the profile is described as *server*, with a specific GUID prefix, and listening locator. This listening locator is configured with the IP *10.1.1.1*, which belongs to the *talker* node LAN, and its physical port is included in the transport descriptor as a known listening port. This, along setting the wan address with the previously mentioned IP address, and setting the transport descriptor and locator types as *TCPv4*, is crucial to ensure the TCP communication.

**Client side**

Include the following XML configuration in the workspace, and name the file as *talker_configuration.xml*. This former configuration describes the *talker* node configuration for the discovery phase, and transport layer.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles" >
   <!-- TCP transport descriptor -->
   <transport_descriptors>
      <!-- Transport descriptor identifier -->
      <transport_id>TCP_talker_transport</transport_id>

      <!-- TCP transport -->
      <type>TCPv4</type>

      <!-- Talker listening (physical) port -->
      <listening_ports>
         <port>10102</port>
      </listening_ports>

      <!-- Talker WAN address -->
      <wan_addr>10.1.0.2</wan_addr>
   </transport_descriptors>

   <!-- Participant profile -->
   <participant profile_name="TCP_talker_profile" is_default_profile="true">
      <!-- Use declared TCP transport descriptor -->
      <userTransports>
         <transport_id>TCP_talker_transport</transport_id>
      </userTransports>
   </participant>
</profiles>
```

(continues on next page)
It involves setting the profile as *client* in the discovery configuration section, and adding the discovery *server* GUID prefix and listening locator. The talker TCP transport descriptor must be configured with the wan address and physical port described. Note that the listening port configured must be different from the set in the discovery *server*.

Then, also include the following listener XML configuration in the workspace, and name the file as `listener_configuration.xml`.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">

<!-- TCP transport descriptor -->
<transport_descriptors>
  <transport_descriptor>
    <!-- Transport descriptor identifier -->
    <transport_id>TCP_listener_transport</transport_id>

    <!-- TCP transport -->
    <type>TCPv4</type>

    <!-- Listener listening (physical) port -->
    <listening_ports>

    </listening_ports>

  </transport_descriptor>
</transport_descriptors>

</profiles>
```
<port>10202</port>
</listening_ports>

<!-- Listener WAN address -->
<wan_addr>10.2.0.2</wan_addr>
</transport_descriptor>
</transport_descriptors>

<!-- Participant profile -->
<participant profile_name="TCP_listener_profile" is_default_profile="true">

<!-- Use declared TCP transport descriptor -->
<userTransports>
  <transport_id>TCP_listener_transport</transport_id>
</userTransports>

<!-- Do not use default builtin transports -->
<useBuiltinTransports>false</useBuiltinTransports>

<builtin>
  <!-- Discovery phase configuration -->
  <discovery_config>
    <!-- Node kind: CLIENT -->
    <discoveryProtocol>CLIENT</discoveryProtocol>
    <!-- Discovery Server configuration -->
    <discoveryServersList>
      <RemoteServer prefix="44.53.00.5f.45.50.52.4f.53.49.4d.41">
        <metatrafficUnicastLocatorList>
          <!-- Set server's locator for discovery phase -->
          <locator>
            <tcpv4>
              <address>10.1.1.1</address>
              <port>10111</port>
              <physical_port>10111</physical_port>
            </tcpv4>
          </locator>
        </metatrafficUnicastLocatorList>
      </RemoteServer>
    </discoveryServersList>
  </discovery_config>
</builtin>
</rtps>
</participant>
</profiles>

Note that the listener discovery server configuration is exactly the same as talker discovery server configuration, but the WAN IP address and the listening port set in the transport descriptor configuration are different according to each LAN.
Create the Docker compose file

Once the XML configuration files have been included in the workspace, create a new file named Dockerfile and paste the following code. It contains the required commands to assemble a docker image based on Vulcanexus Iron. That includes some dependencies, and the recently created XML configuration files.

```
FROM eprosima/vulcanexus:iron

RUN apt update && DEBIAN_FRONTEND=noninteractive apt install -y ros-iron-demo-nodes-cpp net-tools iptables

COPY listener_configuration.xml server_configuration.xml talker_configuration.xml ./
```

Finally, the compose.yml is where all the containers and their configuration are described:

- **fast.dds.discovery_server**: the container is included in both created talker_net and wan_net. The IP addresses has been manually set to 10.1.1.1 in the talker_net, and 10.3.1.1 in the wan_net. This container’s default gateway is redirected to the router container. The iptables has been configured to redirect any traffic from any network and interface.

- **ros_listener**: the container is included in the created listener_net. The IP address has been manually set to 10.2.0.2, and the environment variable FASTRTPS_DEFAULT_PROFILES_FILE is set with the XML configuration listener_configuration.xml. This container’s default gateway is redirected to the router container.

- **ros.talker**: the container is included in the created talker_net. The IP address has been manually set to 10.1.0.2, and the environment variable FASTRTPS_DEFAULT_PROFILES_FILE is set with the XML configuration talker_configuration.xml. This container’s default gateway is redirected to the discovery server, which would behave as a router too.

- **router**: the container is included in both listener_net and wan_net networks. The IP addresses has been manually set to 10.2.1.1 in the listener_net, and 10.3.2.1 in the wan_net. This container’s default gateway is redirected to the discovery server. The iptables has been configured to redirect traffic from any network and interface.

Please, include the following compose.yml file in the workspace.

```
version: "3"

networks:
  listener_net:
    name: listener_net
    external: true
  talker_net:
    name: talker_net
    external: true
  wan_net:
    name: wan_net
    external: true

services:
  fast.dds.discovery_server:
    build:
      context: .
    command: sh -c "iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE && iptables -t nat -A POSTROUTING -o eth1 -j MASQUERADE &&
```

(continues on next page)
iptables -A FORWARD -i eth0 -j ACCEPT &&
iptables -A FORWARD -i eth1 -j ACCEPT &&
iptables -A INPUT -i eth0 -j ACCEPT &&
iptables -A INPUT -i eth1 -j ACCEPT &&
iptables -A OUTPUT -j ACCEPT &&
route del default &&
route add default gw 10.3.2.1 eth1 &&
fastdds discovery -i 0 -x server_configuration.xml"

privileged: true
cap_add:
  - ALL
volumes:
  - /tmp/.X11-unix:/tmp/.X11-unix:rw
environment:
  - DISPLAY
sysctls:
  - net.ipv4.conf.all.forwarding=1
  - net.ipv4.ip_forward=1

networks:
  talker_net:
    ipv4_address: 10.1.1.1
  wan_net:
    ipv4_address: 10.3.1.1

ros_listener:
  build:
    context: .
  command: >-
    sh -c "route del default &&
    route add default gw 10.2.1.1 eth0 &&
    ros2 run demo_nodes_cpp listener"
cap_add:
  - ALL
privileged: true
volumes:
  - /tmp/.X11-unix:/tmp/.X11-unix:rw
environment:
  DISPLAY: ":1"
  FASTRTPS_DEFAULT_PROFILES_FILE: "listener_configuration.xml"

networks:
  listener_net:
    ipv4_address: 10.2.0.2

ros_talker:
  build:
    context: .
  command: >-
    sh -c "route del default &&
    route add default gw 10.1.1.1 eth0 &&
    ros2 run demo_nodes_cpp talker"
privileged: true
Run the example

Make sure that the workspace has been set with all the previous files, and the docker networks and iptables have been set too.

```bash
<workspace>
  ·  compose.yml
  ·  Dockerfile
  ·  listener_configuration.xml
  ·  server_configuration.xml
```

(continues on next page)
As explained in the *Overview*, the expected behavior is that both *talker* and *listener* are able to connect to the *discovery server*, discover each other, and send and receive (respectively) the *HelloWorld* example messages over the WAN simulation using TCP.

Run the example:

```
cd <workspace>
docker compose -f compose.yml up --build
```

**Clean workspace**

Stop the example by pressing `Ctrl + C`, and stop the containers by running:

```
docker compose -f compose.yml down
```

The docker networks, containers and images can be removed using the docker `prune` argument, but using the `rm` argument plus the identifiers would remove only the ones created for this tutorial:

```
docker network rm listener_net talker_net wan_net
docker container rm <workspace_name>_fast_dds_discovery_server_1 <workspace_name>_router_1 <workspace_name>_ros_listener_1 <workspace_name>_ros_talker_1
docker image rm <workspace_name>_fast_dds_discovery_server:latest <workspace_name>_router:latest <workspace_name>_ros_listener:latest <workspace_name>_ros_talker:latest
```

**Note:** The `iptables` configuration would be removed automatically each time the system gets rebooted.

**TCP over real WAN with Discovery Server**

The configuration of the docker networks and the `iptables` has been performed to simulate the WAN scenario locally. The requirements to achieve TCP communication over WAN with Discovery Server in a real deployment are launching the three elements of the communication (*talker*, *listener* and *discovery server*) with their corresponding XML configurations applied, and setting the proper firewall or router configuration rules.

![Fig. 7: Possible scenarios of TCP communication over WAN with Discovery Server](image)

Regardless of whether the discovery *server* belongs, or not, to one of the clients’ LANs, it is necessary to configure one port forwarding rule for the discovery *server*, and another rule per every pair of clients communicating over the WAN, on either one of the sides.

If possible, deploy the different elements of the tutorial in different docker containers over WAN, following one of the possibilities displayed in the image above. In order to address the port forwarding configuration, see the *Configure transversal NAT on the network router* section from *WAN communication over TCP Fast DDS Router* tutorial for further information.

*Discovery server*
# run the server docker image with the XML configuration
docker run -td --name discovery_server --net host eprosima/vulcanexus:iron
docker cp <workspace>/server_configuration.xml discovery_server:/server_configuration.xml
docker exec -it discovery_server bash

# run the discovery server
source /opt/vulcanexus/iron/setup.bash
fastdds discovery -i 0 -x server_configuration.xml

Listener

# run the listener docker image with the XML configuration
docker run -td --name ros_listener --net host eprosima/vulcanexus:iron
docker cp <workspace>/listener_configuration.xml ros_listener:/listener_configuration.xml
docker exec -it ros_listener bash

# run the listener client
source /opt/ros/iron/setup.bash
export FASTRTPS_DEFAULT_PROFILES_FILE="listener_configuration.xml"
ros2 run demo_nodes_cpp listener

Talker

# run the talker docker image with the XML configuration
docker run -td --name ros_talker --net host eprosima/vulcanexus:iron
docker cp <workspace>/talker_configuration.xml ros_talker:/talker_configuration.xml
docker exec -it ros_talker bash

# run the talker client
source /opt/ros/iron/setup.bash
export FASTRTPS_DEFAULT_PROFILES_FILE="talker_configuration.xml"
ros2 run demo_nodes_cpp talker

Make sure that the discovery server public IP has been properly set in all sections of the XML configuration files (in both listener_configuration.xml, server_configuration.xml and talker_configuration.xml). The host public IP address can be obtained by running:

wget -qO- http://ipecho.net/plain | xargs echo

It is also required to set the WAN address for each client in the XML transport descriptor configuration. If any of them are hosted in the same LAN as the discovery server, then make sure that the XML transport descriptor listening ports configured for each of them are different. Also, take into account that the WAN IP address would be the same for both contexts (client and server).
Enabling multicast communication

- **Background**
- **Prerequisites**
- **XML Configuration**
- **Run the example**

**Background**

**Note:** This documentation assumes basic knowledge of UDP/TCP/IP concepts, namely unicast and multicast.

When communicating nodes across different hosts, Vulcanexus applications, by default, utilize UDPv4 based unicast communication with one another (between nodes on the same host, the shared memory transport is used by default, which defaults to multicast communications). This is because multicast based communication is not always a possibility in certain deployment for a myriad of reasons, among them multicast not performing well over certain WiFi networks or IT constrains. However, for deployments in which multicast is an available and reliable means of communication, Vulcanexus generated network traffic can be significantly reduced by leveraging multicast communications, which may also reduce Vulcanexus overhead in terms of CPU utilization and latency while increasing message throughput as the messages need to be copied to the network buffer but once (in opposition to unicast based communications, in which each message needs to be copied once per recipient).

This tutorial showcases how to enable multicast communication between Vulcanexus nodes communicating over a UDPv4 transport by *Configuring Fast-DDS QoS via XML profiles*.

**Prerequisites**

For accomplishing this tutorial, two available hosts with Docker and a Vulcanexus image are required. Please refer to the installation steps detailed in *Docker installation*.

**XML Configuration**

Save the following XML configuration file at the desired location, which will be referred as `<path_to_xml_config_file>` from here onwards.

**Important:** The Data Reader profile name shall match the topic’s Fully Qualified Name (FQN) (see *Creating publishers/subscribers with different profiles*).

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
  <profiles>
    <!-- Profile name shall match topic name -->
    <data_reader profile_name="/chatter"
      <!-- Set the chatter subscriptions listening address and port to a multicast -->
      locator
```

(continues on next page)
Run the example

This tutorial leverages the Vulcanexus iron-desktop Docker image to demonstrate the use of the aforementioned XML configuration file to achieve multicast communication across different hosts.

First, run two Vulcanexus iron-desktop with:

Container one (Subscription)
```bash
docker run \
  --interactive \ 
  --tty \ 
  --rm \ 
  --volume <path_to_xml_config_file_dir>:/root/xml_config \
  ubuntu-vulcanexus:iron-desktop
```

Container two (Publication)
```bash
docker run \
  --interactive \ 
  --tty \ 
  --rm \ 
  ubuntu-vulcanexus:iron-desktop
```

Then, within the container, run the talker-listener demo.

Container one (Subscription)
```bash
source /vulcanexus_entrypoint.sh 
export FASTRTPS_DEFAULT_PROFILES_FILE=/root/xml_config/<xml_config_file> 
ros2 run demo_nodes_cpp listener
```

Container two (Publication)
```bash
source /vulcanexus_entrypoint.sh 
ros2 run demo_nodes_cpp talker
```

Optionally, as shown in the following video, it is possible to use Wireshark to sniff the RTPS traffic and corroborate that the specified multicast address and port are indeed used:
4.11 Vulcanexus Tools Tutorials

4.11.1 ROS 2 network statistics inspection with ROS 2 Monitor

- **Background**
- **Prerequisites**
  - Launch ROS 2 Monitor
  - Execute ROS 2 demo nodes with statistics
  - Monitoring network
    - Alias
    - Physical data
    - Statistical data
    - Introspect metatraffic topics

**Background**

Vulcanexus integrates ROS 2 Monitor, which is a useful tool for monitoring and studying a ROS 2 network as ROS 2 relies on the DDS specification to communicate the different nodes. The automatic discovery of entities in a local network enables to easily identify the different running Participants, their Endpoints, the Topics that each of them is using, and even the network interfaces they are employing to communicate with one another. Additionally, it is possible to receive statistical data from every endpoint in the network leveraging the ROS 2 Statistics Module. This data is very useful to analyze the DDS network performance and seek possible communication problems in it.

This tutorial provides step-by-step instructions to use Vulcanexus for monitoring a ROS 2 talker/listener demo.

**Prerequisites**

Ensure that the Vulcanexus installation includes Vulcanexus Tools (either `vulcanexus-iron-desktop`, `vulcanexus-iron-tools`, or `vulcanexus-iron-base`). Also, remember to source the environment in every terminal in this tutorial.

```
source /opt/vulcanexus/iron/setup.bash
```

**Launch ROS 2 Monitor**

Initiate *ROS 2 Monitor* running the following command:

```
fastdds_monitor
```

Once *ROS 2 Monitor* is launched, start a monitor in domain 0 (default domain).
Execute ROS 2 demo nodes with statistics

In order to activate the publication of statistical data, eProsima Fast DDS requires an environment variable specifying which kinds of statistical data are to be reported. Consequently, before launching the ROS 2 nodes, remember to set FASTDDS_STATISTICS environment variable. Run the following commands in different terminals (remember to source the Vulcanexus environment):

```bash
export FASTDDS_STATISTICS="HISTORY_LATENCY_TOPIC;NETWORK_LATENCY_TOPIC;PUBLICATION_THROUGHPUT_TOPIC;SUBSCRIPTION_THROUGHPUT_TOPIC;RTPS_SENT_TOPIC;RTPS_LOST_TOPIC;HEARTBEAT_COUNT_TOPIC;ACKNACK_COUNT_TOPIC;NACKFRAG_COUNT_TOPIC;GAP_COUNT_TOPIC;DATA_COUNT_TOPIC;RESENT_DATAS_TOPIC;SAMPLE_DATAS_TOPIC;PDP_PACKETS_TOPIC;EDP_PACKETS_TOPIC;DISCOVERY_TOPIC;PHYSICAL_DATA_TOPIC"
ros2 run demo_nodes_cpp listener

export FASTDDS_STATISTICS="HISTORY_LATENCY_TOPIC;NETWORK_LATENCY_TOPIC;PUBLICATION_THROUGHPUT_TOPIC;SUBSCRIPTION_THROUGHPUT_TOPIC;RTPS_SENT_TOPIC;RTPS_LOST_TOPIC;HEARTBEAT_COUNT_TOPIC;ACKNACK_COUNT_TOPIC;NACKFRAG_COUNT_TOPIC;GAP_COUNT_TOPIC;DATA_COUNT_TOPIC;RESENT_DATAS_TOPIC;SAMPLE_DATAS_TOPIC;PDP_PACKETS_TOPIC;EDP_PACKETS_TOPIC;DISCOVERY_TOPIC;PHYSICAL_DATA_TOPIC"
ros2 run demo_nodes_cpp talker
```
Monitoring network

Now, the two new Participants are visible in the ROS 2 Monitor’s DDS Panel.

![DDSGlobal

Alias

Participants in ROS 2 are named / by default. In order to differentiate them, it is possible to change the Participant’s aliases within the ROS 2 Monitor. In this case, the vulcanexus-iron-talker Participant would be the one with a writer, and the vulcanexus-iron-listener Participant would be the one with a reader.

Physical data

In order to see the information of the host and the physical context where every node is running, go to the Explorer Pane and activate the Physical Panel. There, the host, user and process of each node are displayed.

Statistical data

To show statistical data about the communication between the vulcanexus-iron-talker and the vulcanexus-iron-listener, follow the steps to create dynamic series chart.
Chapter 4. Contributing to the documentation
Introspect metatraffic topics

ROS 2 Monitor filters by default the topics used for sharing metatraffic, as well as the endpoints related to them, so users can inspect their network easily. These topics are the ones that ROS 2 uses for discovery and configuration purposes, such as `ros_discovery_info`, as well as those used by Fast DDS to report statistical data.

In order to see these topics in the monitor, click `View->Show Metatraffic` menu button. Now, these topics are shown in the logical panel. Furthermore, the Readers and Writers associated to them are now listed under their respective Participants.

4.11.2 Monitoring ROS 2 with Fast DDS Statistics Backend

- **Background**
- **Prerequisites**
- **Creating the monitor package and application**
  - Creating the application workspace
  - Writing the monitor application
  - Examining the code
  - CMakeLists.txt
- **Running the application**
- **Next steps**
Background

Vulcanexus integrates *Fast DDS Statistics Backend*, which is a useful tool for monitoring and studying a ROS 2 network since ROS 2 relies on the DDS specification to communicate the different nodes. This tool will be used to create a monitoring application tailored to the user's needs. Therefore, this tutorial provides a step by step tutorial on how to create your first application as a ROS 2 package to monitor your ROS 2 deployment.

This tutorial will use the `demo_nodes_cpp` package, available in the Vulcanexus Desktop distribution. First, a ROS 2 *talker* is launched and then a *listener* node is started in the same machine. At this point, the monitor application created using the *Fast DDS Statistics Backend* library will be deployed to record and present the overall communication performance.
Prerequisites

It is required to have previously installed Vulcanexus using one of the following installation methods:

- *Linux binary installation*
- *Linux installation from sources*
- *Docker installation*

Ensure that the Vulcanexus installation includes Vulcanexus Tools (either `vulcanexus-iron-desktop`, `vulcanexus-iron-tools`, or `vulcanexus-iron-base`).

Creating the monitor package and application

In this section the monitoring application is created from scratch, covering the creation of the working environment and the development of the source code.

Creating the application workspace

The application workspace will have the following structure at the end of the project.

```
ros2_ws
|-- src
   |-- monitor_tutorial
      |-- src
         |-- monitor.cpp
         |-- CMakeLists.txt
         |-- package.xml
```

Let’s create the ROS 2 workspace and package by running the following commands:

```
mkdir -p ros2_ws/src
cd ros2_ws/src
ros2 pkg create --build-type ament_cmake monitor_tutorial --dependencies fastcdr, fastrtps fastdds_statistics_backend
```

You will now have a new folder within your workspace `src` directory called `monitor_tutorial`.

Writing the monitor application

From the `ros_ws/src/monitor_tutorial/src` directory in the workspace, run the following command to download the `monitor.cpp` file.

```
wget -O monitor.cpp \
```

This is the C++ source code for the application. This source code can also be found [here](#).
```cpp
#include <chrono>
#include <iostream>
#include <iomanip>
#include <sstream>
#include <string>
#include <thread>
#include <vector>
#include <fastdds_statistics_backend/listener/DomainListener.hpp>
#include <fastdds_statistics_backend/StatisticsBackend.hpp>
#include <fastdds_statistics_backend/types/EntityId.hpp>
#include <fastdds_statistics_backend/types/types.hpp>

using namespace eprosima::statistics_backend;

class Monitor
{
public:

    Monitor(
        uint32_t domain,
        uint32_t n_bins,
        uint32_t t_interval)
        : domain_(domain)
          , n_bins_(n_bins)
          , t_interval_(t_interval)
          , monitor_id_(EntityId::invalid())
          , topic_id_(EntityId::invalid())
    {
    }

    ~Monitor()
    {
        StatisticsBackend::stop_monitor(monitor_id_);
    }

private:

    uint32_t domain_; // domain
    uint32_t n_bins_;  // number of bins
    uint32_t t_interval_; // time interval
    EntityId monitor_id_; // monitor ID
    EntityId topic_id_;   // topic ID
};
```
bool init()
{
    monitor_id_ = StatisticsBackend::init_monitor(domain_);
    if (!monitor_id_.is_valid())
    {
        std::cout << "Error creating monitor" << std::endl;
        return 1;
    }

    StatisticsBackend::set_physical_listener(&physical_listener_);

    return true;
}

void run()
{
    while (true)
    {
        if (!topic_id_.is_valid())
        {
            topic_id_ = get_topic_id("rt/chatter", "std_msgs::msg::dds::String_");
        }
        else
        {
            get_fastdds_latency_mean();
            get_publication_throughput_mean();
        }

        std::this_thread::sleep_for(std::chrono::milliseconds(t_interval_*1000));
    }
}

//! Get the id of the topic searching by topic_name and data_type
EntityId get_topic_id(
    std::string topic_name,
    std::string data_type)
{
    // Get the list of all topics available
    std::vector<EntityId> topics = StatisticsBackend::get_
        →entities(EntityKind::TOPIC);
    Info topic_info;
    // Iterate over all topic searching for the one with specified topic_name and data_type
    for (auto topic_id : topics)
    {
        topic_info = StatisticsBackend::get_info(topic_id);
        if (topic_info["name"] == topic_name && topic_info["data_type"] == data_type)
        {
            return topic_id;
        }
    }
}
return EntityId::invalid();
}

// Get communications latency mean
void get_fastdds_latency_mean()
{
    // Vector of Statistics Data to store the latency data
    std::vector<StatisticsData> latency_data{};

    // Publishers on a specific topic
    std::vector<EntityId> publishers = StatisticsBackend::get_entities(
        EntityKind::DATAWRITER,
        topic_id_);

    // Subscriptions on a specific topic
    std::vector<EntityId> subscriptions = StatisticsBackend::get_entities(
        EntityKind::DATAREADER,
        topic_id_);

    // Get current time
    std::chrono::system_clock::time_point now = std::chrono::system_clock::now();

    // Get the mean of the FASTDDS_LATENCY of the last time interval
    // between the Publishers and Subscriptions publishing in and subscribed to a
    // given topic
    latency_data = StatisticsBackend::get_data(
        DataKind::FASTDDS_LATENCY,
        publishers,                        // Source entities
        subscriptions,                    // Target entities
        n_bins_,                           // Number of bins
        now - std::chrono::seconds(t_interval_), // t_from
        now,                                // t_to
        StatisticKind::MEAN);              // Statistic

    // Iterate over the retrieve data
    for (auto latency : latency_data)
    {
        // Check if there are meaningful values in retrieved vector
        if (std::isnan(latency.second))
        {
            return;
        }

        // Print the latency data
        std::cout << "ROS 2 Latency in topic " << StatisticsBackend::get_info(topic_id_)["name"] << ": [" << timestamp_to_string(latency.first) << ", " << latency.second / 1000 << " s"] << std::endl;
    }
}
```cpp
void get_publication_throughput_mean()
{
    // Vector of Statistics Data to store the publication throughput data
    std::vector<StatisticsData> publication_throughput_data{};

    // Publishers on a specific topic
    std::vector<EntityId> publishers = StatisticsBackend::get_entities(
        EntityKind::DATAWRITER,
        topic_id_);

    // Get current time
    std::chrono::system_clock::time_point now = std::chrono::system_clock::now();

    /*
    * Get the mean of the PUBLICATION_THROUGHPUT of the last time interval
    * of the Publishers publishing in a given topic
    */
    publication_throughput_data = StatisticsBackend::get_data(
        DataKind::PUBLICATION_THROUGHPUT, // DataKind
        publishers, // Source entities
        n_bins_, // Number of bins
        now - std::chrono::seconds(t_interval_), // t_from
        now, // t_to
        StatisticKind::MEAN); // Statistic

    // Iterate over the retrieve data
    for (auto publication_throughput : publication_throughput_data)
    {
        // Check if there are meaningful values in retrieved vector
        if (std::isnan(publication_throughput.second))
        {
            return;
        }

        // Print the publication throughput data
        std::cout << "Publication throughput in topic " << StatisticsBackend::get_˓
        info(topic_id_)["name"] << " : [" ˓
        << timestamp_to_string(publication_throughput.first) << ", " ˓
        << publication_throughput.second << " B/s]" << std::endl;
    }

    // Convert timestamp to string
    std::string timestamp_to_string(const Timestamp timestamp)
    {
        auto timestamp_t = std::chrono::system_clock::to_time_t(timestamp);
        auto msec = std::chrono::duration_cast<std::chrono::milliseconds>(timestamp.time_˓
        - since_epoch()).count();
    }
}
```

(continues on next page)
msec %= 1000;
std::stringstream ss;

ss << std::put_time(localtime(&timestamp_t), "%F %T") << "." << std::setw(3) <<
   std::setfill('0') << msec;

return ss.str();
}

protected:

class Listener : public eprosima::statistics_backend::PhysicalListener
{
public:

Listener()
{
}

Listener() override
{
}

void on_host_discovery(
   EntityId host_id,
   const DomainListener::Status& status) override
{
   Info host_info = StatisticsBackend::get_info(host_id);

   if (status.current_count_change == 1)
   {
      std::cout << "Host " << std::string(host_info["name"])) << " discovered." << std::endl;
   }
   else
   {
      std::cout << "Host " << std::string(host_info["name"])) << " updated info."
   }
}

void on_user_discovery(
   EntityId user_id,
   const DomainListener::Status& status) override
{
   Info user_info = StatisticsBackend::get_info(user_id);

   if (status.current_count_change == 1)
   {
      std::cout << "User " << std::string(user_info["name"])) << " discovered." << std::endl;
   }
}
else
{
    std::cout << "User " << std::string(user_info["name"]) << " updated info."
    << std::endl;
}

void on_process_discovery(
    EntityId process_id,
    const DomainListener::Status& status) override
{
    Info process_info = StatisticsBackend::get_info(process_id);
    if (status.current_count_change == 1)
    {
        std::cout << "Process " << std::string(process_info["name"]) << " discovered."
        << std::endl;
    }
    else
    {
        std::cout << "Process " << std::string(process_info["name"]) << " updated info."
        << std::endl;
    }
}

void on_topic_discovery(
    EntityId domain_id,
    EntityId topic_id,
    const DomainListener::Status& status) override
{
    Info topic_info = StatisticsBackend::get_info(topic_id);
    Info domain_info = StatisticsBackend::get_info(domain_id);
    if (status.current_count_change == 1)
    {
        std::cout << "Topic " << std::string(topic_info["name"]) << " [" << std::string(topic_info["data_type"]) << "] discovered."
        << std::endl;
    }
    else
    {
        std::cout << "Topic " << std::string(topic_info["name"]) << " [" << std::string(topic_info["data_type"]) << "] updated info."
        << std::endl;
    }
}

void on_participant_discovery(
    EntityId domain_id,
    EntityId participant_id,
    const DomainListener::Status& status) override
{
static_cast<void>(domain_id);
Info participant_info = StatisticsBackend::get_info(participant_id);

if (status.current_count_change == 1)
{
    std::cout << "Participant with GUID " << std::string(participant_info["guid"]) << " discovered."
    << std::endl;
}
else
{
    std::cout << "Participant with GUID " << std::string(participant_info["guid"]) << " updated info."
    << std::endl;
}

void on_datareader_discovery(
    EntityId domain_id,
    EntityId datareader_id,
    const DomainListener::Status& status) override
{
    static_cast<void>(domain_id);
    Info datareader_info = StatisticsBackend::get_info(datareader_id);

    if (status.current_count_change == 1)
    {
        std::cout << "DataReader with GUID " << std::string(datareader_info["guid"]) << " discovered."
        << std::endl;
    }
    else
    {
        std::cout << "DataReader with GUID " << std::string(datareader_info["guid"]) << " updated info."
        << std::endl;
    }
}

void on_datawriter_discovery(
    EntityId domain_id,
    EntityId datawriter_id,
    const DomainListener::Status& status) override
{
    static_cast<void>(domain_id);
    Info datawriter_info = StatisticsBackend::get_info(datawriter_id);

    if (status.current_count_change == 1)
    {
        std::cout << "DataWriter with GUID " << std::string(datawriter_info["guid"]) << " discovered."
        << std::endl;
    }
    else
    {
        std::cout << "DataWriter with GUID " << std::string(datawriter_info["guid"]) << " updated info."
        << std::endl;
    }
}
Examining the code

At the beginning of the file, the Doxygen style comment block with the @file field states the name of the file.

```
/**
 * @file monitor.cpp
 *
 */
```

Below are the includes of the C++ headers that allow the use of Fast DDS and Fast DDS Statistics Backend API.

```
#include <fastdds_statistics_backend/listener/DomainListener.hpp>
#include <fastdds_statistics_backend/StatisticsBackend.hpp>
#include <fastdds_statistics_backend/types/EntityId.hpp>
#include <fastdds_statistics_backend/types/types.hpp>
```

Next, we define the namespace that contains the Fast DDS Statistics Backend classes and functions that we are going to use in our application.

```
using namespace eprosima::statistics_backend;
```

The next line creates the Monitor class that implements the monitor.
class Monitor

The public constructor and destructor of the Monitor class are defined below. The constructor initializes the protected data members of the class to the values passed as arguments. The class destructor stops the monitor.

```cpp
Monitor(
    uint32_t domain,
    uint32_t n_bins,
    uint32_t t_interval)
    : domain_(domain),
      n_bins_(n_bins),
      t_interval_(t_interval),
      monitor_id_(EntityId::invalid()),
      topic_id_(EntityId::invalid())
{
}

~Monitor()
{
    StatisticsBackend::stop_monitor(monitor_id_);
}
```

Continuing with the public member functions of the Monitor class, the next snippet of code defines the public initialization member function. This function performs several actions:

1. Initialize the monitor.
2. Check the monitor has a valid id in the Statistics Backend database.
3. Assign the physical listener to the Statistics Backend. This listener will capture any update in the discovery of DDS entities.

```cpp
bool init()
{
    monitor_id_ = StatisticsBackend::init_monitor(domain_);
    if (!monitor_id_.is_valid())
    {
        std::cout << "Error creating monitor" << std::endl;
        return false;
    }
    StatisticsBackend::set_physical_listener(&physical_listener_);
    return true;
}
```

To run the monitor, the public member function run() is implemented. This function searches for the rt/chatter topic in the Statistics Backend database by calling the get_topic_id() function. If this is found, we can proceed to compute the actual statistics data. In order to do so, it calls get_fastdds_latency_mean() and get_publication_throughput_mean() public member functions explained below.

```cpp
void run()
{
    while (true)
    {
        (continues on next page)
    }
```
if (!topic_id_.is_valid())
{
    topic_id_ = get_topic_id("rt/chatter", "std_msgs::msg::dds::String_");
}
else
{
    get_fastdds_latency_mean();
    get_publication_throughput_mean();
}
std::this_thread::sleep_for(std::chrono::milliseconds(t_interval_*1000));

As introduced before, the get_topic_id() public member function get the id of the topic searching by topic name and data type name.

```cpp
// Get the id of the topic searching by topic_name and data_type
EntityId get_topic_id(
    std::string topic_name,
    std::string data_type)
{
    // Get the list of all topics available
    std::vector<EntityId> topics = StatisticsBackend::get_
        →entities(EntityKind::TOPIC);
    Info topic_info;
    // Iterate over all topic searching for the one with specified topic_name and data_type
    for (auto topic_id : topics)
    {
        topic_info = StatisticsBackend::get_info(topic_id);
        if (topic_info["name"] == topic_name && topic_info["data_type"] == data_type)
        {
            return topic_id;
        }
    }
    return EntityId::invalid();
}
```

The public member function get_fastdds_latency_mean() gets the Fast DDS latency mean of the last t_interval seconds between the talker and the listener. To achieve this the function performs several actions:

2. Get the current time.
3. Get the mean of the FASTDDS_LATENCY of the last time interval between the Publishers and Subscriptions publishing under and subscribed to the given topic, rt/chatter in this case.

```cpp
// Get communications latency mean
void get_fastdds_latency_mean()
{
    (continues on next page)
}
```cpp
// Vector of Statistics Data to store the latency data
std::vector<StatisticsData> latency_data{};

// Publishers on a specific topic
std::vector<EntityId> publishers = StatisticsBackend::get_entities(
    EntityKind::DATAWRITER,
    topic_id_);

// Subscriptions on a specific topic
std::vector<EntityId> subscriptions = StatisticsBackend::get_entities(
    EntityKind::DATAREADER,
    topic_id_);

// Get current time
std::chrono::system_clock::time_point now = std::chrono::system_clock::now();

/*
 * Get the mean of the FASTDDS_LATENCY of the last time interval
 * between the Publishers and Subscriptions publishing in and subscribed to a,
 * given topic
 */
latency_data = StatisticsBackend::get_data(
    DataKind::FASTDDS_LATENCY, // DataKind
    publishers, // Source entities
    subscriptions, // Target entities
    n_bins_, // Number of bins
    now - std::chrono::seconds(t_interval_), // t_from
    now, // t_to
    StatisticKind::MEAN); // Statistic

// Iterate over the retrieve data
for (auto latency : latency_data)
{
    // Check if there are meaningful values in retrieved vector
    if (std::isnan(latency.second))
    {
        return;
    }

    // Print the latency data
    std::cout << "ROS 2 Latency in topic " << StatisticsBackend::get_info(topic_id_)["name"] << ": [" <<
        timestamp_to_string(latency.first) << ", " << latency.second / 1000 << "]" << std::endl;
}
```

Finally, the public member function `get_publication_throughput_mean()` gets the publication throughput mean of the last `t_interval` seconds of the talker. The function has a similar execution procedure than the previous one but in this case it queries the mean of the `PUBLICATION_THROUGHPUT` instead of the `FASTDDS_LATENCY`.

```cpp
//! Get publication throughput mean
```
void get_publication_throughput_mean()
{
    // Vector of Statistics Data to store the publication throughput data
    std::vector<StatisticsData> publication_throughput_data{};

    // Publishers on a specific topic
    std::vector<EntityId> publishers = StatisticsBackend::get_entities(
        EntityKind::DATAWRITER,
        topic_id_);

    // Get current time
    std::chrono::system_clock::time_point now = std::chrono::system_clock::now();

    /*
    * Get the mean of the PUBLICATION_THROUGHPUT of the last time interval
    * of the Publishers publishing in a given topic
    */
    publication_throughput_data = StatisticsBackend::get_data(
        DataKind::PUBLICATION_THROUGHPUT, // DataKind
        publishers, // Source entities
        n_bins_, // Number of bins
        now - std::chrono::seconds(t_interval_), // t_from
        now, // t_to
        StatisticKind::MEAN); // Statistic

    // Iterate over the retrieve data
    for (auto publication_throughput : publication_throughput_data)
    {
        // Check if there are meaningful values in retrieved vector
        if (std::isnan(publication_throughput.second))
        {
            return;
        }

        // Print the publication throughput data
        std::cout << "Publication throughput in topic " << StatisticsBackend::get_
            →info(topic_id_)["name"] << ": [" << timestamp_to_string(publication_throughput.first) << ", "
            << publication_throughput.second << " B/s]" << std::endl;
    }
}

Then, the protected Listener class is defined by inheriting from the PhysicalListener class. This class overrides the default PhysicalListener callbacks, which allows the execution of routines in case of an event.

class Listener : public eprosima::statistics_backend::PhysicalListener
{
    public:

        Listener()
        {

        }
}
Within the PhysicalListener class, we can override several callback to adapt how the monitor application reacts to some events. These overridden callbacks are:

- **on_host_discovery()** allows the definition of a series of actions when a new host is detected.

```cpp
void on_host_discovery(
    EntityId host_id,
    const DomainListener::Status& status) override
{
    Info host_info = StatisticsBackend::get_info(host_id);

    if (status.current_count_change == 1)
    {
        std::cout << "Host " << std::string(host_info["name"]) << " discovered." << std::endl;
    }
    else
    {
        std::cout << "Host " << std::string(host_info["name"]) << " updated." << std::endl;
    }
}
```

- **on_user_discovery()** detects when a new user is discovered.

```cpp
void on_user_discovery(
    EntityId user_id,
    const DomainListener::Status& status) override
{
    Info user_info = StatisticsBackend::get_info(user_id);

    if (status.current_count_change == 1)
    {
        std::cout << "User " << std::string(user_info["name"]) << " discovered." << std::endl;
    }
    else
    {
        std::cout << "User " << std::string(user_info["name"]) << " updated." << std::endl;
    }
}
```

- **on_process_discovery()** involves when a new process is discovered.

```cpp
void on_process_discovery(
    EntityId process_id,
    const DomainListener::Status& status) override
{
}
```
\{  
    Info process_info = StatisticsBackend::get_info(process_id);

    if (status.current_count_change == 1) {  
        std::cout << "Process " << std::string(process_info["name"]) << " discovered." << std::endl;
    } else {  
        std::cout << "Process " << std::string(process_info["name"]) << " updated info." << std::endl;
    }
\}

- **on_topic_discovery()** is called when a new Topic is discovered.

```
void on_topic_discovery(  
    EntityId domain_id,
    EntityId topic_id,
    const DomainListener::Status& status) override  
{  
    Info topic_info = StatisticsBackend::get_info(topic_id);
    Info domain_info = StatisticsBackend::get_info(domain_id);

    if (status.current_count_change == 1) {  
        std::cout << "Topic " << std::string(topic_info["name"]) << " [" << std::string(topic_info["data_type"]) << "] discovered." << std::endl;
    } else {  
        std::cout << "Topic " << std::string(topic_info["name"]) << " [" << std::string(topic_info["data_type"]) << "] updated info." << std::endl;
    }
}
```

- **on_participant_discovery()** is called when a new participant is discovered.

```
void on_participant_discovery(  
    EntityId domain_id,
    EntityId participant_id,
    const DomainListener::Status& status) override  
{  
    static_cast<void>(domain_id);
    Info participant_info = StatisticsBackend::get_info(participant_id);

    if (status.current_count_change == 1) {  
        std::cout << "Participant with GUID " << std::string(participant_info["guid"]) << " discovered." << std::endl;
    }
}
```

(continues on next page)
• on_datareader_discovery() and on_datawriter_discovery() involves when a new DataReader or DataWriter respectively are discovered.

```cpp
void on_datareader_discovery(
    EntityId domain_id,
    EntityId datareader_id,
    const DomainListener::Status& status) override
{
    static_cast<void>(domain_id);
    Info datareader_info = StatisticsBackend::get_info(datareader_id);

    if (status.current_count_change == 1)
    {
        std::cout << "DataReader with GUID " << std::string(datareader_info->"guid")["guid"] << " discovered." << std::endl;
    }
    else
    {
        std::cout << "DataReader with GUID " << std::string(datareader_info->"guid")["guid"] << " updated info." << std::endl;
    }
}

void on_datawriter_discovery(
    EntityId domain_id,
    EntityId datawriter_id,
    const DomainListener::Status& status) override
{
    static_cast<void>(domain_id);
    Info datawriter_info = StatisticsBackend::get_info(datawriter_id);

    if (status.current_count_change == 1)
    {
        std::cout << "DataWriter with GUID " << std::string(datawriter_info->"guid")["guid"] << " discovered." << std::endl;
    }
    else
    {
        std::cout << "DataWriter with GUID " << std::string(datawriter_info->"guid")["guid"] << " updated info." << std::endl;
    }
}
```

Finally, the monitor application is initialized and run in main function.
```c
int main()
{
    int domain = 0;
    int n_bins = 1;
    int t_interval = 5;

    Monitor monitor(domain, n_bins, t_interval);

    if (monitor.init())
    {
        monitor.run();
    }
    return 0;
}
```

**CMakeLists.txt**

Include at the end of the `CMakeList.txt` file you created earlier the following code snippet. This adds all the source files needed to build the executable, and links the executable and the library together.

```cmake
# Compile executable and link with dependencies
add_executable(${PROJECT_NAME} src/monitor.cpp)
# Do not use ament as dependencies are not linked with ament
target_link_libraries(${PROJECT_NAME} fastrtps fastcdr fastdds_statistics_backend)

# Install executable
install(
    TARGETS ${PROJECT_NAME}
    DESTINATION lib/${PROJECT_NAME}
)
```

This file can also be downloaded with this command in `ros_ws/src/monitor_tutorial` directory:

```
wget -O CMakeList.txt \  
˓CMakeLists.txt
```

**Running the application**

At this point the project is ready for building, compiling and running the application. From the base workspace directory (`ros_ws`), run the following commands.

```
colcon build
source install/setup.bash
ros2 run monitor_tutorial monitor_tutorial
```

Then open two more terminals and load the Vulcanexus environment. Then, in one of them run atalker and in the other one a listener of the `demo_nodes_cpp` ROS 2 package, available in the Vulcanexus Desktop distribution.

- Terminal 1:
source /opt/vulcanexus/iron/setup.bash
export FASTDDS_STATISTICS="HISTORY_LATENCY_TOPIC;PUBLICATION_THROUGHPUT_TOPIC;PHYSICAL_DATA_TOPIC"
ros2 run demo_nodes_cpp talker

• Terminal 2:
source /opt/vulcanexus/iron/setup.bash
export FASTDDS_STATISTICS="HISTORY_LATENCY_TOPIC;PUBLICATION_THROUGHPUT_TOPIC;PHYSICAL_DATA_TOPIC"
ros2 run demo_nodes_cpp listener

Note: In order to monitor other statistics topics, add them to environment variable FASTDDS_STATISTICS. Check the statistics topics available in the Fast DDS Documentation.

You should be able to see something similar to the following image.

Next steps
Now you can develop more functionalities in your application, such as collecting more performance data or monitoring other topics. You can check also this tutorial explaining how to connect an application developed with the Fast DDS Statistics Backend to a visualization tool like Grafana.

For more information about Fast DDS Statistics Backend features please refer to the project’s documentation.
4.11.3 Fast DDS Statistics Backend Monitoring to SQL

- **Background**
- **Prerequisites**
- **Resultant Database**
  - Creating the monitor package and application
    - Creating the application workspace
    - Writing the monitor application
    - Examining the code
    - CMakeLists.txt
- **Running the application**
- **Next steps**

**Background**

Vulcanexus integrates Fast DDS Statistics Backend, which is a useful tool for monitoring and studying a ROS 2 network since ROS 2 relies on the DDS specification to communicate the different nodes. The interaction with this tool is through a C++ API, which could be leveraged to create powerful monitoring tools developed by the user. In this tutorial we show how to create an application consisting of a Fast DDS Statistics Backend connected with a SQL in-disk database.

![Diagram](image)

**Note:** This tutorial assumes the reader has already reviewed previous tutorial and understands how Fast DDS Statistics Backend works and how to interact with it.

Within this tutorial we explain how to create an application using the Fast DDS Statistics Backend to store instrumentation data. The final application created will store latency and throughput data of chatter topic in an in-disk relational SQL database.
Prerequisites

It is required to have previously installed Vulcanexus using one of the following installation methods:

- *Linux binary installation*
- *Linux installation from sources*
- *Docker installation*

Ensure that the Vulcanexus installation includes Vulcanexus Tools (either `vulcanexus-iron-desktop`, `vulcanexus-iron-tools`, or `vulcanexus-iron-base`).

**Note:** This tutorial uses SQLite3 as SQL library to connect with an in-disk database. This SQLite3 is already installed in Vulcanexus environment.

Resultant Database

This tutorial executable `monitor_sql_tutorial` produces a database stored with name `vulcanexus_monitor.db` in the workspace where launched. This database contains one table called `data` with 3 columns:

- `timestamp [key]` Time since linux based time in milliseconds
- `latency_median` Median of Latency in the interval `timestamp - 5000 : timestamp` in nanoseconds.
- `throughput_mean` Median of Throughput in the interval `timestamp - 5000 : timestamp` in MB/second.

Every 5000 ms the program calls the *Statistics Backend* API and stores the results for latency median and throughput mean for all Nodes using `chatter` topic. The `timestamp` column is the key of the table as it cannot be repeated. It is stored as a number and not as string or timestamp to simplify the tutorial.

There exists a useful browser application to visualize the data inside a database file: [http://inloop.github.io/sqlite-viewer/](http://inloop.github.io/sqlite-viewer/). The resultant database should look similar to the following one:

<table>
<thead>
<tr>
<th>timestamp</th>
<th>latency_median</th>
<th>throughput_mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>167145504994</td>
<td>265357</td>
<td>24.992413</td>
</tr>
<tr>
<td>1671455046002</td>
<td>254266.5</td>
<td>25.000411</td>
</tr>
</tbody>
</table>

**Warning:** It is possible that some data is not available because it is not being published from the entities. In these cases *Statistics Backend* returns NaN, which is parsed as a 0 when inserted in the database to avoid format issues.

Creating the monitor package and application

This section explains the source code required to implement this tutorial. However, some code is reused from a previous tutorial and will not be repeated here.
Creating the application workspace

The application workspace will have the following structure at the end of the tutorial.

```
ros2_ws
  └ src
    └ monitor_sql_tutorial
        └ src
            └ sql_monitor.cpp
            └ CMakeLists.txt
            └ package.xml
```

Let's create the ROS 2 workspace and package by running the following commands:

```
mkdir -p ros2_ws/src
cd ros2_ws/src
ros2 pkg create --build-type ament_cmake monitor_sql_tutorial --dependencies fastcdr...
```

You should now see a new folder within your workspace `src` directory called `monitor_sql_tutorial`. This command also creates an include folder that is not needed for this tutorial.

Writing the monitor application

From the `ros_ws/src/monitor_sql_tutorial/src` directory in the workspace, run the following command to download the `sql_monitor.cpp` file.

```
    src/sql_monitor.cpp
```

This is the C++ source code for the application. This source code can also be found here.

```cpp
// Copyright 2022 Proyectos y Sistemas de Mantenimiento SL (eProsima).
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

/**
 * @file sql_monitor.cpp
 *
 */

#include <chrono>
```
```cpp
#include <iostream>
#include <iomanip>
#include <sstream>
#include <string>
#include <thread>
#include <vector>
#include <sqlite3.h>
#include <fastdds_statistics_backend/listener/DomainListener.hpp>
#include <fastdds_statistics_backend/StatisticsBackend.hpp>
#include <fastdds_statistics_backend/types/EntityId.hpp>
#include <fastdds_statistics_backend/types/types.hpp>
#include <fastdds_statistics_backend/exception/Exception.hpp>

using namespace eprosima::statistics_backend;

constexpr const char* DB_FILENAME = "vulcanexus_monitor.db";
constexpr uint32_t DOMAIN = 0;
constexpr uint32_t INTERVAL_MS = 5000;
constexpr const char* TOPIC_NAME = "rt/chatter";
constexpr const char* DATA_TYPE_NAME = "std_msgs::msg::dds::String_";

class Monitor
{
public:

    Monitor()
        : monitor_id_(EntityId::invalid())
        , topic_id_(EntityId::invalid())
        , database_(nullptr)
    {
        // Initialize sqlite database
        open_or_create_database();
        if (!database_)
        {
            throw Error("Error initializing database.");
        }

        // Initialize Monitor
        monitor_id_ = StatisticsBackend::init_monitor(DOMAIN);
        if (!monitor_id_.is_valid())
        {
            throw Error("Error initializing monitor.");
        }

        std::cout << "Backend to SQL running." << std::endl;
    }

    ~Monitor()
    {
    }
};
```
StatisticsBackend::stop_monitor(monitor_id_);
close_database();
}

// void run()
void run()
{
    while (true)
    {
        if (!topic_id_.is_valid())
        {
            // If topic still not exist, do nothing
            topic_id_ = get_topic_id(TOPIC_NAME, DATA_TYPE_NAME);
            std::cout << "Topic " << TOPIC_NAME << " does not exist yet." << std::endl;
        }
        else
        {
            // If it exist, store data in database
            store_to_db();
            std::this_thread::sleep_for(std::chrono::milliseconds(INTERVAL_MS));
        }
    }
}

// Get the id of the topic searching by topic_name and data_type
EntityId get_topic_id(
    std::string topic_name,
    std::string data_type)
{
    // Get the list of all topics available
    std::vector<EntityId> topics = StatisticsBackend::get_entities(EntityKind::TOPIC);
    Info topic_info;
    // Iterate over all topic searching for the one with specified topic_name and data_type
    for (auto topic_id : topics)
    {
        topic_info = StatisticsBackend::get_info(topic_id);
        if (topic_info["name"] == topic_name && topic_info["data_type"] == data_type)
        {
            return topic_id;
        }
    }
    return EntityId::invalid();
}

// Get communications latency median
StatisticsData get_latency_data()
{
    // Publishers on a specific topic
    std::vector<EntityId> publishers = StatisticsBackend::get_entities(
        EntityKind::DATAWRITER,
// Subscriptions on a specific topic
std::vector<EntityId> subscriptions = StatisticsBackend::get_entities(
    EntityKind::DATAREADER,
    topic_id_);

// Get current time
std::chrono::system_clock::time_point now = std::chrono::system_clock::now();

/*
* Get the median of the FASTDDS_LATENCY of the last time interval
* between the Publishers and Subscriptions publishing in and subscribed to a
given topic
*/
auto data = StatisticsBackend::get_data(
    DataKind::FASTDDS_LATENCY, // DataKind
    publishers, // Source entities
    subscriptions, // Target entities
    1, // Number of bins
    now - std::chrono::milliseconds(INTERVAL_MS), // t_from
    now, // t_to
    StatisticKind::MEDIAN); // Statistic

// There is only one value, check it is not nan
return data[0];

//! Get publication throughput mean
StatisticsData get_publication_throughput_mean()
{
    // Publishers on a specific topic
    std::vector<EntityId> publishers = StatisticsBackend::get_entities(
        EntityKind::DATAWRITER,
        topic_id_);

    // Get current time
    std::chrono::system_clock::time_point now = std::chrono::system_clock::now();

    /*
    * Get the mean of the PUBLICATION_THROUGHPUT of the last time interval
    * of the Publishers publishing in a given topic
    */
    auto data = StatisticsBackend::get_data(
        DataKind::PUBLICATION_THROUGHPUT, // DataKind
        publishers, // Source entities
        1, // Number of bins
        now - std::chrono::milliseconds(INTERVAL_MS), // t_from
        now, // t_to
        StatisticKind::MEAN); // Statistic

    // There is only one value, check it is not nan
    (continues on next page)
return data[0];
}

void open_or_create_database()
{
    // Open database
    int flags = SQLITE_OPEN_READWRITE | SQLITE_OPEN_CREATE | SQLITE_OPEN_FULLMUTEX |
    SQLITE_OPEN_SHAREDCACHE;
    if (sqlite3_open_v2(DB_FILENAME, &database_, flags, 0) != SQLITE_OK)
    {
        std::cerr << "Error opening or creating database." << std::endl;
        sqlite3_close(database_);
        database_ = nullptr;
        return;
    }

    // Create table if it does not exist
    std::cout << "Creating table with query: \"" << create_table_statement_ << \
    "\"."
    if (sqlite3_exec(database_, create_table_statement_, 0, 0, 0) != SQLITE_OK)
    {
        std::cerr << "Error creating table." << std::endl;
        sqlite3_close(database_);
        database_ = nullptr;
        return;
    }
}

void close_database()
{
    sqlite3_close(database_);
}

void store_to_db()
{
    // Get data
    auto latency_data = get_latency_data();
    auto throughput_data = get_publication_throughput_mean();

    // Parse the query to add values. Use only latency timestamp as both would be
    // almost the same
    sprintf(
        query_,
        insert_data_query_,
        std::chrono::duration_cast<std::chrono::milliseconds>(latency_data.first.
        time_since_epoch()).count(),
        (std::isnan(latency_data.second) ? 0 : latency_data.second),
        (std::isnan(throughput_data.second) ? 0 : throughput_data.second));

    std::cout << "Storing data in SQLite DataBase with query: \"" << query_ << \n    "\"."
    << std::endl;
}
// Insert in database (if fails, do nothing)
if(sqlite3_exec(database_, query_, 0, 0, 0) != SQLITE_OK)
{
    std::cerr << "An error occurred inserting new data: " << sqlite3_errno(database_) << std::endl;
}

protected:

    static constexpr const char* create_table_statement_ =
        "CREATE TABLE IF NOT EXISTS data(
            timestamp BIGINT,
            latency_median FLOAT,
            throughput_mean FLOAT,
            PRIMARY KEY(timestamp)
        ) WITHOUT ROWID;"
;

    static constexpr const char* insert_data_query_ =
        "INSERT INTO data (timestamp, latency_median, throughput_mean) VALUES(%lu, %f, %f);";

    char query_[200];

    DomainId domain_;  
    uint32_t t_interval_;  
    EntityId monitor_id_;  
    EntityId topic_id_;  
    sqlite3* database_;  
};

int main()
{
    // Create Monitor instance
    Monitor monitor;

    // Start Monitor
    monitor.run();

    return 0;
}
Examine the code

Before declaring the main class, there are some definitions that could be changed before compiling.

```cpp
constexpr const char* DB_FILENAME = "vulcanexus_monitor.db";
constexpr uint32_t DOMAIN = 0;
constexpr uint32_t INTERVAL_MS = 5000;
constexpr const char* TOPIC_NAME = "rt/chatter";
constexpr const char* DATA_TYPE_NAME = "std_msgs::msg::dds::String";
```

The main class called `Monitor` creates the database and initializes the monitor in construction.

```cpp
Monitor()
: monitor_id_(EntityId::invalid()),
  topic_id_(EntityId::invalid()),
  database_(nullptr)
{
  // Initialize sqlite database
  open_or_create_database();
  if (!database_)
  {
    throw Error("Error initializing database.");
  }

  // Initialize Monitor
  monitor_id_ = StatisticsBackend::init_monitor(DOMAIN);
  if (!monitor_id_.is_valid())
  {
    throw Error("Error initializing monitor.");
  }

  std::cout << "Backend to SQL running." << std::endl;
}
```

And it closes them in destruction.

```cpp
~Monitor()
{
  StatisticsBackend::stop_monitor(monitor_id_);
  close_database();
}
```

The routine of the instance is an infinite loop where, if the topic has already been discovered, it stores data in the database. This is similar to previous tutorial.

```cpp
void run()
{
  while (true)
  {
    if (!topic_id_.is_valid())
    {
      // If topic still not exist, do nothing
      topic_id_ = get_topic_id(TOPIC_NAME, DATA_TYPE_NAME);
    }
```
std::cout << "Topic " << TOPIC_NAME << " does not exist yet." << std::endl;
else
{
  // If it exist, store data in database
  store_to_db();
  std::this_thread::sleep_for(std::chrono::milliseconds(INTERVAL_MS));
}
}

The database is opened if exists, or created otherwise. It is initialized with the table data. In case opening or creating the table fails, the execution will finish. In exit, it closes the database.

```cpp
void open_or_create_database()
{
  // Open database
  int flags = SQLITE_OPEN_READWRITE | SQLITE_OPEN_CREATE | SQLITE_OPEN_FULLMUTEX | SQLITE_OPEN_SHAREDCACHE;
  if (sqlite3_open_v2(DB_FILENAME, &database_, flags, 0) != SQLITE_OK)
  {
    std::cerr << "Error opening or creating database." << std::endl;
    sqlite3_close(database_);
    database_ = nullptr;
    return;
  }

  // Create table if it does not exist
  std::cout << "Creating table with query: \"" << create_table_statement_ << "\"." << std::endl;
  if (sqlite3_exec(database_, create_table_statement_, 0, 0, 0) != SQLITE_OK)
  {
    std::cerr << "Error creating table." << std::endl;
    sqlite3_close(database_);
    database_ = nullptr;
    return;
  }
}
```

The routine to store new data in the database firstly calls the Statistics Bakend. Then it loads the data received in a query previously written. Finally it executes the query to insert the data.

```cpp
void store_to_db()
{
  // Get data
  auto latency_data = get_latency_data();
  auto throughput_data = get_publication_throughput_mean();

  // Parse the query to add values. Use only latency timestamp as both would be almost the same
  sprintf("\n```
```
The queries used to interact with the database are defined inside the class as static const variables.

```cpp
static constexpr const char* create_table_statement_ =
"CREATE TABLE IF NOT EXISTS data(
"timestamp BIGINT,
"latency_median FLOAT,
"throughput_mean FLOAT,
"PRIMARY KEY(timestamp)"
") WITHOUT ROWID;";

static constexpr const char* insert_data_query_ =
"INSERT INTO data (timestamp,latency_median,throughput_mean) VALUES(%lu,%f,%f);";
```

Finally, the monitor application is initialized and run in main function.

```cpp
int main()
{
    // Create Monitor instance
    Monitor monitor;

    // Start Monitor
    monitor.run();

    return 0;
}
```
CMakeLists.txt

Include at the end of the CMakeLists.txt file you created earlier the following code snippet. This adds all the source files needed to build the executable, and links the executable and the library together.

```
fnd_package(SQLite3 REQUIRED) # provided by sqlite3_vendor
# Compile executable and link with dependencies
add_executable(${PROJECT_NAME} src/sql_monitor.cpp)
# Do not use ament as dependencies are not linked with ament
target_link_libraries(${PROJECT_NAME} fastrtps fastcdr fastdds_statistics_backend sqlite3)
# Install executable
install(
  TARGETS ${PROJECT_NAME}
  DESTINATION lib/${PROJECT_NAME}
)
```

This file can also be downloaded with this command in ros_ws/src/monitor_sql_tutorial directory:

```
wget -O CMakeLists.txt \  
  https://raw.githubusercontent.com/eProsima/vulcanexus/main/code/monitor_sql_tutorial/  \  
  CMakeLists.txt
```

Running the application

At this point the project is ready for building, compiling and running the application. From the base workspace directory (ros_ws), run the following commands.

```
colcon build
source install/setup.bash
ros2 run monitor_sql_tutorial monitor_sql_tutorial
```

Now open two more terminals and load the Vulcanexus environment. Then, run a talker and a listener of the demo_nodes_cpp ROS 2 package, available in the Vulcanexus Desktop distribution, in a different terminal each.

- **Terminal 1:**
  
  ```
  source /opt/vulcanexus/iron/setup.bash
  export FASTDDS_STATISTICS="HISTORY_LATENCY_TOPIC;PUBLICATION_THROUGHPUT_TOPIC;PHYSICAL_DATA_TOPIC"
  ros2 run demo_nodes_cpp talker
  ```

- **Terminal 2:**
  
  ```
  source /opt/vulcanexus/iron/setup.bash
  export FASTDDS_STATISTICS="HISTORY_LATENCY_TOPIC;PUBLICATION_THROUGHPUT_TOPIC;PHYSICAL_DATA_TOPIC"
  ros2 run demo_nodes_cpp listener
  ```

**Note:** In order to monitor other statistics topics, add them to environment variable FASTDDS_STATISTICS. Check the statistics topics available in the Fast DDS Documentation.
Next steps

Now you can develop more functionalities in your application, such as collecting more performance data or monitoring other topics. You may also check this tutorial explaining how to connect an application developed with the Fast DDS Statistics Backend to a visualization tool like Grafana.

For more information about Fast DDS Statistics Backend features please refer to the project’s documentation.

4.11.4 Hand-on ROS 2 Shapes Demo

- Background
- Prerequisites
  - Launch ROS 2 Shapes Demo
- Learning DDS capabilities
  - Discovery and basic connectivity
  - History and Durability
  - Liveliness
  - Content Based Filter
- Next Steps

Background

Vulcanexus integrates ROS 2 Shapes Demo, which is a useful application for learning the capabilities of DDS or as a proof of eProsima Fast DDS interoperability with other DDS-compliant implementations.

Prerequisites

It is required to have previously installed Vulcanexus using one of the following installation methods:

- Linux binary installation
- Linux installation from sources
- Docker installation

Ensure that the Vulcanexus installation includes Vulcanexus Tools (either vulcanexus-iron-desktop, vulcanexus-iron-tools, or vulcanexus-iron-base).

Run Vulcanexus Docker image with:

```bash
xhost local:root
docker run
  -it
  --privileged
  -e DISPLAY=$DISPLAY
  -v /tmp/.X11-unix:/tmp/.X11-unix
  ubuntu-vulcanexus:iron-desktop
```
To run more than one session within the same container, *Vulcanexus* installation must be sourced. Given a running container, you can open another session by:

```
docker exec -it <running-container-id> bash
```

Then, within the container, source the *Vulcanexus* installation with:

```
source /opt/vulcanexus/iron/setup.bash
```

### Launch ROS 2 Shapes Demo

Initiate *ROS 2 Shapes Demo* running the following command:

```
shapesdemo
```

**Warning:** Please verify that you are running the ROS 2 Shapes Demo version. To check this simply verify that the Vulcanexus banner is displayed in the background of the main box. If the eProsima logo appears instead, activate the ROS 2 version in *Options > Participant Configuration > Use ROS 2 Topics*.

### Learning DDS capabilities

This tutorials present step-by-step instructions to test the functionalities provided by DDS thought the *ROS 2 Shapes Demo* application.

#### Discovery and basic connectivity

In *Fast DDS*, the discovery task is automatic. *Fast DDS* performs the task of finding the relevant information and distributing it to its destination. It means that new nodes are automatically discovered by any other in the network. Please refer to the *Fast DDS Discovery Documentation* for more information on the various *Fast DDS* discovery mechanisms.

In this test, three Publishers and three Subscribers are launched. At the end, two additional squares will be displayed in each window, reflecting the movements of the original square in real time. That is, subscribers subscribing to the “Square” topics are matched with the publishers of the other instances.

First, three publishers must be created.

1. Create a red square publisher:
   - Start *ROS 2 Shapes Demo* (this instance will be referred to as *Instance1*).
   - Click on Publish.
   - Select SQUARE option for Shape and RED for Color.

2. Create a blue square publisher:
   - Start *ROS 2 Shapes Demo* (this instance will be referred to as *Instance2*).
   - Click on Publish.
   - Select SQUARE option for Shape and BLUE for Color.

3. Create a black square publisher:
   - Start *ROS 2 Shapes Demo* (this instance will be referred to as *Instance3*).
• Click on Publish.
• Select SQUARE option for Shape and BLACK for Color.

The current setting should be similar to that shown in the figure below.

Then, three subscribers must be created.

1. Click Subscribe on Instance1.
   • Select SQUARE option for Shape.
   • Change the History field from 6 to 1.
2. Click Subscribe on Instance2.
   • Select SQUARE option for Shape.
   • Change the History field from 6 to 1.
3. Click Subscribe on Instance3.
   • Select SQUARE option for Shape.
   • Change the History field from 6 to 1.

The ROS 2 Shapes Demo windows should look similar to the following image.

**History and Durability**

A publisher can send messages throughout a Topic even if there are no DataReaders on the network. Moreover, a DataReader that joins to the Topic after some data has been written could be interested in accessing that information. The durability defines how the system will behave regarding those samples that existed on the Topic before the subscriber joins. Please refer to Fast DDS DurabilityQosPolicy Documentation for more information on Durability QoS.

In the following example, the publishers’ history is set to KEEP_LAST, and there are two options for the durability configuration which are VOLATILE and TRANSIENT_LOCAL. If VOLATILE is selected, the previous data samples will not be sent. However, if TRANSIENT_LOCAL is selected, the $n^{th}$ previous data samples will be sent to the late-joining subscriber.
In this example, one hundred red squares will be displayed in *Instance2* and *Instance3*, reflecting the movements of the red square of the publisher from *Instance1*. The leading square indicates the current position of the published square.

First, three instances are launched and a publisher is created in each of them:

1 - Create a red square publisher:
   - Start *ROS 2 Shapes Demo* (this instance will be referred to as *Instance1*).
   - Click on Publish.
   - Select SQUARE option for Shape and RED for Color.
   - Change the History field from 6 to 100.
   - Select TRANSIENT_LOCAL.

2 - Create an orange square publisher:
   - Start *ROS 2 Shapes Demo* (this instance will be referred to as *Instance2*).
   - Click on Publish.
   - Select SQUARE option for Shape and ORANGE for Color.
   - Change the History field from 6 to 100.
   - Select TRANSIENT_LOCAL.

3 - Create a black square publisher:
   - Start *ROS 2 Shapes Demo* (this instance will be referred to as *Instance3*).
   - Click on Publish.
   - Select SQUARE option for Shape and BLACK for Color.
   - Change the History field from 6 to 100.
   - Select TRANSIENT_LOCAL.

The *ROS 2 Shapes Demo* environment should look similar to the following figure.

Then, subscriber in each instance is created.
4. Click Subscribe on *Instance1*.
   - Select SQUARE option for Shape.
   - Change the History field from 6 to 100.

5. Click Subscribe on *Instance2*.
   - Select SQUARE option for Shape.
   - Change the History field from 6 to 100.

6. Click Subscribe on *Instance3*.
   - Select SQUARE option for Shape.
   - Change the History field from 6 to 100.

The *ROS 2 Shapes Demo* environment should look similar to the following figure.
Liveliness

The Liveliness QoS can be used to ensure whether specific entities are alive or not. There are three values to specify the liveliness’ kind: AUTOMATIC, MANUAL_BY_PARTICIPANT or MANUAL_BY_TOPIC liveliness. If any of the first two is selected, a value for the lease duration and announcement period can be set. However, if MANUAL_BY_TOPIC is selected, only the lease duration can be configured, as the announcement period is not used with this configuration. With the AUTOMATIC liveliness kind, the service takes the responsibility for renewing the timer associated to the lease duration, and as long as the remote participant keeps running and remains connected, all the entities within that participant will be considered alive. The other two kinds (MANUAL_BY_PARTICIPANT and MANUAL_BY_TOPIC) need a periodic assertion to consider the remote participants as alive. Please refer to Fast DDS LivelinessQosPolicy Documentation for more information on Liveliness QoS.

In this test, a publisher and subscriber using AUTOMATIC liveliness will be created, and a lease duration value higher than the write rate of the publisher will be set.

First, launch two instances and create a publisher and a subscriber:

1. Create a red square publisher:
   - Start ROS 2 Shapes Demo. (We will refer to this instance as Instance1)
   - Click on Publish.
   - Select SQUARE option for Shape and RED for Color.
   - Select AUTOMATIC for liveliness kind.
   - Set Lease Duration to 150. (The default write rate is 75 ms)
   - Set Announcement Period to 140.

2. Create a square subscriber:
   - Start ROS 2 Shapes Demo. (We will refer to this instance as Instance2)
   - Click on Subscribe.
   - Select SQUARE option for Shape.
   - Select AUTOMATIC for liveliness kind.
   - Set a value for the Lease Duration higher or equal to the one stated for the publisher. (If the value of subscriber lease duration is lower the entities do not match)

The Output Tab of Instance2 shows that the subscriber has recovered the liveliness once it matches with the publisher.

Then, kill the process corresponding to the publisher (Instance1). As a result, the subscriber reported that liveliness was lost, as the publisher did not terminate cleanly.

Content Based Filter

In Fast DDS, the data available to the subscriber can be restricted to control network and CPU usage. The Content Based Filter can be checked when a new subscriber is deployed. This filter draws a shaded region in the instance windows. Only the samples that are covered by the shade will be available to the subscriber. This region can be resized and moved dynamically.

In this test, two Publishers and two subscriber will be created, one of the latter with Content Based.

First, you have to launch two instances and create a Publisher in each of them:

1. Create a red square publisher:
   - Start ROS 2 Shapes Demo (this instance will be referred to as Instance1).
ROS 2 Shapes Demo (on 9Feb605ef34d)

Control   Options   Help

Publish

Subscribe

Endpoints   Output

[06:48:08.265]: Participant ready in domainId 0
[06:48:46.023]: Subscriber created in topic: rt/Square
[06:48:46.071]: DataReader 01.0f.eb.7d.bc.01.8a.4f.01.00.00.00] 0.1.4 detects liveliness recovered
[06:48:57.406]: DataReader 01.0f.eb.7d.bc.01.8a.4f.01.00.00.00] 0.1.4 detects liveliness lost
• Click on Publish.
• Select SQUARE option for Shape and RED for Color.
• Change the History field from 6 to 1.

2. Create an orange circle publisher:
   • Start ROS 2 Shapes Demo (this instance will be referred to as Instance2).
   • Click on Publish.
   • Select CIRCLE option for Shape and ORANGE for Color.
   • Change the History field from 6 to 1.

Your windows should look similar to the following image.

Then, create two subscribers:

3. Create a circle subscriber:
   • Start ROS 2 Shapes Demo (this instance will be referred to as Instance3).
   • Click on Subscribe.
   • Select CIRCLE option for Shape.
   • Change the History field from 6 to 1.
   • Check Content Based.

4. Create a square subscriber:
• Click on Subscribe in Instance3.
• Select SQUARE option for Shape.
• Change the History field from 6 to 1.

In the following figure, a shaded rectangle in Instance3 is shown. This is the filter for the samples of the Circle Shape. If the circle is out of the rectangle, it is not available for the subscriber.

However, if the instance is in the rectangle, it is available for the subscriber.

The rectangle is configurable, i.e. it can be resized and moved dynamically. The following images show examples of the content filter.
Next Steps

Visit ROS 2 Shapes Demo for more information on how to use this application.

4.11.5 ROS 2 network statistics inspection with Prometheus

- **Background**
- **Prerequisites**
  - Vulcanexus Tools
  - Prometheus exporter
  - Prometheus
  - Backend application
- **Execute ROS 2 demo nodes with statistics**
- **Launch backend application**
- **Visualizing with Prometheus**

**Background**

Vulcanexus integrates ROS 2 Monitor, which is a useful tool for monitoring and studying a ROS 2 network as ROS 2 relies on the DDS specification to communicate the different nodes. This other tutorial demonstrates how to use this powerful tool in a ROS 2 demo_nodes_cpp talker/listener scenario. However, Vulcanexus Tools offers more possibilities when it comes to statistics visualization: ROS 2 Monitor is a frontend application relying on Fast DDS Statistics Backend, but the latter may also be leveraged with alternative visualization utilities such as Prometheus and Grafana.

This tutorial provides step-by-step instructions on how to monitor a ROS 2 talker/listener in Vulcanexus by using Fast DDS Statistics Backend along with Prometheus.
Prerequisites

Vulcanexus Tools

Ensure that the Vulcanexus installation includes Vulcanexus Tools (either vulcanexus-iron-desktop, vulcanexus-iron-tools, or vulcanexus-iron-base). Also, remember to source the environment in every terminal in this tutorial.

```bash
source /opt/vulcanexus/iron/setup.bash
```

Prometheus exporter

Additionally, C++ Prometheus exporter is required to forward the statistical data received by the backend to Prometheus. Install this specific version of Prometheus exporter by following any of the methods described in the given link.

Prometheus

It is recommended (although not required) to install Prometheus for being able to visualize gathered statistics. If not installed, data will still be collected and visible in the terminal, but no graphs will be displayed. Follow the installation guide from the official website. The configuration file needed to replicate this demo can be found here.

Backend application

It only remains to install a simple application that, by making use of Fast DDS Statistics Backend, will collect statistical data and export it through the previously installed Prometheus exporter.

```bash
mkdir ~/Fast-DDS-statistics-backend && cd ~/Fast-DDS-statistics-backend
git clone https://github.com/eProsima/Fast-DDS-statistics-backend.git
cd ~/Fast-DDS-statistics-backend/examples/cpp/ROS2Prometheus
mkdir build && cd build
cmake .. && cd build
```
Execute ROS 2 demo nodes with statistics

In order to activate the publication of statistical data, eProsima Fast DDS requires an environment variable specifying which kinds of statistical data are to be reported. Consequently, before launching the ROS 2 nodes, remember to set FASTDDS_STATISTICS environment variable. Run the following commands in different terminals (remember to source the Vulcanexus environment):

```
export FASTDDS_STATISTICS="HISTORY_LATENCY_TOPIC;PUBLICATION_THROUGHPUT_TOPIC;PHYSICAL_DATA_TOPIC"
ros2 run demo_nodes_cpp listener

export FASTDDS_STATISTICS="HISTORY_LATENCY_TOPIC;PUBLICATION_THROUGHPUT_TOPIC;PHYSICAL_DATA_TOPIC"
ros2 run demo_nodes_cpp talker
```

Launch backend application

Once communication between the talker and listener has been established, launch the application for gathering and exporting statistical data to Prometheus.

```
cd ~/Fast-DDS-statistics-backend/examples/cpp/ROS2Prometheus/build
./ROS2Prometheus
```

Traces showing the latency and publication throughput should be visible on screen, with an update period of 5 seconds.
Visualizing with Prometheus

Launch Prometheus with the given configuration file and create a graph for each of the metrics exported (fastdds_latency and publication_throughput).

```
cd prometheus-install-path
./prometheus --config.file=prometheus.yml
xdg-open http://localhost:9090
```

**Note:** If running Vulcanexus in Docker containers, and if Prometheus has been installed in the host’s system, it is required to pass `--net=host` argument to the backend application container in order to enable communication between them.

Prometheus is a versatile open source monitoring system offering seamless interoperability with other analytics and visualization applications such as Grafana.

Feel free to further explore the number of possibilities that Vulcanexus Tools and Prometheus together have to offer.

### 4.12 Vulcanexus Cloud Tutorials

This tutorials aims to show the user how to recreate a robot deployment in a distributed network with the capability of monitoring, control and tracking through the Cloud. This way, a series of tutorials are presented that intend to build a complex final architecture step by step, explaining in detail the different steps of the process and the configuration of the different Vulcanexus nodes and WAN communication enabling components.

The following are the tutorials that make up this course on communication in distributed scenarios using the Vulcanexus framework.
4.12.1 Change ROS 2 Domain Id

- **Background**
- **Prerequisites**
- **Deploy ROS 2 nodes**
  - Environment setup
  - Running ROS 2 nodes
- **Deploy ROS 2 Router**
  - Running ROS 2 Router
- **Communicating multiple Domains**

**Background**

eProsima ROS 2 Router, a.k.a DDS Router, is an end-user software application that enables the connection of distributed ROS 2 networks (see ROS 2 Router documentation [here](#)). That is, ROS 2 nodes such as publishers and subscriptions, or clients and services, deployed in one geographic location and using a dedicated local network will be able to communicate with other ROS 2 nodes deployed in different geographic areas on their own dedicated local networks as if they were all on the same network through the use of ROS 2 Router.

This tutorial explains how to use the ROS 2 Router to communicate ROS 2 nodes in different Domain Ids. The DDS protocol define Domain Id as a parameter for every DomainParticipant. Different entities in different Domain Ids will never discover each other, and thus they will not communicate to each other. Using the ROS 2 Router as a bridge between ROS 2 Domains, every node will be able to communicate with any other node independently of the Domain where they are deployed.

As already mentioned, the approach of this tutorial is straightforward and is illustrated in the following figure:
This tutorial will use the `demo_nodes_cpp` package, available in the Vulcanexus Desktop distribution. First, a ROS 2 talker is launched and then a listener node is started in a different ROS 2 Domain. This will prevent the two from communicating. At this point, the ROS 2 Router will be deployed as a bridge between the two Domains and will enable the talker-listener communication.

**Prerequisites**

It is required to have previously installed Vulcanexus using one of the following installation methods:

- *Linux binary installation*
- *Linux installation from sources*
- *Docker installation*

**Deploy ROS 2 nodes**

First let’s run the ROS 2 talker and listener nodes.

**Environment setup**

Setup the Vulcanexus environment to have the `demo_nodes_cpp` package available. For this, there are two possible options:

1. Running the Vulcanexus Docker image.
   
   Run the Vulcanexus Docker image with:
   
   ```
   docker run -it ubuntu-vulcanexus:iron-desktop
   ```
   
   Then, within the container, source the Vulcanexus installation with:
   
   ```
   source /opt/vulcanexus/iron/setup.bash
   ```

2. Setting up the development environment on the local host. For this second option, it is necessary to have installed the `vucanexus-iron-desktop` package, since this is the one that includes all the simulation tools, demos and tutorials.

   Source the following file to setup the Vulcanexus environment:
   
   ```
   source /opt/vulcanexus/iron/setup.bash
   ```
Running ROS 2 nodes

Once the environment has been setup using one of the above options, run the ROS 2 talker node in one terminal.

```
ros2 run demo_nodes_cpp talker
```

Then, on another terminal, run the ROS 2 listener node in ROS 2 Domain 1.

```
ROS_DOMAIN_ID=1 ros2 run demo_nodes_cpp listener
```

At this point, the listener should not receive any data. If not, please go again through the previous steps.

Deploy ROS 2 Router

Then, create the ROS 2 Router configuration file as the one shown below.

```
version: v3.0

participants:

- name: ROS_2_Domain_0
  kind: local
  domain: 0

- name: ROS_2_Domain_1
  kind: local
  domain: 1
```

This configuration defines 2 different Router Participants, internal “interfaces” for the ROS 2 Router. Each of this Participants will create DDS Entities in each of the domains, and they will forward all the data received from one Domain to the other. Topics, Data Types, Quality of Services and order of messages will be respected when redirecting the data.

Running ROS 2 Router

Now, run the ROS 2 Router with the configuration file created as an argument.

```
ddsrouter --config-path <path/to/file>/change_domain.yaml
```

At this point you should see some information like the one shown below. This indicates that the ROS 2 Router has started correctly and it is currently running.
Starting DDS Router Tool execution.
DDS Router running.

In order to close the execution, just press ^C or send a signal (SIGINT 2 or SIGTERM 15) to close it.

**Communicating multiple Domains**

The *ROS 2 Router* can equally inter-communicate 2 or more Domain Ids. Just add as many Participants as desired to the configuration file and this will redirect all messages from every Domain to all the others. In the following figure we could see the use case and the configuration required for communicating 4 different Domains.

```
version: v3.0

participants:
  - name: ROS_2_Domain_0
    kind: local
    domain: 0
  - name: ROS_2_Domain_1
    kind: local
    domain: 1
  - name: Other_ROS_2_Participant
    kind: local
    domain: 2
  - name: And_One_More
    kind: local
    domain: 33
```
4.12.2 Change ROS 2 Domain to Discovery Server

- **Background**
  - DDS Discovery Mechanisms

- **Prerequisites**
  - Deploy ROS 2 nodes
    - Environment setup
    - Running ROS 2 Discovery Server
    - Running ROS 2 nodes
  - Deploy ROS 2 Router
    - Running ROS 2 Router
  - Alternative deployment: ROS 2 Router as DS Server

**Background**

*eProsima ROS 2 Router*, a.k.a DDS Router, is an end-user software application that enables the connection of distributed ROS 2 networks (see [ROS 2 Router documentation here](#)). That is, ROS 2 nodes such as publishers and subscriptions, or clients and services, deployed in one geographic location and using a dedicated local network will be able to communicate with other ROS 2 nodes deployed in different geographic areas on their own dedicated local networks as if they were all on the same network through the use of *ROS 2 Router*.

This tutorial explains how to connect ROS 2 nodes using *Simple Discovery* with other nodes that are connected via *Discovery Server*. These entities are unable to discover each other if using different discovery mechanisms, and thus they will not communicate to each other. Using the *ROS 2 Router* as a bridge between these 2 discovery mechanisms, every node will be able to communicate with any other node independently of the Domain where they are deployed or the Discovery Server to which they connect.

As already mentioned, the approach of this tutorial is straightforward and is illustrated in the following figure:

This tutorial will use the demo_nodes_cpp package, available in the Vulcanexus Desktop distribution. First, a ROS 2 talker is launched and then a listener node connected to a discovery server. talker will be executed normally and will use *Simple Discovery* while listener will use *Fast DDS Discovery Server Environment Variable* to connect.
with the *Discovery Server*. This will prevent the two from communicating. At this point, the *ROS 2 Router* will be deployed as a bridge between the two discovery mechanisms and will enable the *talker*-listener communication.

## DDS Discovery Mechanisms

The DDS protocol define different *Discovery Mechanisms* in order to DDS Participants discover each other automatically. This mechanism can be configured by the user.

### Simple Discovery

This mechanism is the standard behavior of DDS and it uses multicast capabilities so every Node discovers any other node in the network and all its *Endpoints*. This is the mechanism used by default in ROS 2. Nodes can be isolated by using different *Domain Ids*. Please, check the following *tutorial* to know how to communicate ROS 2 Nodes in different *Domain Ids*.

### Discovery Server

Some networks do not support multicast, or are so big that discovering the whole network implies high amount of time. In this cases, a new *Discovery Mechanism* specific of Fast DDS can be used, called *Discovery Server Discovery Mechanism*. This mechanisms uses a *Server* or network of *Servers* that centralize the discovery traffic and distributes it to only the required entities, while the communication still occurs *Peer to Peer*. In order to know more about *Discovery Server*, please refer to Fast DDS Documentation or check this *tutorial* about how to configure and use *Discovery Server*.

## Prerequisites

It is required to have previously installed Vulcanexus using one of the following installation methods:

- *Linux binary installation*
- *Linux installation from sources*
- *Docker installation*

### Deploy ROS 2 nodes

First let’s run the ROS 2 *Discovery Server* and afterwards the *talker* and *listener* nodes.

### Environment setup

Setup the Vulcanexus environment to have the *demo_nodes_cpp* package available. For this, there are two possible options:

1. Running the Vulcanexus Docker image.

   Run the Vulcanexus Docker image with:

   ```bash
   docker run -it ubuntu-vulcanexus:iron-desktop
   ```

   Then, within the container, source the Vulcanexus installation with:
2. Setting up the development environment on the local host. For this second option, it is necessary to have installed the `vulcanexus-iron-desktop` package, since this is the one that includes all the simulation tools, demos and tutorials.

Source the following file to setup the Vulcanexus environment:

```
source /opt/vulcanexus/iron/setup.bash
```

**Running ROS 2 Discovery Server**

In order to run the *Discovery Server* use the *Fast DDS* CLI. Take care of using the public IP of the machine hosting the *Discovery Server* and an available port.

```
fastdds discovery --server-id 0 --ip-address 127.0.0.1 --port 11666
```

**Note:** The IP used in this tutorial is `localhost` and thus it will only work if all the commands are executed in the same host.

**Running ROS 2 nodes**

Once the environment has been setup using one of the above options, run the ROS 2 *talker* node in one terminal.

```
ros2 run demo_nodes_cpp talker
```

Then, on another terminal, run the ROS 2 *listener* node in ROS 2 using `ROS_DISCOVERY_SERVER` environment variable to connect it with the server. The IP and port must match with the ones set in the server (*check here*).

```
ROS_DISCOVERY_SERVER="127.0.0.1:11666" ros2 run demo_nodes_cpp listener
```

At this point, the *listener* should not receive any data. If not, please go again through the previous steps.

**Deploy ROS 2 Router**

Then, create the *ROS 2 Router* configuration file as the one shown below. The IP and port of the connection of the *ROS 2 Router* participant configured in this configuration file must match with the ones set in the server (*check here*).

**Note:** There is an available configuration file in *DDS Router* repository.

**Note:** If deploying Vulcanexus from the Docker image, note that you will need to have a configuration file (config.yaml) for the *ROS 2 Router* Edge accessible from your Docker container. This can be achieved by mounting a shared volume when launching the container, by copying the file from the local host to the container in case it is already running, or by editing a file from the Docker container itself.
This configuration defines 2 different Router Participants, internal “interfaces” for the ROS 2 Router. One Participant is created in Domain Id 0, while the other will connect as a client to the already running Discovery Server. Topics, Data Types, Quality of Services and order of messages will be respected when redirecting the data.

Running ROS 2 Router

Now, run the ROS 2 Router with the configuration file created as an argument.

ddsrouter --config-path <path/to/file>/simple_ds_bridge.yaml

At this point you should see some information like the one shown below. This indicates that the ROS 2 Router has started correctly and it is currently running.

Starting DDS Router Tool execution.
DDS Router running.

In order to close the execution, just press Ctrl+C or send a signal (SIGINT 2 or SIGTERM 15) to close it.

Alternative deployment: ROS 2 Router as DS Server

The ROS 2 Router can also work as the Discovery Server itself, working as a reliable stable node in the network that distribute discovery traffic and communicate multiple networks. Just adding listening-addresses to a discovery server ROS 2 Router Participant will make it behave as a Discovery Server. In the following figure shows this specific use case and the configuration required for communicating a ROS 2 Domain with a Discovery Server network established by the ROS 2 Router.

version: v3.0
participants:

- **name**: ROS_2_Domain_0
  - **kind**: simple
  - **domain**: 0

- **name**: ROS_2_DS_Server
  - **kind**: discovery-server

  **discovery-server-guid**:
  - **ros-discovery-server**: true
  - **id**: 0

  **listening-addresses**:
  - **domain**: localhost
  - **port**: 11666

### 4.12.3 Debugging ROS 2 Router

- **Background**
- **Prerequisites**
- **Deploy ROS 2 nodes**
  - Environment setup
  - Running ROS 2 nodes
- **Deploy ROS 2 Router**
  - Echo Participant
  - Running ROS 2 Router
- **Advance debugging**
  - Building ROS 2 Router from sources
  - Running ROS 2 Router
Monitor ROS 2 Router internal operation

Background

eProsima ROS 2 Router, a.k.a DDS Router, is an end-user software application that enables the connection of distributed ROS 2 networks (see ROS 2 Router documentation here). That is, ROS 2 nodes such as publishers and subscriptions, or clients and services, deployed in one geographic location and using a dedicated local network will be able to communicate with other ROS 2 nodes deployed in different geographic areas on their own dedicated local networks as if they were all on the same network through the use of ROS 2 Router.

This tutorial explains how to debug the ROS 2 Router to detect errors in the network configuration that may cause the discovery of the ROS 2 nodes and the deployed ROS 2 Routers to not be discovered, preventing the correct behaviour of automatic data routing.

To accomplish this task, this tutorial introduces a simple use case that configures the DDS Router to obtain as much information as possible from its execution. Thus, the user will be able to apply the configuration that best suits his needs when debugging the DDS Router and understanding its behavior.

Moreover, depending on the user's knowledge, it will be possible to configure the amount of information that the ROS 2 Router will report, such as precise and accurate information about the discovery of ROS 2 nodes, the topics automatically found in the network, or the data being relayed.

As already mentioned, the approach of this tutorial is straightforward and is illustrated in the following figure:

This tutorial will use the demo_nodes_cpp package, available in the Vulcanexus Desktop distribution. First, a ROS 2 talker is launched and then a listener node is started in a different ROS 2 Domain. This will prevent the two from communicating. At this point, the ROS 2 Router will be deployed as a bridge between the two Domains and will enable the talker-listener communication. Please take into account that a specific configuration will be applied to the ROS 2 Router in order to see its status and operation at runtime.
**Warning:** It is important to mention that the performance of the *ROS 2 Router* will be affected due to the generation of extra code to present this information so it is not recommended to compile the *ROS 2 Router* following this tutorial in critical or production environments.

### Prerequisites

It is required to have previously installed Vulcanexus using one of the following installation methods:

- *Linux binary installation*
- *Linux installation from sources*
- *Docker installation*

### Deploy ROS 2 nodes

First let’s run the ROS 2 *talker* and *listener* nodes.

### Environment setup

Setup the Vulcanexus environment to have the *demo_nodes_cpp* package available. For this, there are two possible options:

1. Running the Vulcanexus Docker image.
   
   Run the Vulcanexus Docker image with:

   ```bash
   docker run -it ubuntu-vulcanexus:iron-desktop
   ```

   Then, within the container, source the Vulcanexus installation with:

   ```bash
   source /opt/vulcanexus/iron/setup.bash
   ```

2. Setting up the development environment on the local host. For this second option, it is necessary to have installed the *vucanexus-iron-desktop* package, since this is the one that includes all the simulation tools, demos and tutorials.

   Source the following file to setup the Vulcanexus environment:

   ```bash
   source /opt/vulcanexus/iron/setup.bash
   ```

### Running ROS 2 nodes

Once the environment has been setup using one of the above options, run the ROS 2 *talker* node in one terminal.

```bash
ros2 run demo_nodes_cpp talker
```

Then, on another terminal, run the ROS 2 *listener* node in ROS 2 Domain 1.

```bash
ROS_DOMAIN_ID=1 ros2 run demo_nodes_cpp listener
```

At this point, the *listener* should not receive any data. If not, please go again through the previous steps.
Deploy ROS 2 Router

Then, create the *ROS 2 Router* configuration file as the one shown below.

**Note:** If deploying Vulcanexus from the Docker image, note that you will need to have a configuration file (config.yaml) for the DDS Router Edge accessible from your Docker container.

This can be achieved by mounting a shared volume when launching the container, by copying the file from the local host to the container in case it is already running, or by editing a file from the Docker container itself.

```
version: v3.0

participants:

- name: ROS_2_Domain_0
  kind: local
  domain: 0

- name: ROS_2_Domain_1
  kind: local
  domain: 1

- name: Echo_Participant
  kind: echo
  discovery: true
  data: true
  verbose: true
```

Main aspect of this configuration is the Echo Participant. This type of participant, specific for debugging purposes, is explained in detail below.

**Echo Participant**

The *Echo Participant* is a participant/interface of the *ROS 2 Router* that prints in stdout all discovery information and/or user data that is received by the *ROS 2 Router*. Therefore, this participant does not perform any discovery or data reception functionality.

In the case of discovery traces, messages such as the following will be displayed:

```
New endpoint discovered: Endpoint{<endpoint_guid>;<endpoint_kind>;<topic_info>}. 
```

For data reception messages, the traces show the following information:

```
Received data in Participant: <participant_id> in topic: <topic>.
```

These logs contain the Participant Name of the participant that has originally received the message, and the Topic Information where this message has been received. The Topic Information shown the Topic Name, Topic Data Type Name and the QoS. Additionally, extra information such as the data Payload (in hexadecimal format) and source Endpoint Guid is displayed in verbose mode:

```
In Endpoint: <endpoint_guid> from Participant: <participant_id> in topic: <topic>.
_payload received: <payload> with specific qos: <specific_qos>.
```
Running ROS 2 Router

Now, run the *ROS 2 Router* with the configuration file created as an argument.

```
ddsrouter -c <path/to/file>/echo.yaml
```

At this point you should see some information like the one shown below. This indicates that the *ROS 2 Router* has discovered the deployed ROS 2 nodes, their topics, the QoS of each topic and is relaying the information coming from the talker to the listener (from Domain 0 to Domain 1).

Starting DDS Router Tool execution.
DDS Router running.
New endpoint discovered: Endpoint{01.0f.eb.7d.3c.00.ad.17.01.00.00.00|0.0.1.3;writer; _DdsTopic{ros_discovery_info;rmw_dds_common::msg::dds::ParticipantEntitiesInfo_;Fuzzy _-{Partitions{};OwnershipStrength{0}};Active;ParticipantId{SimpleParticipant_Domain_0}}.
New endpoint discovered: Endpoint{01.0f.eb.7d.3c.00.ad.17.01.00.00.00|0.0.3.3;writer; _DdsTopic{rt/rosout;rcl_interfaces::msg::dds::Log_;Fuzzy{Level(20) TopicQoS{TRANSIENT_LOCAL;RELIABLE;SHARED;depth(5000)}}};SpecificEndpointQoS{Partitions{};OwnershipStrength{0}};Active;ParticipantId{SimpleParticipant_Domain_0}}.
New endpoint discovered: Endpoint{01.0f.eb.7d.3c.00.ad.17.01.00.00.00|0.0.5.3;writer; _DdsTopic{rr/talker/get_parametersReply;rcl_interfaces::srv::dds::GetParameters_Response_;Fuzzy{Level(20) TopicQoS{VOLATILE;RELIABLE;SHARED;depth(5000)}}};SpecificEndpointQoS{Partitions{};OwnershipStrength{0}};Active;ParticipantId{SimpleParticipant_Domain_0}}.
New endpoint discovered: Endpoint{01.0f.eb.7d.3c.00.ad.17.01.00.00.00|0.0.7.3;writer; _DdsTopic{rt/chatter;std_msgs::msg::dds::String_;Fuzzy{Level(20) TopicQoS{VOLATILE;RELIABLE;SHARED;depth(5000)}}};SpecificEndpointQoS{Partitions{};OwnershipStrength{0}};Active;ParticipantId{SimpleParticipant_Domain_0}}.
...

All this information may be overwhelming and not easy to read at first. Therefore, *Echo Participant* settings can be changed to match what you want to see, as described in the previous section.

4.12. Vulcanexus Cloud Tutorials 1615
For example, if you just want to show which nodes the *ROS 2 Router* is discovering, it is possible to disable the data and verbose levels of the *Echo Participant* configuration.

**Advance debugging**

The *ROS 2 Router* can be built with the built-in debug traces so that the internal behavior of the software can be fully monitored. To do so, the user/developer should recompile the *ROS 2 Router* package that comes with Vulcanexus by adding the required compilation options that enable the debug operation mode.

This provides some additional options to filter the debug information shown by the *ROS 2 Router*. We will explain this in detail below.

**Note:** Do not mistake the debug mode of *ROS 2 Router* with the Debug compilation of C++ code, since *ROS 2 Router* is still compiled in *Release* mode but with debug traces.

**Building ROS 2 Router from sources**

Building the *ROS 2 Router* from sources in Vulcanexus is straightforward as Vulcanexus ships with all the necessary tools for this task, so there is no prerequisite other than the one already shown in this tutorial.

The steps to follow are described below:

1. Load the Vulcanexus environment.
   - If you are running Vulcanexus from the distributed Docker image, simply run the Vulcanexus Docker container.
     
     ```bash
docker run -it ubuntu-vulcanexus:iron-desktop
     ```
   - If you have installed Vulcanexus on your local machine, load the environment with the following command.
     
     ```bash
source /opt/vulcanexus/iron/setup.bash
     ```

1. Create the development workspace, download the *ROS 2 Router* from GitHub, and build it by executing the following commands:

   ```bash
   mkdir -p ddsrouter_ws/src
cd ddsrouter_ws
git clone https://github.com/eProsima/DDS-Router src/ddsrouter
colcon build --cmake-args -DLOG_INFO=ON
   ```

**Running ROS 2 Router**

Update the environment setup to use the built *ROS 2 Router* instead of the one delivered in Vulcanexus.

```bash
source ddsrouter_ws/install/setup.bash
```
Monitor ROS 2 Router internal operation

ROS 2 Router offers several input arguments to configure the information displayed when it is built with the internal debug traces. These are:

- `--log-verbosity <info|warning|error>`: set the verbosity level so only log messages with equal or higher importance level are shown.
- `--log-filer <regex>`: set a regex string as filter. Only log messages with a category that matches this regex will be printed (ERROR messages will be always shown unless `log-verbosity` argument is set to ERROR).
- `--debug`: set `log-verbosity` to info and `log-filer` to DDSROUTER.

Thus, ROS 2 Router can be run as follows:

```
.ddsrouter -c <path/to/file>/echo.yaml --log-verbosity warning --log-filer DDSROUTER_DISCOVERY_DATABASE
```

The ROS 2 Router should prompt some information like the one show below.

```
2022-11-21 12:52:35.912 [DDSROUTER_DISCOVERY_DATABASE Info] Inserting a new discovered Endpoint Endpoint{01.0f.3a.59.2c.00.41.5f.02.00.00.00|0.0.6.4;reader;DdsTopic{rq/slam_toolbox/get_parameter_typesRequest;rcl_interfaces::srv::GetParameterTypes_Request_;Fuzzy{Level(20) TopicQoS{VOLATILE;RELIABLE;SHARED;depth(5000)}}};SpecificEndpointQoS{Partitions{};OwnershipStrength{0}};Active;ParticipantId{SimpleParticipant_Domain_1}}. -> Function add_endpoint

New endpoint discovered: Endpoint{01.0f.3a.59.2c.00.41.5f.02.00.00.00|0.0.6.4;reader;DdsTopic{rq/slam_toolbox/get_parameter_typesRequest;rcl_interfaces::srv::GetParameterTypes_Request_;Fuzzy{Level(20) TopicQoS{VOLATILE;RELIABLE;SHARED;depth(5000)}}};SpecificEndpointQoS{Partitions{};OwnershipStrength{0}};Active;ParticipantId{SimpleParticipant_Domain_1}}.
```

```
2022-11-21 12:52:35.921 [DDSROUTER_DISCOVERY_DATABASE Info] Inserting a new discovered Endpoint Endpoint{01.0f.3a.59.2c.00.41.5f.02.00.00.00|0.0.8.4;reader;DdsTopic{rq/slam_toolbox/set_parametersRequest;rcl_interfaces::srv::SetParameters_Request_;Fuzzy{Level(20) TopicQoS{VOLATILE;RELIABLE;SHARED;depth(5000)}}};SpecificEndpointQoS{Partitions{};OwnershipStrength{0}};Active;ParticipantId{SimpleParticipant_Domain_1}}. -> Function add_endpoint

New endpoint discovered: Endpoint{01.0f.3a.59.2c.00.41.5f.02.00.00.00|0.0.8.4;reader;DdsTopic{rq/slam_toolbox/set_parametersRequest;rcl_interfaces::srv::SetParameters_Request_;Fuzzy{Level(20) TopicQoS{VOLATILE;RELIABLE;SHARED;depth(5000)}}};SpecificEndpointQoS{Partitions{};OwnershipStrength{0}};Active;ParticipantId{SimpleParticipant_Domain_1}}.
```

```
2022-11-21 12:52:35.933 [DDSROUTER_DISCOVERY_DATABASE Info] Inserting a new discovered Endpoint Endpoint{01.0f.3a.59.2c.00.41.5f.02.00.00.00|0.0.8.4;reader;DdsTopic{rq/slam_toolbox/set_parametersAtomicallyRequest;rcl_interfaces::srv::SetParametersAtomically_Request_;Fuzzy{Level(20) TopicQoS{VOLATILE;RELIABLE;SHARED;depth(5000)}}};SpecificEndpointQoS{Partitions{};OwnershipStrength{0}};Active;ParticipantId{SimpleParticipant_Domain_1}}. -> Function add_endpoint

New endpoint discovered: Endpoint{01.0f.3a.59.2c.00.41.5f.02.00.00.00|0.0.8.4;reader;DdsTopic{rq/slam_toolbox/set_parametersAtomicallyRequest;rcl_interfaces::srv::SetParametersAtomically_Request_;Fuzzy{Level(20) TopicQoS{VOLATILE;RELIABLE;SHARED;depth(5000)}}};SpecificEndpointQoS{Partitions{};OwnershipStrength{0}};Active;ParticipantId{SimpleParticipant_Domain_1}}.
```

(continues on next page)
4.12.4 Vulcanexus Cloud and Kubernetes

- **Background**
- **Prerequisites**
- **Local setup**
  - Local router
  - Talker
- **Kubernetes setup**
  - DDS-Router deployment
  - Listener deployment

**Background**

This walk-through tutorial sets up both a Kubernetes (K8s) network and a local environment in order to establish communication between a pair of ROS nodes, one sending messages from a LAN (talker) and another one receiving them in the Cloud (listener). Cloud environments such as container-oriented platforms can be connected using eProsima DDS Router, and thus, by launching a DDS Router instance at each side, communication can be established.
Prerequisites

Ensure that the Vulcanexus installation includes the cloud and the ROS 2 demo nodes package (it is suggested to use vulcanexus-iron-desktop). Also, remember to source the environment in every terminal in this tutorial.

```
source /opt/vulcanexus/iron/setup.bash
```

**Warning:** For the full understanding of this tutorial basic understanding of Kubernetes is required.

Local setup

The local instance of **DDS Router** (local router) only requires to have a Simple Participant and a WAN Participant that will play the client role in the discovery process of remote participants (see Initial Peers discovery mechanism).

After having acknowledged each other’s existence through Simple DDS discovery mechanism (multicast communication), the local participant will start receiving messages published by the ROS 2 talker node, and will then forward them to the WAN participant. Next, these messages will be sent to another participant hosted on a K8s cluster to which it connects via WAN communication over UDP/IP. Following there is a representation of the above-described scenario:

![Diagram of DDS Router and Participants]

Local router

The configuration file used by the local router will be the following:

```
# local-ddsrouter.yaml

version: v3.0

allowlist:
  - name: "rt/chatter"
    type: "std_msgs::msg::dds::String_

participants:
```

(continues on next page)
- **name**: SimpleParticipant
  
  **kind**: local
  
  **domain**: 0

- **name**: LocalWAN
  
  **kind**: wan
  
  **listening-addresses**: # Needed for UDP communication
  
  - **ip**: 3.3.3.3
    
    # LAN public IP
  
  - **port**: 30003
    
    **transport**: udp
  
  **connection-addresses**: # Public IP exposed by the k8s cluster to reach the cloud
  
  - **ip**: 2.2.2.2
    
    # Public IP exposed by the k8s cluster to reach the cloud
  
    **DDS-Router**
  
    - **port**: 30002
      
      **transport**: udp

Please, copy the previous configuration snippet and save it to a file in your current working directory with name local-ddsrouter.yaml.

Note that the simple participant will be receiving messages sent in DDS domain 0. Also note that, due to the choice of UDP as transport protocol, a listening address with the LAN public IP address needs to be specified for the local WAN participant, even when behaving as client in the participant discovery process. Make sure that the given port is reachable from outside this local network by properly configuring port forwarding in your Internet router device. The connection address points to the remote WAN participant deployed in the K8s cluster. For further details on how to configure WAN communication, please have a look at WAN Configuration.

**Note**: As an alternative, TCP transport may be used instead of UDP. This has the advantage of not requiring to set a listening address in the local router’s WAN participant (TCP client), so there is no need to fiddle with the configuration of your Internet router device.

To launch the local router, execute the following command (remember to source the Vulcanexus environment):

```
ddsrouter --config-path local-ddsrouter.yaml
```

**Talker**

In another terminal, run the following command in order to start the ROS 2 node that publishes messages in DDS domain 0 (remember to source the Vulcanexus environment):

```
ros2 run demo_nodes_cpp talker
```
**Kubernetes setup**

Two different deployments are required to receive the talker messages in the Cloud, each in a different K8s pod; the first one being a DDS Router cloud instance (cloud router), which consists of two participants:

- A **WAN Participant** that receives the messages coming from our LAN through the aforementioned UDP communication channel.
- A **Local Discovery Server** (local DS) that propagates them to a ROS 2 listener node hosted in a different K8s pod.

**Note:** The choice of a Local Discovery Server instead of a Simple Participant to communicate with the listener has to do with the difficulty of enabling multicast routing in cloud environments.

The other deployment is the ROS 2 listener node. This node has to be launched as a Client to the local DS running on the first deployment.

The described scheme is represented in the following figure:

```
kind: Service
apiVersion: v1
metadata:
  name: ddsrouter
  labels:
    app: ddsrouter
spec:
  ports:
    - name: UDP-30002
      protocol: UDP
      port: 30002
      targetPort: 30002
  selector:

(continues on next page)
```
### app: ddsrouter
### type: LoadBalancer

<table>
<thead>
<tr>
<th>kind: Service</th>
<th>apiVersion: v1</th>
</tr>
</thead>
<tbody>
<tr>
<td>metadata:</td>
<td></td>
</tr>
<tr>
<td>name: local-ddsrouter</td>
<td></td>
</tr>
<tr>
<td>spec:</td>
<td></td>
</tr>
<tr>
<td>ports:</td>
<td></td>
</tr>
<tr>
<td>- name: UDP-30001</td>
<td></td>
</tr>
<tr>
<td>protocol: UDP</td>
<td></td>
</tr>
<tr>
<td>port: 30001</td>
<td></td>
</tr>
<tr>
<td>targetPort: 30001</td>
<td></td>
</tr>
<tr>
<td>selector:</td>
<td></td>
</tr>
<tr>
<td>app: ddsrouter</td>
<td></td>
</tr>
<tr>
<td>clusterIP: 192.168.1.11 # Private IP only reachable within the k8s cluster to communicate with the ddsrouter application</td>
<td></td>
</tr>
<tr>
<td>type: ClusterIP</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** An Ingress needs to be configured for the LoadBalancer service to make it externally-reachable. In this example we consider the assigned public IP address to be 2.2.2.2.

The configuration file used for the cloud router will be provided by setting up a ConfigMap:

<table>
<thead>
<tr>
<th>kind: ConfigMap</th>
<th>apiVersion: v1</th>
</tr>
</thead>
<tbody>
<tr>
<td>metadata:</td>
<td></td>
</tr>
<tr>
<td>name: ddsrouter-config</td>
<td></td>
</tr>
<tr>
<td>data:</td>
<td></td>
</tr>
</tbody>
</table>
| ddsrouter.config.file: | -
| version: v3.0 |
| allowlist:     |               |
| - name: rt/chatter |
| type: std_msgs::msg::dds::String |
| participants:  |               |
| - name: LocalDiscoveryServer |
| kind: local-discovery-server |
| discovery-server-guid: |
| ros-discovery-server: true |
| id: 1 |
| listening-addresses: |
| - ip: 192.168.1.11 # Private IP only reachable within the k8s cluster to communicate with the ddsrouter application |
| port: 30001 |
| transport: udp |
| - name: CloudWAN |
kind: wan  
listening-addresses:  
  - ip: 2.2.2.2 # Public IP exposed by the k8s cluster to reach the  
    cloud DDS-Router  
    port: 30002  
    transport: udp

Following there is a representation of the overall K8s cluster configuration:

**DDS-Router deployment**

The cloud router is launched from within a *Vulcanexus Cloud* Docker image (that can be downloaded in *Vulcanexus webpage*), which uses as configuration file the one hosted in the previously set up ConfigMap. Assuming the name of the generated Docker image is `ubuntu-vulcanexus-cloud:iron`, the cloud router will then be deployed with the following settings:

```yaml
kind: Deployment  
apiVersion: apps/v1  
metadata:  
  name: ddsrouter  
labels:  
  app: ddsrouter  
spec:  
  replicas: 1  
selector:  
  matchLabels:  
    app: ddsrouter  
template:  
  metadata:  
    labels:  
      app: ddsrouter  
spec:  
  volumes:  
    - name: config  
      configMap:  
        name: ddsrouter-config  
        items:  
          - key: ddsrouter.config.file  
            path: DDSROUTER_CONFIGISTRATION.yaml  
containers:
```

(continues on next page)
- **name**: ubuntu-vulcanexus-cloud
  - **image**: ubuntu-vulcanexus-cloud:iron
  - **ports**:
    - **containerPort**: 30001
      - **protocol**: UDP
    - **containerPort**: 30002
      - **protocol**: UDP
  - **volumeMounts**:
    - **name**: config
      - **mountPath**: /tmp
    - **args**: ""-r", "ddsrouter -r 10 -c /tmp/DDSROUTER_CONFIGURATION.yaml"
  - **restartPolicy**: Always

**Listener deployment**

Since ROS 2 demo nodes package is not installed by default in *Vulcanexus Cloud*, a new Docker image adding in this functionality must be generated. Also, the IP address and port of the local Discovery Server must be specified, so a custom entrypoint is also provided.

Copy the following snippet and save it to the current directory as **Dockerfile**:

```bash
FROM ubuntu-vulcanexus-cloud:iron

# Install demo-nodes-cpp
RUN source /opt/vulcanexus/iron/setup.bash && \
    apt update && \
    apt install -y ros-iron-demo-nodes-cpp

COPY ./run.bash /
RUN chmod +x /run.bash

# Setup entrypoint
ENTRYPOINT ["/run.bash"]
```

Copy the following snippet and save it to the current directory as **run.bash**:

```bash
#!/bin/bash

if [[ $1 == "listener" ]]
then
    NODE="listener"
else
    NODE="talker"
fi

SERVER_IP=$2
SERVER_PORT=$3

# Setup environment
source "/opt/vulcanexus/iron/setup.bash"
```

(continues on next page)
Build the docker image running the following command:

```
docker build -t vulcanexus-cloud-demo-nodes:iron -f Dockerfile
```

Now, the listener pod can be deployed by providing the following configuration:

```yaml
kind: Deployment
apiVersion: apps/v1
metadata:
  name: ros2-iron-listener
  labels:
    app: ros2-iron-listener
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ros2-iron-listener
  template:
    metadata:
      labels:
        app: ros2-iron-listener
    spec:
      containers:
      - name: vulcanexus-cloud-demo-nodes
        image: vulcanexus-cloud-demo-nodes:iron
        args:
          - listener
          - 192.168.1.11
          - '30001'
        restartPolicy: Always
```

Once all these components are up and running, communication should have been established between the talker and listener nodes, so that messages finally manage to reach the listener pod and get printed in its STDOUT.

Feel free to interchange the locations of the ROS nodes by slightly modifying the provided configuration files, hosting the talker in the K8s cluster while the listener runs in the LAN.

### 4.12.5 Edge-Cloud communication on WAN

- **Background**
- **Prerequisites**
- **Deployment on LAN 1**
  - Running turtlesim_node on the edge
  - Running DDS Router Edge
- **Deployment on Cloud**


**Background**

This tutorial will cover the first steps to setup a distributed network of remotely controlled robots from the Cloud or an edge device. More specifically, it will focus on a basic edge-cloud architecture in which there is an edge robot deployed on a LAN with access to the Internet, and a server in the Cloud reachable through the Internet.

**Warning:** This tutorial is intended for WAN communication. However, if communication through a LAN is your only option, it is still possible to follow the tutorial by changing the ROS 2 Domain Ids so that each ROS 2 node uses a different Domain (0 and 1). This way the ROS 2 nodes are logically isolated and will not discover other nodes out of their ROS 2 Domain.

Following, all the elements involved in this architecture will be studied, starting with the edge robot, continuing with the controller hosted in the cloud also built as a ROS 2 node and concluding with the intermediate elements that enable communication over the Internet.

The image below describes the scenario presented in this tutorial.

Several key elements can be observed in it:

1. **ROS 2 Application.** *Turtlesim* is the application used for this tutorial. *Turtlesim* is a ROS 2 application, first developed for ROS, aimed at teaching the basic concepts of ROS 2 such as publish/subscribe, services and actions. The edge robot will then be a *turtlesim_node*, which is a simulator of a robot making use of these communication methods.
2. **ROS 2 Device Controller.** This is a ROS 2 application that sends commands to the edge robot. A basic C++ application has been developed for this tutorial that sends publications under the topic on which the turtlesim_node listens. By means of these publications (commands) from the controller, and the feedback information that the controller receives from the turtlesim_node, it is possible to control this node automatically without the need for user intervention which facilitates the deployment of the scenario at hand. The key feature of the DDS Router is that it is easy to configure, allowing to connect different networks with ROS 2 applications without requiring to apply any changes to the developer’s software or applications.

3. **ROS 2 Router / DDS Router.** eProsima ROS 2 Router, a.k.a DDS Router, is an end-user software application that enables the connection of distributed ROS 2 networks (see DDS Router documentation here). That is, ROS 2 nodes such as publishers and subscribers, or clients and services, deployed in one geographic location and using a dedicated local network will be able to communicate with other ROS 2 nodes deployed in different geographic areas on their own dedicated local networks as if they were all on the same network through the use of DDS Router.

This example presents two routers that enable Internet communication:

- **DDS Router Edge.** This is the DDS Router that is deployed on the edge robot side. This way it is possible for the robot to communicate out-of-the-box with an external server.

- **DDS Router Cloud.** It plays the server role in the communication. It will expose a public network address to which the nodes connect to establish communication.

### Prerequisites

This tutorial will require two machines (Robot 1 and Cloud Server) deployed on different networks (LAN 1 and Cloud). It is possible to simulate the scenario by deploying everything needed on the same machine and two virtual networks but let's focus on the case of a real deployment.

It is also necessary to have previously installed Vulcanexus using one of the following installation methods:

- Linux binary installation
- Linux installation from sources
- Docker installation

### Deployment on LAN 1

First, let's deploy the turtlesim_node and DDS Router Edge on a machine on LAN 1.

### Running turtlesim_node on the edge

Setup the Vulcanexus environment to have the turtlesim_node available. For this, there are two possible options:

1. Running the Vulcanexus Docker image.

   Run the Vulcanexus Docker image with:

   ```
xhost local:root
docker run \
   -it \
   --privileged \ 
   -e DISPLAY=$DISPLAY \ 
   -v /tmp/.X11-unix:/tmp/.X11-unix \
   ubuntu-vulcanexus:iron-desktop
   ```
Then, within the container, source the Vulcanexus installation with:

```
source /opt/vulcanexus/iron/setup.bash
```

2. Setting up the development environment on the local host. For this second option, it is necessary to have installed the `vulcanexus-iron-desktop` package, since this is the one that includes all the simulation tools, demos and tutorials.

Source the following file to setup the Vulcanexus environment:

```
source /opt/vulcanexus/iron/setup.bash
```

Once the environment has been setup using one of the above options, simply run the `turtlesim_node`.

```
ros2 run turtlesim turtlesim_node
```

And a popup window like the following should appear:

![TurtleSim (on 95b590ea7f11)](image)

As can be seen, it is not necessary to perform any additional configuration in the ROS 2 application.
Running DDS Router Edge

Then, to run the DDS Router Edge configure the environment as in the previous step.

Note: If deploying Vulcanexus from the Docker image, note that you will need to have a configuration file (config.yaml) for the DDS Router Edge accessible from your Docker container.

This can be achieved by mounting a shared volume when launching the container, by copying the file from the local host to the container in case it is already running, or by editing a file from the Docker container itself.

Setup the Vulcanexus environment, either in a Docker container or on the local host, running the following command:

```
source /opt/vulcanexus/iron/setup.bash
```

Let’s create a DDS Router configuration file as the one shown below.

```
version: v3.0

participants:
  - name: ROS_2_LAN_1
    kind: local
  - name: Router_Client
    kind: wan
    connection-addresses:
      - ip: 123.123.123.123
        port: 45678
        transport: tcp
```

Next, the most relevant aspects of this configuration file are explained.

The participants are the interfaces of the DDS Router to communicate with other networks. In this case, we have two kinds of participants:

- **local**: this is a simple participant that communicates with all ROS 2 nodes it finds. For more information about this participant please refer to the Simple Participant section of the DDS Router documentation.

- **wan**: it is a participant designed for the communication between two DDS Routers. It uses the Initial Peers discovery mechanism to establish a point-to-point communication between two DDS entities, two DDS Routers in this case.

For the DDS Router Edge, a connection address shall be defined which must be the same as the one exposed by the Cloud Server.

Note: In this case, the DDS Router will forward all topics found in the network. However, it is important to mention that the ROS 2 topics relayed by the DDS Router can be filtered by configuring the allowlist and blocklist. If this is the case please refer to the DDS Router documentation for information on how to do this.

The following figure summarizes the deployment on the edge.

Now, run the DDS Router with the configuration file created as an argument.
ddsrouter -c <path/to/file>/ddsrouter_edge.yaml

Deployment on Cloud

Running the turtlesim_square_move on the Cloud

Run the turtlesim_square_move in the Cloud Server machine, which is the controller of the edge turtlesim_node. This will send commands to the ROS 2 application to the edge to move the turtle and receive information about the current state of the turtle at any time.

A ROS 2 application that moves the turtle by drawing a square has been developed for this purpose. The application is based on the ROS 2 tutorials, but has been slightly modified in order to make it easier to understand and adapt to the ROS 2 and modern C++ programming methods.

Then, start by creating the workspace of this application and downloading the source code:

```bash
mkdir -p turtlesim_move_ws/src && cd turtlesim_move_ws/src
git clone --branch iron https://github.com/eProsima/vulcanexus.git
mv vulcanexus/code/turtlesim .
rm -rf vulcanexus
cd ..
```

Once created the workspace, source the Vulcanexus environment and build the turtlesim_square_move application.

```bash
source /opt/vulcanexus/iron/setup.bash
colcon build
```

**Note:** Since the purpose of this tutorial is not to explain how to create a ROS 2 node, but rather, the communication of these in distributed environments, the code will not be discussed in detail. Stay tuned for new tutorials in which we will discuss how to configure ROS 2 nodes, publishers/subscribers, services and actions.

Then source the turtlesim_move_ws workspace:
source turtlesim_move_ws/install/setup.bash

And finally, run the application:

WAN

ros2 run turtlesim turtlesim_square_move

LAN

ROS_DOMAIN_ID=1 ros2 run turtlesim turtlesim_square_move

**Note:** As stated [here](#), change the ROS 2 Domain Id if running the edge and cloud applications on the same LAN.

The important points to note in this application are the following:

- The control application sends the movement commands to the `turtlesim_node` through a publisher in the `geometry_msgs/msg/Twist` topic. This topic expresses the velocity at which the turtle has to move divided into linear velocity (**linear**) and angular velocity (**angular**).

- The application knows the position of the turtle on the map at any moment and whether it is stopped or moving by subscribing to the `turtlesim/msg/pose` topic. This topic provides information about the turtle’s coordinates (**x** and **y**) and the turtle’s rotation (**theta**). We can also know its linear and angular velocity (**linear_velocity** and **angular_velocity**).

**Running the DDS Router Cloud**

**Configure transversal NAT on the network router**

The first thing to do before starting to configure DDS Router is to configure the network router to allow a remote communication from the Internet to reach a specific device on the LAN, more specifically to expose an IP address and a port to the network that will be used by our DDS Router application.

This configuration will depend on your network router, but it should be similar to the one shown in the following image.

**Warning:** Due to a current limitation of DDS Router, the external port and internal port must match. Stay tuned for new versions of DDS Router that are intended to address this limitation.

**Configure the DDS Router Cloud**

The DDS Router Cloud configuration file is quite similar to the DDS Router Edge configuration file, as can be seen below:

WAN

```yaml
version: v3.0
participants:
  - name: ROS_2_Cloud
    kind: local
```

(continues on next page)
network

These rules are needed to authorise a remote communication from Internet to reach a specific device of your LAN. You can also define the ports(s) that this communication will use.

warning: make sure you have not filtered these ports in the firewall.

LAN

version: v3.0

participants:

- name: Router_Server
  kind: wan
  listening-addresses:
  - ip: 123.123.123.123
    port: 45678
    transport: tcp

- name: ROS_2_Cloud
  kind: local
  domain: 1

- name: Router_Server
  kind: wan
  listening-addresses:
  - ip: 123.123.123.123
    port: 45678
    transport: tcp
In this case there are also two participants, two communication interfaces for the DDS Router. The first one communicates the DDS Router with any ROS 2 node, while the second one enables to establish a communication channel with another DDS Router.

Although quite similar to the WAN participant in the DDS Router Edge instance, notice that this participant sets a listening address (listening-addresses), rather than a connection address. This is because it is the participant that waits for incoming communications since it has this network address exposed and accessible from the Internet.

To finish, as done in the previous steps, setup the Vulcanexus environment sourcing the `setup.bash` file and run the DDS Router Cloud with the above configuration.

```
source /opt/vulcanexus/iron/setup.bash
ddsrouter -c <path/to/file>/ddsrouter_cloud.yaml
```

The following figure summarizes the deployment on the Cloud.

![DDS Router Cloud Diagram]

### Results

If all the steps in this tutorial have been followed, the turtle in the `turtlesim_node` on the edge should move around creating a square,

and the `turtlesim_square_move` should prompt the following traces.

```
root@dbf79a437eb3:/turtlesim_move_ws# ros2 run turtlesim turtlesim_square_move
[INFO] [1657870899.585667136] [turtlesim_square_move]: New goal [7.544445 5.544445, 0.000000]
[INFO] [1657870901.585656311] [turtlesim_square_move]: Reached goal
[INFO] [1657870901.585767260] [turtlesim_square_move]: New goal [7.448444 5.544445, 1.570796]
[INFO] [1657870905.685637930] [turtlesim_square_move]: Reached goal
[INFO] [1657870905.685753714] [turtlesim_square_move]: New goal [7.466837 7.544360, 1.561600]
[INFO] [1657870907.885655744] [turtlesim_square_move]: Reached goal
[INFO] [1657870907.885742857] [turtlesim_square_move]: New goal [7.466837 7.544360, 3.132396]
```

(continues on next page)
Next steps

Feel free to read the following tutorials extending this one to similar scenarios:

- *Edge-Cloud TLS communication on WAN*: secure Edge-Cloud communication channel by using TLS protocol.

### 4.12.6 Edge-Cloud TLS communication on WAN

**Warning:** In this example it is assumed the reader has basic knowledge of *TLS* concepts since terms like *CA*, *Public-key cryptography*, *RSA*, and *Diffie-Hellman* are not explained in detail.

- *Background*
- *Prerequisites*
- *TLS configuration*
- Certification Authority (CA)
- DDS Router Cloud Certificate
- Diffie-Hellman Parameters

• Deployment on LAN 1
  - Running turtlesim_node on the edge
  - Running DDS Router Edge

• Deployment on Cloud
  - Running the turtlesim_square_move on the Cloud
  - Running the DDS Router Cloud

• Results

Background

This tutorial builds on the previous one (Edge-Cloud communication on WAN), further showing how to secure the edge-cloud TCP communication channel with TLS protocol. It is recommended to follow these tutorials in order, as some concepts or installations may be already covered.

Warning: This tutorial is intended for WAN communication. However, if communication through a LAN is your only option, it is still possible to follow the tutorial by changing the ROS 2 Domain Ids so that each ROS 2 node uses a different Domain (0 and 1). This way the ROS 2 nodes are logically isolated and will not discover other nodes out of their ROS 2 Domain.

Following, all the elements involved in this architecture will be studied, starting with the edge robot, continuing with the controller hosted in the cloud also built as a ROS 2 node and concluding with the intermediate elements that enable communication over the Internet.

The image below describes the scenario presented in this tutorial.

Several key elements can be observed in it:

1. **ROS 2 Application.** Turtlesim is the application used for this tutorial. Turtlesim is a ROS 2 application, first developed for ROS, aimed at teaching the basic concepts of ROS 2 such as publish/subscribe, services and actions. The edge robot will then be a turtlesim_node, which is a simulator of a robot making use of these communication methods.

2. **ROS 2 Device Controller.** This is a ROS 2 application that sends commands to the edge robot. A basic C++ application has been developed for this tutorial that sends publications under the topic on which the turtlesim_node listens. By means of these publications (commands) from the controller, and the feedback information that the controller receives from the turtlesim_node, it is possible to control this node automatically without the need for user intervention which facilitates the deployment of the scenario at hand. The key feature of the DDS Router is that it is easy to configure, allowing to connect different networks with ROS 2 applications without requiring to apply any changes to the developer’s software or applications.

3. **ROS 2 Router / DDS Router.** eProsima ROS 2 Router, a.k.a DDS Router, is an end-user software application that enables the connection of distributed ROS 2 networks (see DDS Router documentation here). That is, ROS 2 nodes such as publishers and subscribers, or clients and services, deployed in one geographic location and using a dedicated local network will be able to communicate with other ROS 2 nodes deployed in different geographic areas on their own dedicated local networks as if they were all on the same network through the use of DDS Router.
This internet connection will be established using TCP protocol secured with TLS. This means that the data connection between both servers will be guaranteed to be secured, and the data encrypted.

This example presents two routers that enable Internet communication:

- **DDS Router Cloud.** It plays the server role in the communication. It will expose a public network address to which the nodes connect to establish communication. This DDS Router will work as the TLS Server, which means it must be authenticated in order for the client to trust and connect it.

- **DDS Router Edge.** This is the DDS Router that is deployed on the edge robot side. This way it is possible for the robot to communicate out-of-the-box with an external server. This DDS Router will work as the TLS Client, so it must accept the Server CA in order to authenticate it before the communication is established.

In the following steps it will be explained how to configure TLS in both sides, so the communication is secured.

**Prerequisites**

This tutorial will require two machines (*Robot 1* and *Cloud Server*) deployed on different networks (*LAN 1* and *Cloud*). It is possible to simulate the scenario by deploying everything needed on the same machine and two virtual networks but let’s focus on the case of a real deployment.

It is also necessary to have previously installed *Vulcanexus* using one of the following installation methods:

- *Linux binary installation*
- *Linux installation from sources*
- *Docker installation*

**Note:** *OpenSSL*, required in this tutorial to generate security keys and certificates, is already part of *Vulcanexus* toolset.
TLS configuration

Let us first generate the TLS credentials with which DDS Router instances will be configured. In this example, Elliptic Curve (EC) keys will be generated, and with/without password versions of the commands will be provided.

These files should be generated in cloud machine, or in a neutral host. The Cloud machine requires to hold the private key, as some other TLS files, while the Edge machine only requires to know the CA so it can assure it is communicating with a trusted server.

Certification Authority (CA)

Create first a mock CA that will be used to generate a certificate for the DDS Router Cloud:

```
# Generate the Certificate Authority (CA) Private Key > ca.key
openssl ecparam -name prime256v1 -genkey -noout -out ca.key
# openssl ecparam -name prime256v1 -genkey | openssl ec -aes256 -out ca.key -passout...
    →pass:cakey # with password

# Generate the Certificate Authority Certificate > ca.crt
openssl req -new -x509 -sha256 -key ca.key -out ca.crt -days 365 -config ca.cnf
# openssl req -new -x509 -sha256 -key ca.key -out ca.crt -days 365 -config ca.cnf -
    →passin pass:cakey # with password
```

DDS Router Cloud Certificate

We can now generate a certificate for the DDS Router Cloud instance following the steps below:

```
# Generate the DDS-Router Certificate Private Key > ddsrouter.key
openssl ecparam -name prime256v1 -genkey -noout -out ddsrouter.key
# openssl ecparam -name prime256v1 -genkey | openssl ec -aes256 -out ddsrouter.key -
    →passout pass:ddsrouterpass # with password

# Generate the DDS-Router Certificate Signing Request > ddsrouter.csr
openssl req -new -sha256 -key ddsrouter.key -out ddsrouter.csr -config ddsrouter.cnf
# openssl req -new -sha256 -key ddsrouter.key -out ddsrouter.csr -config ddsrouter.cnf -
    →passin pass:ddsrouterpass # with password

# Generate the DDS-Router Certificate (computed on the CA side) > ddsrouter.crt
openssl x509 -req -in ddsrouter.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out...
    →ddsroutert.crt -days 1000 -sha256
# openssl x509 -req -in ddsrouter.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out...
    →ddsroutert.crt -days 1000 -sha256 -passin pass:cakey # with password
```
**Diffie-Hellman Parameters**

It only remains to generate the Diffie-Hellman (DF) parameters to define how OpenSSL performs the DF key-exchange:

```
# Generate Diffie-Hellman (DF) parameters > dh_params.pem
openssl dhparam -out dh_params.pem 2048
```

**Deployment on LAN 1**

First, let's deploy the `turtlesim_node` and DDS Router Edge on a machine on *LAN 1*.

**Running turtlesim_node on the edge**

Setup the Vulcanexus environment to have the `turtlesim_node` available. For this, there are two possible options:

1. Running the Vulcanexus Docker image.
   - Run the Vulcanexus Docker image with:
     ```
xhost local:root
docker run \
    -it \
    --privileged \
    -e DISPLAY=$DISPLAY \
    -v /tmp/.X11-unix:/tmp/.X11-unix \
    ubuntu-vulcanexus:iron-desktop
     ```
   - Then, within the container, source the Vulcanexus installation with:
     ```
     source /opt/vulcanexus/iron/setup.bash
     ```

2. Setting up the development environment on the local host. For this second option, it is necessary to have installed the `vucanexus-iron-desktop` package, since this is the one that includes all the simulation tools, demos and tutorials.
   - Source the following file to setup the Vulcanexus environment:
     ```
     source /opt/vulcanexus/iron/setup.bash
     ```

Once the environment has been setup using one of the above options, simply run the `turtlesim_node`.

```
ros2 run turtlesim turtlesim_node
```

And a popup window like the following should appear:

As can be seen, it is not necessary to perform any additional configuration in the ROS 2 application.
Running DDS Router Edge

Then, to run the DDS Router Edge configure the environment as in the previous step.

**Note:** If deploying Vulcanexus from the Docker image, note that you will need to have a configuration file (`config.yaml`) for the DDS Router Edge accessible from your Docker container.

This can be achieved by mounting a shared volume when launching the container, by copying the file from the local host to the container in case it is already running, or by editing a file from the Docker container itself.

Setup the Vulcanexus environment, either in a Docker container or on the local host, running the following command:

```
source /opt/vulcanexus/iron/setup.bash
```

Let’s create a DDS Router configuration file as the one shown below.

```
version: v3.0

participants:
  - name: ROS_2_LAN_1
    kind: local
  - name: Router_Client
    kind: wan
    connection-addresses:
```
- ip: 123.123.123.123
  port: 45678
  transport: tcp

tls:
  ca: ca.crt

Next, the most relevant aspects of this configuration file are explained.

The participants are the interfaces of the DDS Router to communicate with other networks. In this case, we have two kinds of participants:

- **local**: this is a simple participant that communicates with all ROS 2 nodes it finds. For more information about this participant please refer to the Simple Participant section of the DDS Router documentation.

- **wan**: it is a participant designed for the communication between two DDS Routers. It uses the Initial Peers discovery mechanism to establish a point-to-point communication between two DDS entities, two DDS Routers in this case.

For the DDS Router Edge, a connection address shall be defined which must be the same as the one exposed by the Cloud Server. TLS configuration parameters required to establish a secure connection over TCP are provided under the `tls` tag. For clients, only the certificate authority (CA) needs to be provided. Please refer to DDS Router documentation for more details on how to configure TLS in a WAN participant.

**Note:** In this case, the DDS Router will forward all topics found in the network. However, it is important to mention that the ROS 2 topics relayed by the DDS Router can be filtered by configuring the allowlist and blocklist. If this is the case please refer to the DDS Router documentation for information on how to do this.

The following figure summarizes the deployment on the edge.

Now, run the DDS Router with the configuration file created as an argument.
Deployment on Cloud

Running the turtlesim_square_move on the Cloud

Run the turtlesim_square_move in the Cloud Server machine, which is the controller of the edge turtlesim_node. This will send commands to the ROS 2 application to the edge to move the turtle and receive information about the current state of the turtle at any time.

A ROS 2 application that moves the turtle by drawing a square has been developed for this purpose. The application is based on the ROS 2 tutorials, but has been slightly modified in order to make it easier to understand and adapt to the ROS 2 and modern C++ programming methods.

Then, start by creating the workspace of this application and downloading the source code:

```
mkdir -p turtlesim_move_ws/src && cd turtlesim_move_ws/src
git clone --branch iron https://github.com/eProsima/vulcanexus.git
mv vulcanexus/code/turtlesim .
rm -rf vulcanexus
cd ..
```

Once created the workspace, source the Vulcanexus environment and build the turtlesim_square_move application:

```
source /opt/vulcanexus/iron/setup.bash
colcon build
```

**Note:** Since the purpose of this tutorial is not to explain how to create a ROS 2 node, but rather, the communication of these in distributed environments, the code will not be discussed in detail. Stay tuned for new tutorials in which we will discuss how to configure ROS 2 nodes, publishers/subscribers, services and actions.

Then source the turtlesim_move_ws workspace:

```
source turtlesim_move_ws/install/setup.bash
```

And finally, run the application:

**WAN**

```
ros2 run turtlesim turtlesim_square_move
```

**LAN**

```
ROS_DOMAIN_ID=1 ros2 run turtlesim turtlesim_square_move
```

**Note:** As stated here, change the ROS 2 Domain Id if running the edge and cloud applications on the same LAN.

The important points to note in this application are the following:

- The control application sends the movement commands to the turtlesim_node through a publisher in the geometry_msgs/msg/Twist topic. This topic expresses the velocity at which the turtle has to move divided into linear velocity (linear) and angular velocity (angular).
The application knows the position of the turtle on the map at any moment and whether it is stopped or moving by subscribing to the turtlesim/msg/pose topic. This topic provides information about the turtle's coordinates (x and y) and the turtle's rotation (theta). We can also know its linear and angular velocity (linear_velocity and angular_velocity).

Running the DDS Router Cloud

In case this device is working under a NAT, check previous tutorial for more information about how to configure the NAT to be accessible from the outside.

Configure the DDS Router Cloud

The DDS Router Cloud configuration file is quite similar to the DDS Router Edge configuration file, as can be seen below:

WAN

```
version: v3.0

participants:
  - name: ROS_2_Cloud
    kind: local
  - name: Router_Server
    kind: wan
    listening-addresses:
      - ip: 123.123.123.123
        port: 45678
        transport: tcp
    tls:
      ca: ca.crt
      password: ddsrouterpass
      private_key: ddsrouter.key
      cert: ddsrouter.crt
      dh_params: dh_params.pem
```

LAN

```
version: v3.0

participants:
  - name: ROS_2_Cloud
    kind: local
    domain: 1
  - name: Router_Server
    kind: wan
    listening-addresses:
      - ip: 123.123.123.123
```

(continues on next page)
port: 45678
transport: tcp

tls:
  ca: ca.crt
  password: ddsrouterpass
  private_key: ddsrouter.key
  cert: ddsrouter.crt
  dh_params: dh_params.pem

Note: As stated here, set the ROS 2 Domain Id on the local participant in order to discover the turtlesim_square_move ROS 2 node.

In this case there are also two participants, two communication interfaces for the DDS Router. The first one communicates the DDS Router with any ROS 2 node, while the second one enables to establish a communication channel with another DDS Router.

Even so there are some differences in the second participant that are worth mentioning:

1. This participant sets a listening address (listening-addresses), rather than a connection address. This is because it is the participant that waits for incoming communications since it has this network address exposed and accessible from the Internet.

2. The previously generated TLS parameters and certificates are attached under the tls tag. Note that server and client configurations differ, as explained in DDS Router documentation.

To finish, as done in the previous steps, setup the Vulcanexus environment sourcing the setup.bash file and run the DDS Router Cloud with the above configuration.

source /opt/vulcanexus/iron/setup.bash
ddsrouter -c <path/to/file>/ddsrouter_cloud.yaml

The following figure summarizes the deployment on the Cloud.

---

4.12. Vulcanexus Cloud Tutorials 1643
**Results**

If all the steps in this tutorial have been followed, the turtle in the `turtlesim_node` on the edge should move around creating a square,

![TurtleSim Image](image)

and the `turtlesim_square_move` should prompt the following traces:

```
root@dbf79a437eb3:/turtlesim_move_ws# ros2 run turtlesim turtlesim_square_move
[INFO] [1657870899.585667136] [turtlesim_square_move]: New goal [7.544445 5.544445, 0.000000]
[INFO] [1657870901.585656311] [turtlesim_square_move]: Reached goal
[INFO] [1657870901.585767260] [turtlesim_square_move]: New goal [7.448444 5.544445, 1.570796]
[INFO] [1657870905.685637930] [turtlesim_square_move]: Reached goal
[INFO] [1657870905.685753714] [turtlesim_square_move]: New goal [7.466837 7.544360, 1.561600]
[INFO] [1657870911.985655175] [turtlesim_square_move]: Reached goal
[INFO] [1657870914.085652821] [turtlesim_square_move]: Reached goal
```

1644 Chapter 4. Contributing to the documentation
4.12.7 Edge-Edge communication via Repeater

**Background**

This tutorial will move one step further from the previous tutorial *Edge-Cloud communication on WAN*. It is recommended to follow these tutorials in order, as some concepts or installations may be already covered. It will focus on an edge-edge architecture in which both edge robots are deployed on different networks (WAN) with access to the Internet, and communicate through a server hosted in the Cloud.

The scenario we are considering in this tutorial is the one where the edges are not directly connected (are under different LANs) and each robot does not have access to the other’s location and network (as they may be behind intermediate NATs or be part of dynamic networks). In this scenario, the Cloud server will work as a **TURN** (Traversal Using Relays around NAT), a.k.a **Repeater**. This kind of servers are meant to be accessed from every point in the WAN network, and forward the messages received from any edge to the rest of devices connected to the server. Thus, it creates a bridge between 2 networks that do not have direct access to each other, facilitating network configuration aspects such as NATs Traversals and dynamic addresses.

**Warning**: This tutorial is intended for WAN communication. However, if communication through a LAN is your only option, it is still possible to follow the tutorial by changing the ROS 2 Domain Ids so that each edge uses a different Domain (0 and 1). This way the ROS 2 nodes are logically isolated and will not discover other nodes out of their ROS 2 Domain.

Following, all the elements involved in this architecture will be studied, starting with the edge robots and continuing with the intermediate elements that enable communication over the Internet between each edge and the Cloud. One edge will work as a controller and the other as a robot (as already explained in previous tutorial).

The image below describes the scenario presented in this tutorial.

Several key elements can be observed in it:

1. **ROS 2 Application.** *Turtlesim* is the application used for this tutorial. *Turtlesim* is a ROS 2 application, first developed for ROS, aimed at teaching the basic concepts of ROS 2 such as publish/subscribe, services and actions. The edge robot will then be a *turtlesim_node*, which is a simulator of a robot making use of these communication methods.

2. **ROS 2 Controller.** A basic C++ application has been developed for this tutorial that sends publications under the topic on which the *turtlesim_node* listens. It has been developed a basic C++ application for this tutorial that
sends publications on the topic that the `turtlesim_node` listens. By means of these publications (commands) from the controller, and the feedback information that the controller receives from the `turtlesim_node`, it is possible to control this node automatically without the need for user intervention which facilitates the deployment of the scenario at hand. The key feature of the `DDS Router` is that it is easy to configure, allowing to connect different networks with ROS 2 applications without requiring to apply any changes to the developer's software or applications.

3. **ROS 2 Router / DDS Router.** `eProsima ROS 2 Router`, a.k.a `DDS Router`, is an end-user software application that enables the connection of distributed ROS 2 networks (see DDS Router documentation [here](#)). That is, ROS 2 nodes such as publishers and subscribers, or clients and services, deployed in one geographic location and using a dedicated local network will be able to communicate with other ROS 2 nodes deployed in different geographic areas on their own dedicated local networks as if they were all on the same network through the use of `DDS Router`.

This example presents two routers that enable Internet communication:

- **DDS Router Edge.** This is the DDS Router that is deployed on the edge robot side. This way it is possible for the robot to communicate out-of-the-box with an external server.

- **DDS Router Repeater.** It plays the TURN server role in the communication. It will expose some public network addresses to which the edge Routers could establish communication, and will forward the messages from one LAN to the other.

### Prerequisites

This tutorial will require three machines (`Robot 1`, `Controller 1` and `Cloud Server`) deployed on different networks (`LAN 1`, `LAN 2` and `Cloud`). It is possible to simulate the scenario by deploying everything needed on the same machine and three virtual networks but let’s focus on the case of a real deployment.

It is also necessary to have previously installed Vulcanexus using one of the following installation methods:

- Linux binary installation
- Linux installation from sources
- Docker installation
Deployment on LAN 1

First, let's deploy the `turtlesim_node` and DDS Router Edge on a machine on LAN 1.

Running `turtlesim_node` on Edge 1

Setup the Vulcanexus environment to have the `turtlesim_node` available. For this, there are two possible options:

1. Running the Vulcanexus Docker image.
   
   Run the Vulcanexus Docker image with:

   ```
   xhost local:root
docker run --privileged
                   -e DISPLAY=$DISPLAY
                   -v /tmp/.X11-unix:/tmp/.X11-unix
   ubuntu-vulcanexus:iron-desktop
   ```

   Then, within the container, source the Vulcanexus installation with:

   ```
   source /opt/vulcanexus/iron/setup.bash
   ```

2. Setting up the development environment on the local host. For this second option, it is necessary to have installed the `vulcanexus-iron-desktop` package, since this is the one that includes all the simulation tools, demos and tutorials.

   Source the following file to setup the Vulcanexus environment:

   ```
   source /opt/vulcanexus/iron/setup.bash
   ```

Once the environment has been setup using one of the above options, simply run the `turtlesim_node`.

```ros2 run turtlesim turtlesim_node```

And a popup window like the following should appear:

As can be seen, it is not necessary to perform any additional configuration in the ROS 2 application.

Running DDS Router Edge 1

Then, to run the DDS Router Edge configure the environment as in the previous step.

**Note:** If deploying Vulcanexus from the Docker image, note that you will need to have a configuration file (`config.yaml`) for the DDS Router Edge accessible from your Docker container.

This can be achieved by mounting a shared volume when launching the container, by copying the file from the local host to the container in case it is already running, or by editing a file from the Docker container itself.

Setup the Vulcanexus environment, either in a Docker container or on the local host, running the following command:

```source /opt/vulcanexus/iron/setup.bash```
Let’s create a DDS Router configuration file as the one shown below.

```
version: v3.0

participants:

- name: ROS_2_LAN_1
  kind: local

- name: Router_Client
  kind: wan
  connection-addresses:
    - ip: 123.123.123.123
      port: 45601
      transport: tcp
```

Next, the most relevant aspects of this configuration file are explained.

The participants are the interfaces of the DDS Router to communicate with other networks. In this case, we have two kinds of participants:

- **local**: this is a simple participant that communicates with all ROS 2 nodes it finds in domain 0. For more information about this participant please refer to the Simple Participant section of the DDS Router documentation.

- **wan**: it is a participant designed to communicate with a WAN Participant configured as server (repeater in this case). It uses the Initial Peers discovery mechanism to establish a point-to-point communication between two DDS entities, two DDS Routers in this case.
For the DDS Router Edge, a connection address shall be defined which must be the same as the one exposed by the Cloud Server.

Note: In this case, the DDS Router will forward all topics found in the network. However, it is important to mention that the ROS 2 topics relayed by the DDS Router can be filtered by configuring the allowlist and blocklist. If this is the case please refer to the DDS Router documentation for information on how to do this.

The following figure summarizes the deployment on the edge 1.

![Deployment Diagram](image)

Now, run the DDS Router with the configuration file created as an argument.

```bash
ddsrouter -c <path/to/file>/ddsrouter_edge_1.yaml
```

### Deployment of LAN 2

#### Running the turtlesim_square_move on Edge 2

Run the `turtlesim_square_move` in the Edge 2 machine, which is the controller of the Edge 1 `turtlesim_node`. This will send commands to the ROS 2 application to the edge to move the turtle and receive information about the current state of the turtle at any time.

A ROS 2 application that moves the turtle by drawing a square has been developed for this purpose. The application is based on the ROS 2 tutorials, but has been slightly modified in order to make it easier to understand and adapt to the ROS 2 and modern C++ programming methods.

Then, start by creating the workspace of this application and downloading the source code:

```bash
mkdir -p turtlesim_move_ws/src && cd turtlesim_move_ws/src
git clone --branch iron https://github.com/eProsima/vulcanexus.git
mv vulcanexus/code/turtlesim .
rm -rf vulcanexus
cd ..
```

Once created the workspace, source the Vulcanexus environment and build the `turtlesim_square_move` application.

4.12. Vulcanexus Cloud Tutorials
source /opt/vulcanexus/iron/setup.bash
colcon build

Note: Since the purpose of this tutorial is not to explain how to create a ROS 2 node, but rather, the communication of these in distributed environments, the code will not be discussed in detail. Stay tuned for new tutorials in which we will discuss how to configure ROS 2 nodes, publishers/subscribers, services and actions.

Then source the turtlesim_move_ws workspace:

source turtlesim_move_ws/install/setup.bash

And finally, run the application:

WAN

ros2 run docs_turtlesim turtlesim_square_move

LAN

ROS_DOMAIN_ID=1 ros2 run docs_turtlesim turtlesim_square_move

Note: As stated here, change the ROS 2 Domain Id if running the edge and cloud applications on the same LAN.

The important points to note in this application are the following:

- The control application sends the movement commands to the turtlesim_node through a publisher in the geometry_msgs/msg/Twist topic. This topic expresses the velocity at which the turtle has to move divided into linear velocity (linear) and angular velocity (angular).

- The application knows the position of the turtle on the map at any moment and whether it is stopped or moving by subscribing to the turtlesim/msg/pose topic. This topic provides information about the turtle's coordinates (x and y) and the turtle's rotation (theta). We can also know its linear and angular velocity (linear_velocity and angular_velocity).

Running DDS Router Edge 2

As the Repeater server is the same for both edges, the configuration of the DDS Router Edge 2 is very similar to the one for the DDS Router Edge 1. In this example both edges use different ports to communicate with the Repeater, simulating 2 different networks available in the Cloud. However this is not needed, and only one address could be used. The following snippet shows a configuration file (changing Domain for LAN scenarios):

WAN

version: v3.0

participants:

- name: ROS_2_LAN_2
  kind: local
- name: Router_Client

(continues on next page)
LAN

```yaml
kind: wan
connection-addresses:
  - ip: 123.123.123.123
    port: 45602
    transport: tcp

version: v3.0

participants:

- name: ROS_2_LAN_2
  kind: local
  domain: 1

- name: Router_Client
  kind: wan
  connection-addresses:
    - ip: 123.123.123.123
      port: 45602
      transport: tcp
```

Note: As stated here, set the ROS 2 Domain Id on the local participant in order to discover the turtlesim_square_move ROS 2 node.

Now, run the DDS Router with the configuration file created as an argument.

```
ddsrouter -c <path/to/file>/ddsrouter_edge_2.yaml
```

The following figure summarizes the deployment on the edge 2.
Deployment of Cloud Repeater

In order to communicate both edges, a DDS Router configured as *Repeater* is used, forwarding the messages from one edge to the other. This machine should be accessible from the Internet. In case this device is working under a NAT, check *previous tutorial* for more information about how to configure the NAT to be accessible from the outside. The following snippet shows the configuration file for this DDS Router:

```yaml
version: v3.0
participants:
  - name: Router_Repeater
    kind: wan
    repeater: true
    listening-addresses:
      - ip: 123.123.123.123
        port: 45601
        transport: tcp
      - ip: 123.123.123.123
        port: 45602
        transport: tcp
```

In this case, there is only one Participant configured as *repeater*. This Participant will wait for external Participants to communicate via Initial Peers from its *listening-addresses*. Once the discovery of clients occurs, this Repeater will forward the data from one edge to the other.

To run the DDS Router with the configuration file created as an argument, execute the following command after having sourced the Vulcanexus environment:

```
ddsrouter -c <path/to/file>/dds_router_repeater.yaml
```

*Note:* The Repeater Participant is not limited by number of listening-addresses, neither by number of edge Routers. It can open as many ports and interfaces as needed, and can forward messages from any number of edges, without re-sending redundant information or sending back any message.

## Results

If all the steps in this tutorial have been followed, the turtle in the *turtlesim_node* on the edge should move around creating a square, and the *turtlesim_square_move* should prompt the following traces:

```
root@dbf79a437eb3:/turtlesim_move_ws# ros2 run turtlesim turtlesim_square_move
[INFO] [1657870899.585667136] [turtlesim_square_move]: New goal [7.544445 5.544445, 0.000000]
[INFO] [1657870901.585656311] [turtlesim_square_move]: Reached goal
[INFO] [1657870901.585767260] [turtlesim_square_move]: New goal [7.448444 5.544445, 1.570796]
[INFO] [1657870905.685655744] [turtlesim_square_move]: Reached goal
```

(continues on next page)
4.12.8 Cloud ROS 2 MicroServices

- Background
- Prerequisites
- Deployment on LAN 1
  - Running Robot 1 on the edge
  - Running DDS Router Edge
  - Running more than one Edge
- Deployment on Cloud
  - Running Services on the Cloud
  - Running the DDS Router Cloud
**Background**

This tutorial will cover how to deploy a *Cloud Micro Service* architecture using *Vulcanexus*. Specifically, it will be shown how to deploy ROS 2 services in a Cloud device and access them from Node Clients in different edges. The idea is to maintain an out-of-the-box ROS 2 group of services accessible from any location and capable of executing high performance tasks, thus reducing the computational effort made by the edge nodes.

**Warning:** This tutorial is intended for WAN communication. However, if communication through a LAN is your only option, it is still possible to follow the tutorial by changing the ROS 2 Domain Ids so that each ROS 2 node uses a different Domain (0 and 1). This way the ROS 2 nodes are logically isolated and will not discover other nodes out of their ROS 2 Domain.

**Note:** This tutorial uses the ROS 2 Service communication method. Refer to ROS 2 Documentation for more information regarding ROS 2 Services, how they work and how to use them.

Following, all the elements involved in this architecture will be studied, starting with the edge robot, continuing with the controller hosted in the cloud also built as a ROS 2 node and concluding with the intermediate elements that enable communication over the Internet.

The image below describes the scenario presented in this tutorial.

![Diagram of Cloud Micro Service architecture](image)

Several key elements can be observed in it:

1. **ROS 2 Application.** The application used for this tutorial are trivial ROS 2 clients. These clients will connect to ROS 2 Services hosted in Cloud. This example will focus on *Robot 1* that will connect to services + and −.

2. **ROS 2 “Micro” Services.** This ROS 2 application will handle several ROS 2 servers, each providing a different service. Two basic Python applications have been developed for this tutorial, representing the server and client.
nodes in a service, which is nothing more than a solver of trivial mathematical operations. The functionality of the services is not relevant for this tutorial, they are only meant to exemplify a micro-service architecture. In the actual deployment of this architecture, the services could manage any kind of task. The services implemented are:

- addition a.k.a. + adds two integer numbers.
- subtraction a.k.a. - subtracts two integer numbers.

3. **ROS 2 Router / DDS Router.** *eProsima ROS 2 Router*, a.k.a *DDS Router*, is an end-user software application that enables the connection of distributed ROS 2 networks (see DDS Router documentation here). That is, ROS 2 nodes such as publishers and subscriptions, or clients and services, deployed in one geographic location and using a dedicated local network will be able to communicate with other ROS 2 nodes deployed in different geographic areas on their own dedicated local networks as if they were all on the same network through the use of *DDS Router*.

This example presents two routers that enable Internet communication:

- **DDS Router Edge.** This is the DDS Router that is deployed on the edge robot side. This way it is possible for the robot to communicate out-of-the-box with an external server.
- **DDS Router Cloud.** It plays the server role in the communication. It will expose a public network address to which the nodes connect to establish communication.

**Warning:** ROS 2 Services are (in principle) not meant to be provided by more than one node in the same network. The use of multiple servers on the same Service will result in a very low-performance system, as messages will be received and processed by every server.

The number of different services that could be hosted in the Cloud is not limited. The number of clients able to reach those servers is also unlimited, as well as the services that each device handles (edge or Cloud), allowing the same ROS 2 application to serve as a client or server of multiple services.

**Prerequisites**

This tutorial will require at least two machines (*Robot 1* and *Cloud Server*) deployed on different networks (*LAN 1* and *Cloud*). In order to recreate a more realistic scenario, there could be as many edge robots as desired without additional configurations. It is possible to simulate the scenario by deploying everything needed on the same machine and two virtual networks, but let’s focus on the case of a real deployment.

It is also necessary to have previously installed Vulcanexus using one of the following installation methods:

- Linux binary installation
- Linux installation from sources
- Docker installation
**Deployment on LAN 1**

First, let’s deploy the Client and DDS Router Edge on a machine on *LAN 1*.

**Running Robot 1 on the edge**

Run any of the clients for the ROS 2 Services in this machine. This will publish Request messages to every server listening in the network. Once the whole network has been established, those messages will arrive to the server hosted in cloud, and will answer with a Response.

In this tutorial, an example on how to implement some of these services (for servers and clients) is provided. Following are the ones used in this example, however the implementation of new and complex ones is straight-forward:

- **addition_service**
  - **Client**: generate 2 random numbers from 0 to 100 and send request to server. Command: `ros2 run clients_py client_addition`
  - **Server**: add 2 integers get in request, and send solution in response. Command: `ros2 run microservers_py server_addition`

- **subtraction_service**
  - **Client**: generate 2 random numbers from 0 to 100 and send request to server. Command: `ros2 run clients_py client_subtraction`
  - **Server**: subtract 2 integers get in request, and send solution in response. Command: `ros2 run microservers_py server_subtraction`

These files can be found at `/code/microservices` in [Vulcanexus Documentation Repository](https://github.com/eProsima/vulcanexus) as a Colcon python package. In order to install these applications, follow these steps:

1. Start by creating the workspace of this application and downloading the source code:
   ```bash
   mkdir -p clients_ws/src && cd clients_ws/src
   git clone --branch iron https://github.com/eProsima/vulcanexus.git
   mv vulcanexus/code/microservices/clients_py .
   rm -rf vulcanexus
   cd ..
   ```
2. Once created the workspace, source the Vulcanexus environment and build the `clients_py` package:
   ```bash
   source /opt/vulcanexus/iron/setup.bash
   colcon build
   ```
3. Then source the `clients_ws` workspace:
   ```bash
   source install/setup.bash
   ```
4. And finally, run the application:
   ```bash
   ros2 run clients_py client_addition # client_subtraction
   ```

**Note**: Since the purpose of this tutorial is not to explain how to create a ROS 2 node, but rather, the communication of these in distributed environments, the code will not be discussed in detail. This code has been developed following...
this ROS 2 tutorial and the comments along the code make it self-explanatory. Stay tuned for new tutorials where we will discuss how to configure ROS 2 nodes, publishers/subscribers, services and actions.

**Running DDS Router Edge**

Then, to run the DDS Router Edge configure the environment as in the previous step.

**Note:** If deploying Vulcanexus from the Docker image, note that you will need to have a configuration file (`config.yaml`) for the DDS Router Edge accessible from your Docker container. This can be achieved by mounting a shared volume when launching the container, by copying the file from the local host to the container in case it is already running, or by editing a file from the Docker container itself.

Setup the Vulcanexus environment, either in a Docker container or on the local host, running the following command:

```
source /opt/vulcanexus/iron/setup.bash
```

Let's create a DDS Router configuration file as the one shown below.

```yaml
version: v3.0
participants:
  - name: ROS_2_LAN_1
    kind: local
  - name: Router_Client
    kind: wan
    connection-addresses:
      - ip: 123.123.123.123
        port: 45678
        transport: tcp
```

Next, the most relevant aspects of this configuration file are explained.

The **participants** are the interfaces of the DDS Router to communicate with other networks. In this case, we have two kinds of participants:

- **local**: this is a simple participant that communicates with all ROS 2 nodes it finds. For more information about this participant please refer to the Simple Participant section of the DDS Router documentation.

- **wan**: it is a participant designed for the communication between two DDS Routers. It uses the Initial Peers discovery mechanism to establish a point-to-point communication between two DDS entities, two DDS Routers in this case.

For the DDS Router Edge, a connection address shall be defined which must be the same as the one exposed by the Cloud Server.

**Note:** In this case, the DDS Router will forward all topics found in the network. However, it is important to mention that the ROS 2 topics relayed by the DDS Router can be filtered by configuring the allowlist and blocklist. If this is the case please refer to the DDS Router documentation for information on how to do this.
To finish this step, run the DDS Router with the configuration file created as an argument.

```shell
ddsrouter -c <path/to/file>/ddsrouter_edge.yaml
```

### Running more than one Edge

Several edge robots can be deployed on the same scenario without affecting the functionality or configuration. Robot 2 shown in figure could be deployed using the exact configuration used for Robot 1. The services that each Robot connects to could be configured in the ROS 2 application running in each device. The scenario is not dependent on the ROS 2 Services in the network, neither the clients or servers running. This is a very flexible and scalable scenario.

### Deployment on Cloud

#### Running Services on the Cloud

In order to install and use the MicroServers, follow the instructions in section Running Robot 1 on the edge but use the package microservers_py instead of client_py (change these strings in every command).

Once the whole workspace has been installed and correctly sourced, each server could be run with the following commands:

**WAN**

```shell
ros2 run microservers_py server_addition # To run Addition (+) Server
```

# In different terminal or background

```shell
ros2 run microservers_py server_subtraction # To run Subtraction (-) Server
```

**LAN**

```shell
ROS_DOMAIN_ID=1 ros2 run microservers_py server_addition # To run Addition (+) Server
```

(continues on next page)
# In different terminal or background
ROS_DOMAIN_ID=1 ros2 run microservers_py server_subtraction # To run Subtraction (-)

...Server

Note: As stated here, change the ROS 2 Domain Id if running the edge and cloud applications on the same LAN.

Running the DDS Router Cloud

Configure transversal NAT on the network router

The first thing to do before starting to configure DDS Router is to configure the network router to allow a remote communication from the Internet to reach a specific device on the LAN, more specifically to expose an IP address and a port to the network that will be used by our DDS Router application.

This configuration will depend on your network router, but it should be similar to the one shown in the following image.

network

These rules are needed to authorise a remote communication from Internet to reach a specific device of your LAN. You can also define the ports(s) that this communication will use.

warning: make sure you have not filtered these ports in the firewall.

<table>
<thead>
<tr>
<th>status</th>
<th>application / service</th>
<th>internal port</th>
<th>external port</th>
<th>protocol</th>
<th>device</th>
<th>activate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Web Server (HTTP)</td>
<td>80</td>
<td>80</td>
<td>TCP</td>
<td>PC-1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DDS Router Cloud</td>
<td>45678</td>
<td>45678</td>
<td>TCP</td>
<td>192.168.2.100</td>
<td>✔</td>
</tr>
</tbody>
</table>

...
Configure the DDS Router Cloud

The DDS Router Cloud configuration file is quite similar to the DDS Router Edge configuration file, as can be seen below:

WAN

```yaml
version: v3.0

participants:
  - name: ROS_2_Cloud
    kind: local
  - name: Router_Server
    kind: wan
    listening-addresses:
      - ip: 123.123.123.123
      port: 45678
      transport: tcp
```

LAN

```yaml
version: v3.0

participants:
  - name: ROS_2_Cloud
    kind: local
    domain: 1
  - name: Router_Server
    kind: wan
    listening-addresses:
      - ip: 123.123.123.123
      port: 45678
      transport: tcp
```

Note: As stated here, set the ROS 2 Domain Id on the local participant in order to discover only those nodes required.

In this case there are also two participants, two communication interfaces for the DDS Router. The first one communicates the DDS Router with any ROS 2 node, while the second one enables to establish a communication channel with another DDS Router.

Although quite similar to the WAN participant in the DDS Router Edge instance, notice that this participant sets a listening address (listening-addresses), rather than a connection address. This is because it is the participant that waits for incoming communications since it has this network address exposed and accessible from the Internet.

To finish, as done in the previous steps, setup the Vulcanexus environment sourcing the `setup.bash` file and run the DDS Router Cloud with the above configuration.

```bash
source /opt/vulcanexus/iron/setup.bash
ddsrouter -c <path/to/file>/ddsrouter_cloud.yaml
```
The following figure summarizes the deployment on the Cloud.

Results

If all the steps in this tutorial have been followed, each client running in an edge will print in stdout messages such as the following one:

```
root@dbf79a437eb3:/clients_ws# ros2 run clients_py client_addition
[INFO] [1659078463.756352817] [AdditionMicroClient]: Service not available, waiting... again...
[INFO] [1659078528.309955103] [AdditionMicroClient]: Request sent, waiting for server result.
[INFO] [1659078530.288999295] [AdditionMicroClient]: Result of operation: for 63 + 40 = 103
```

While in the server side, messages as the one shown below should be printed:

```
root@dbf79a437eb3:/turtlesim_move_ws# ros2 run microservers_py server_addition
[INFO] [1659078530.288140374] [AdditionMicroServer]: Incoming request { 63 + 40 = 103 }
```
4.13 Vulcanexus Micro Tutorials

micro-ROS already provides several tutorials that can be also run within Vulcanexus. Please, visit micro-ROS tutorial webpage.

4.13.1 Custom Transports

This tutorial aims at providing step-by-step guidance for those users interested in creating micro-ROS custom transports, instead of using the ones provided by default in the micro-ROS tools set.

This tutorial starts from a previously created micro-ROS environment. Check the first steps of Getting started micro-ROS for instructions on how to create a micro-ROS environment for embedded platforms.

The micro-ROS middleware, eProsima Micro XRCE-DDS, provides a user API that allows interfacing with the lowest level transport layer at runtime, which enables users to implement their own transports in both the micro-ROS Client and micro-ROS Agent libraries.

Thanks to this, the Micro XRCE-DDS wire protocol can be transmitted over virtually any protocol, network or communication mechanism. In order to do so, two general communication modes are provided:

- **Stream-oriented mode**: the communication mechanism implemented does not have the concept of packet. HDLC framing will be used.
- **Packet-oriented mode**: the communication mechanism implemented is able to send a whole packet that includes an XRCE message.

These two modes can be selected by activating and deactivating the `framing` parameter in both the micro-ROS Client and the micro-ROS Agent functions, default defines available:

```
#define MICROROS_TRANSPORTS_FRAMING_MODE 1
#define MICROROS_TRANSPORTS_PACKET_MODE 0
```

**micro-ROS Client**

A full example can be found on micro-ROS demos repository custom_transports example.

An example on how to set these external transport callbacks in the micro-ROS Client API is:

```
#include <rmw_microros/rmw_microros.h>

...

struct custom_args {
    ...
}
```

(continues on next page)
struct custom_args args;

rmw_uros_set_custom_transport(
    MICROROS_TRANSPORTS_FRAMING_MODE, // Framing enabled here. Using Stream-oriented,
    (void *) &args,
    my_custom_transport_open,
    my_custom_transport_close,
    my_custom_transport_write,
    my_custom_transport_read
);

It is important to notice that in rmw_uros_set_custom_transport a pointer to custom arguments is set. This reference will be available on every transport callbacks.

In general, four functions must be implemented. The behaviour of these functions is slightly different, depending on the selected mode, in all of them transport->args holds the arguments passed through rmw_uros_set_custom_transport:

Open function

```c
bool my_custom_transport_open(uxrCustomTransport* transport)
{
    ...
}
```

This function should open and init the custom transport. It returns a boolean indicating if the opening was successful.

Close function

```c
bool my_custom_transport_close(uxrCustomTransport* transport)
{
    ...
}
```

This function should close the custom transport. It returns a boolean indicating if closing was successful.

Write function

```c
size_t my_custom_transport_write(
    uxrCustomTransport* transport,
    const uint8_t* buffer,
    size_t length,
    uint8_t* errcode)
{
    ...
}
```

This function should write data to the custom transport. It returns the number of bytes written.
• **Stream-oriented mode**: The function can send up to length bytes from buffer.

• **Packet-oriented mode**: The function should send length bytes from buffer. If less than length bytes are written, errcode can be set.

**Read function**

```c
size_t my_custom_transport_read(
    uxrCustomTransport* transport,
    uint8_t* buffer,
    size_t length,
    int timeout,
    uint8_t* errcode)
{
    ...
}
```

This function should read data from the custom transport. It returns the number of bytes read.

• **Stream-oriented mode**: The function should retrieve up to length bytes from the transport and write them into buffer in timeout milliseconds.

• **Packet-oriented mode**: The function should retrieve length bytes from transport and write them into buffer in timeout milliseconds. If less than length bytes are read, errcode can be set.

**micro-ROS Agent**

A full example can be found on Micro-XRCE-DDS-Agent repository custom_agent example.

The micro-ROS Agent profile for custom transports is enabled by default.

An example on how to set the external transport callbacks in the micro-ROS Agent API is:

```c
eprosima::uxr::Middleware::Kind mw_kind(eprosima::uxr::Middleware::Kind::FASTDDS);
eprosima::uxr::CustomEndPoint custom_endpoint;

// Add transport ending parameters
custom_endpoint.add_member<
    uint32_t>("param1");
custom_endpoint.add_member<
    uint16_t>("param2");
custom_endpoint.add_member<
    std::string>("param3");

eprosima::uxr::CustomAgent custom_agent(  
    "my_custom_transport",
    &custom_endpoint,
    mw_kind,
    true, // Framing enabled here. Using Stream-oriented mode.
    my_custom_transport_open,
    my_custom_transport_close,
    my_custom_transport_write
    my_custom_transport_read);

custom_agent.start();
```

As in the Client API, four functions should be implemented. The behavior of these functions is sightly different depending on the selected mode.
**CustomEndPoint**

The custom_endpoint is an object of type eprosima::uxr::CustomEndPoint and it is in charge of handling the endpoint parameters. The Agent, unlike the Client, can receive messages from multiple Clients so it must be able to differentiate between them.

Therefore, the eprosima::uxr::CustomEndPoint should be provided with information about the origin of the message in the read callback, and with information about the destination of the message in the write callback.

In general, the members of a eprosima::uxr::CustomEndPoint object can be unsigned integers and strings.

CustomEndPoint defines three methods:

**Add member**

```cpp
bool eprosima::uxr::CustomEndPoint::add_member<const KIND*>(const std::string& member_name);
```

This function allows to dynamically add a new member to the endpoint definition.

It returns true if the member was correctly added, false if something went wrong (for example, if the member already exists).

- **KIND**: To be chosen from: uint8_t, uint16_t, uint32_t, uint64_t, uint128_t or std::string.
- **member_name**: The tag used to identify the endpoint member.

**Set member value**

```cpp
void eprosima::uxr::CustomEndPoint::set_member_value(const std::string& member_name, const KIND* & value);
```

This function sets the specific value (numeric or string) for a certain member, which must previously exist in the CustomEndPoint.

- **member_name**: The member whose value is going to be modified.
- **value**: The value to be set, of KIND: uint8_t, uint16_t, uint32_t, uint64_t, uint128_t or std::string.

**Get member**

```cpp
const KIND* & eprosima::uxr::CustomEndPoint::get_member(const std::string& member_name);
```

This function gets the current value of the member registered with the given parameter.

The retrieved value might be an uint8_t, uint16_t, uint32_t, uint64_t, uint128_t or std::string.

- **member_name**: The CustomEndPoint member name whose current value is requested.

**Open function**

```cpp
eprosima::uxr::CustomAgent::InitFunction my_custom_transport_open = [&]() -> bool {
    ... 
}
```

This function should open and init the custom transport. It returns a boolean indicating if the opening was successful.
Close function

```cpp
eprosima::uxr::CustomAgent::FiniFunction my_custom_transport_close = [&]() -> bool {
    ...
}
```

This function should close the custom transport. It returns a boolean indicating if the closing was successful.

Write function

```cpp
eprosima::uxr::CustomAgent::SendMsgFunction my_custom_transport_write = [&] (const 
    eprosima::uxr::CustomEndPoint* destination_endpoint,
    uint8_t* buffer,
    size_t length,
    eprosima::uxr::TransportRc& transport_rc) -> ssize_t {
    ...
}
```

This function should write data to the custom transport, must use the `destination_endpoint` members to set the data destination, returns the number of bytes written and set `transport_rc` indicating the result of the operation.

- **Stream-oriented mode**: The function can send up to `length` bytes from `buffer`.
- **Packet-oriented mode**: The function should send `length` bytes from `buffer`. If less than `length` bytes are written, `transport_rc` can be set.

Read function

```cpp
eprosima::uxr::CustomAgent::RecvMsgFunction my_custom_transport_read = [&] (const 
    eprosima::uxr::CustomEndPoint* source_endpoint,
    uint8_t* buffer,
    size_t length,
    int timeout,
    eprosima::uxr::TransportRc& transport_rc) -> ssize_t {
    ...
}
```

This function should read data to the custom transport, must fill `source_endpoint` members with data source, returns the number of bytes read and set `transport_rc` indicating the result of the operation.

- **Stream-oriented mode**: The function should retrieve up to `length` bytes from the transport and write them into `buffer` in `timeout` milliseconds.
- **Packet-oriented mode**: The function should retrieve `length` bytes from the transport and write them into `buffer` in `timeout` milliseconds. If less than `length` bytes are read, `transport_rc` can be set.
4.13.2 Creating custom types

This tutorial aims at providing step-by-step guidance for those users interested in adding custom message definition to the micro-ROS build. Those instructions can be used to extend the type definition for topics, services and actions provided by ROS 2 and micro-ROS.

Further information can be found in Implementing custom interfaces.

Note: This tutorial starts from a previously created micro-ROS environment. Check the first steps of Getting started micro-ROS for instructions on how to create a micro-ROS environment for embedded platforms.

For creating custom types in a Build System Component, check the instructions in each component repository. For example Renesas e2 Studio Readme or ST Micro ST Cube IDE/MX.

Simple types

Once the micro-ROS workspace is created, navigate to firmware/mcu_ws and create a new package for the custom messages:

```bash
  cd firmware/mcu_ws
  ros2 pkg create --build-type ament_cmake my_custom_message
  cd my_custom_message
  mkdir msg
  touch msg/MyCustomMessage.msg
```

In the generated CMakeLists.txt file the following lines shall be added just before ament_package() line:

```bash
  ...
  find_package(rosidl_default_generators REQUIRED)
  rosidl_generate_interfaces(${PROJECT_NAME}
    "msg/MyCustomMessage.msg"
  )
  ...
```

In the generated package.xml file the following lines shall be added:

```xml
  ...
  <build_depend>rosidl_default_generators</build_depend>
  <exec_depend>rosidl_default_runtime</exec_depend>
  <member_of_group>rosidl_interface_packages</member_of_group>
  ...
```

The content of the msg/MyCustomMessage.msg file contains the message definition. For example:
Now, the micro-ROS workspace workspace can be built as usual. As explained in Getting started micro-ROS, the `ros2 run micro_ros_setup build_firmware.sh` command will build all packages located inside `mcu_ws`. In the micro-ROS application code the new message type can be used as usual:

```
#include <my_custom_message/msg/my_custom_message.h>
...

my_custom_message__msg__MyCustomMessage msg = {0};

msg.byte_test = 3;
msg.uint32_test = 42;
...

rclc_publisher_init_default(&publisher, &node, ROSIDL_GET_MSG_TYPE_SUPPORT(my_custom_message, msg, MyCustomMessage), "my_custom_publisher");
rcl_publish(&publisher, &msg, NULL);
...
```

### Type composition

It is possible to create custom types that compose members from another ROS 2 message type packages. For example a member with type `Point32` from the ROS 2 package `geometry_msgs`.

First of all, the dependency shall also be included in the `CMakeLists.txt`:

```
...  
find_package(rosidl_default_generators REQUIRED)
find_package(geometry_msgs REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}  
  "msg/MyCustomMessage.msg"
```

(continues on next page)
The dependency shall be included in package.xml:

... 
<build_depend>rosidl_default_generators</build_depend> 
<exec_depend>rosidl_default_runtime</exec_depend> 
<member_of_group>rosidl_interface_packages</member_of_group> 
<depend>geometry_msgs</depend> 
... 

At this point, message definition in msg/MyCustomMessage.msg can now include types from the geometry_msgs package:

... 
int64 int64_test 
uint64 uint64_test 
geometry_msgs/Point32 point32_test 
... 

And finally, the new member can be accessed in the custom type:

#include <my_custom_message/msg/my_custom_message.h> 
...

my_custom_message__msg__MyCustomMessage msg; 

msg.byte_test = 3; 
msg.uint32_test = 42; 

msg.point32_test.x = 1.23; 
msg.point32_test.y = 2.31; 
msg.point32_test.z = 3.12; 
...

Note: More details on how to handle micro-ROS types memory can be found on [Handling messages memory tutorial](#).

### 4.13.3 Reconnections and liveliness

This tutorial aims at providing insight on mechanism for handling reconnections between the micro-ROS Agent and Client and assert the connection liveliness.
This features can be useful in multiple situations when the Agent or the Client are no longer available or the communication channel between them is broken. Those situations can be, among others:

- Transport faults: Detect transport fault or disconnections and act accordingly.
- Agent or Client restarts: Make the system resilient to a possible restart on the MCU or Agent side.
- Client plug and play: Useful on systems with multiple MCU boards that are attached on demand.

### Client side: Ping API

micro-ROS Client default rmw layer `rmw_microxrcedds_c` offers a ping utility to test the connection with the Agent. This ping utility can be used at any stage of the micro-ROS application, allowing users to check the Agent availability before attempting to initialize a micro-ROS session or create any entities.

Usage details on this utility can be found in the [Middleware API: Ping Agent](#) tutorial.

### State machine approach

Reconnections can be handled manually on the micro-ROS Client side using the Agent ping functionality with the following sequence:

1. Wait until Agent is reachable.
2. Create required micro-ROS entities.
3. Use the micro-ROS API as usual: spin entities, publish, etc.
4. When a failure is detected in steps 2 or 3, destroy the created entities and go back to the first step.

This approach allows a micro-ROS Client to handle Agent restarts or to follow a plug and play approach, where the micro-ROS app will only run when the Agent connection is available. The following code shows an example of this sequence:

```c
// Timeout for each ping attempt
const int timeout_ms = 100;

// Number of ping attempts
const uint8_t attempts = 1;

// Spin period
const unsigned int spin_timeout = RCL_MS_TO_NS(100);

// Enum with connection status
enum states {
    WAITING_AGENT,
    AGENT_AVAILABLE,
    AGENT_CONNECTED,
    AGENT_DISCONNECTED
} state;

while (true)
{
    switch (state)
    {
    case WAITING_AGENT:
```

(continues on next page)
// Check for agent connection
state = (RMW_RET_OK == rmw_uros_ping_agent(timeout_ms, attempts)) ? AGENT AVAILABLE : WAITING_AGENT;
break;

case AGENTAVAILABLE:
  // Create micro-ROS entities
  state = (true == create_entities()) ? AGENT_CONNECTED : WAITING_AGENT;

  if (state == WAITING_AGENT)
  {
    // Creation failed, release allocated resources
    destroy_entities();
  }
  break;

case AGENTCONNECTED:
  // Check connection and spin on success
  state = (RMW_RET_OK == rmw_uros_ping_agent(timeout_ms, attempts)) ? AGENT CONNECTED : AGENT_DISCONNECTED;

  if (state == AGENT_CONNECTED)
  {
    rclc_executor_spin_some(&executor, spin_timeout);
  }
  break;

case AGENT_DISCONNECTED:
  // Connection is lost, destroy entities and go back to first step
  destroy_entities();
  state = WAITING_AGENT;
  break;

default:
  break;
}

A working example with this approach can be found on micro-ROS for Arduino repository micro-ros_reconnection example.

Agent side: Hard liveliness check

The main problem with the previous section’s method is that entity destruction always happens on micro-ROS Client’s request. This implies that other ROS 2 entities will not be aware of the micro-ROS Client destruction.

The Hard Liveliness Check mechanism allows the micro-ROS Agent to ping the Client periodically. This way, the Agent will take care of ensuring that the micro-ROS client is alive and will destroy the created entities if a certain timeout happens without any response from the Client side. This means that the nodes, publishers, subscribers (and any other entity) created by the Client will be removed from the ROS 2 graph.

This mechanism does not have a penalty on the application throughput, as it will avoid sending ping messages if the Agent is receiving data from the Client.
In other cases, the micro-ROS Client shall spin an executor to give a response to the Agent liveness check messages, an empty executor can be used for this purpose.

**Note:** Note that the Client shall also be aware of the disconnection to create the micro-ROS entities again, this can be achieved by including the previous section approach.

### Configuration

This feature is enabled by default in the micro-ROS Agent and **must be enabled** by means of *colcon.meta* parameters in the micro-ROS Client:

- `UCLIENT_HARD_LIVELINESS_CHECK`: Enable hard liveness check
- `UCLIENT_HARD_LIVELINESS_CHECK_TIMEOUT`: Configure connection timeout in milliseconds (Default value: 10000).

Example configuration on *colcon.meta* file:

```yaml
# colcon.meta example with Hard Liveliness Check configuration
{
   "names": {
      "microxrcedds_client": {
         "cmake-args": [
            "-DUCLIENT_HARD_LIVELINESS_CHECK=ON",
            "-DUCLIENT_HARD_LIVELINESS_CHECK_TIMEOUT=5000"
         ]
      }
   }
}
```

### 4.13.4 Integrating micro-ROS

Integrating micro-ROS in a platform is a process that highly depends on the target platform. In general, micro-ROS provides ready-to-use solutions for integrating the micro-ROS Client library in multiple platforms and build systems, more information about this micro-ROS modules can be found in *Build System Components* section.

This tutorial aims to go beyond those modules and provide some ideas on how to integrate micro-ROS in a new platform. This task can be divided in two main parts: generating the micro-ROS Client library and header directory and linking it against the target embedded application.

It is important to note that micro-ROS Client library is designed to be platform independent. This means that the library can be built as a standalone library with the only requirement of using the toolchain and *libc* implementation of the target platform.

This tutorial will cover the former topics along these sections:

Table of Contents

- *Generating a micro-ROS Client library*
  - micro-ROS generate_lib script
  - Creating a custom build script
• Integrating a custom build system
  – Bare gcc approach
  – Makefile
  – CMake

• micro-ROS system dependencies
  – Time reference
  – Transport layer
  – Allocators

Generating a micro-ROS Client library

The micro-ROS Client library, in most cases, is compound of:

• A static library built with an specific toolchain. Normally a .a file.

• A include folder where all the required headers are located. Normally a include folder.

Given that, most of common build system tools such as CMake or Make will be able to link against the static library and use the include folder to compile the application.

In order to generate those two components, two approaches are provided: using a micro-ROS tool for generating them or creating a custom script for handling this build.

micro-ROS generate_lib script

Note: This sections starts from a previously created micro-ROS environment. Check the first steps of Getting started micro-ROS for instructions on how to create a micro-ROS environment for embedded platforms.

The micro_ros_setup tool provides a script for generating and building the micro-ROS Client library according to a specific configuration and build parameters.

The following command will download all the required packages:

```
ros2 run micro_ros_setup create_firmware_ws.sh generate_lib
```

For configuring both the micro-ROS library and the build process a colcon.meta file and a CMake toolchain are required.

An example of a colcon.meta file can be:

```json
{
  "names": {
    "tracetools": {
      "cmake-args": [
        "-DTRACETOOLS_DISABLED=ON",
        "-DTRACETOOLS_STATUS_CHECKING_TOOL=OFF"
      ],
      "rosidl_typesupport": {
      }
    }
  }
}
```

(continues on next page)
An example of a CMake toolchain for a Cortex-M3 platform can be:

```cmake
set(CMAKE_SYSTEM_NAME Generic)
set(CMAKE_CROSSCOMPILING 1)
set(CMAKE_TRY_COMPILE_TARGET_TYPE STATIC_LIBRARY)

# SET HERE THE PATH TO YOUR C99 AND C++ COMPILERS
```

(continues on next page)
set(CMAKE_C_COMPILER arm-none-eabi-gcc)
set(CMAKE_CXX_COMPILER arm-none-eabi-g++)

set(CMAKE_C_COMPILER_WORKS 1 CACHE INTERNAL "")
set(CMAKE_CXX_COMPILER_WORKS 1 CACHE INTERNAL "")

# SET HERE YOUR BUILDING FLAGS
set(FLAGS "-O2 -ffunction-sections -fdata-sections -fno-exceptions -mcpu=cortex-m3 -mthumb -nostdlib --param max-inline-insns-single=500 -DF_CPU=84000000L" CACHE STRING "FORCE")
set(CMAKE_C_FLAGS_INIT "-std=c11 ${FLAGS} -DCLOCK_MONOTONIC=0 -D'_attribute__(x)='"_" "=" CACHE STRING "FORCE")
set(CMAKE_CXX_FLAGS_INIT "-std=c++11 ${FLAGS} -fno-rtti -DCLOCK_MONOTONIC=0 -D'_attribute__(x)='"_" "=" CACHE STRING "FORCE")

Once both files are ready, the micro-ROS library can be generated and built using the following command:

```
ros2 run micro_ros_setup build_firmware.sh $(pwd)/my_custom_toolchain.cmake $(pwd)/my_custom_colcon.meta
```

### Creating a custom build script

A basic understanding on how to proceed can be extracted analyzing the code of the custom library generation script explained above.

But in general the following points shall be taken into account:

---

**Note:** When following this instructions sourcing a ROS 2 environment shall be avoided in order to avoid mixing the ROS 2 build system with the micro-ROS build system.

1. Create a micro-ROS development environment with the following packages in the correct branch:
   - *ament_cmake* ([https://github.com/ament/ament_cmake](https://github.com/ament/ament_cmake))
   - *ament_lint* ([https://github.com/ament/ament_lint](https://github.com/ament/ament_lint))
   - *googletest* ([https://github.com/ament/googletest](https://github.com/ament/googletest))
   - *ament_cmake_ros* ([https://github.com/ros2/ament_cmake_ros](https://github.com/ros2/ament_cmake_ros))
   - *ament_index* ([https://github.com/ament/ament_index](https://github.com/ament/ament_index))

2. Build this environment locally using `colcon build --cmake-args -DBUILD_TESTING=OFF`

3. Source the local environment using `source install/local_setup.bash`

4. Create a new micro-ROS workspace and clone the micro-ROS Client packages inside it

---

4.13. Vulcanexus Micro Tutorials
5. Make sure that the following packages are removed or ignored:

- rosidl/rosidl_typesupport_introspection_cpp
- rcl_logger/rcl_logger_log4cxx
- rcl_logger/rcl_logger_spdlog
- rcl/rclc_examples
- rcl/rcl_yaml_param_parser

6. Build the micro-ROS workspace using colcon, your required colcon.meta and your custom compiler flags using a CMake toolchain file my_toolchain.cmake:

```
colcon build
  --merge-install
  --packages-ignore-regex=.*_cpp
  --metas my_colcon.meta
  --cmake-force-configure
  --cmake-clean-cache
  --cmake-args
    "--no-warn-unused-cli"
  --log-level=ERROR
  -DCMAKE_POSITION_INDEPENDENT_CODE:BOOL=OFF
  -DTHIRDPARTY=ON
  -DBUILD_SHARED_LIBS=OFF
  -DBUILD_TESTING=OFF
  -DCMAKE_BUILD_TYPE=$BUILD_TYPE
  -DCMAKE_TOOLCHAIN_FILE=my_toolchain.cmake
  -DCMAKE_VERBOSE_MAKEFILE=ON;
```
7. (Optional) Merge the generated .a libraries using `ar` utility.

**Integrating a custom build system**

At this point, the generated .a and include folder generated in the previous step shall be linked against a micro-ROS application.

Some approaches for integrating micro-ROS Client library on a platform build system can be:

**Bare gcc approach**

Using a common gcc command line, the following steps can be followed:

```
[TRIPLET PREFIX]-gcc -o microros_app.elf [COMPILER FLAGS] -I[MICROROS INCLUDE FOLDER] -lmicroros.a
```

**Note:** Note that [COMPILER FLAGS] shall be the same when generating the micro-ROS Client library and when building the micro-ROS application.

**Makefile**

An example on how to integrate micro-ROS Client library in a Make build system can be found in the micro-ROS app for TI Tiva C Series.

**CMake**

An example on how to integrate micro-ROS Client library in a CMake build system can be found in the micro-ROS example for Mbed RTOS.

**micro-ROS system dependencies**

There are three points where micro-ROS Client library needs to use functionality of the target platform beyond the libc implementation:

- Obtaining a time reference
- Configuring the transport layer
- Dealing with memory allocation
Time reference

In order to operate in a time-based approach, the micro-ROS library will need at link time an implementation of the function `int clock_gettime(clockid_t, struct timespec *)` from the POSIX specification.

This function will need to fill `struct timespec *` input argument implement with a monotonic time reference. In the case that the target platform does not provide this function, it is possible to implement it at application level and let the linker to resolve the symbol when linking the micro-ROS Client library.

A reference implementation can be:

```c
#include <sys/time.h>

#define USEC_IN_SEC 1000000
#define NSEC_IN_USEC 1000

int clock_gettime(clockid_t clock_id, struct timespec *tp)
{
    (void) clock_id;

    uint64_t microseconds_elapsed = my_platform_get_microseconds();

    // Handle here possible rollovers of your platform timers if required.
    tp->tv_sec = microseconds_elapsed / USEC_IN_SEC;
    tp->tv_nsec = (microseconds_elapsed % USEC_IN_SEC) * NSEC_IN_USEC;

    return 0;
}
```

Transport layer

micro-ROS Client will need a transport implementation for communication with the micro-ROS Agent.

Details on how to implement this transports can be found in Custom Transports tutorial.

Note: This tutorial explains how to implement custom transports in both micro-ROS Client and Agent side. In the most common use case both parts are communicated using serial ports or UDP sockets. That means that only Client side transport shall be implemented and the Agent side transport can be used as it is provided.

Allocators

More details about micro-ROS allocators are provided at Memory management allocators tutorial.
4.13.5 Memory management

- Allocators
- Middleware memory
  - eProsima Micro XRCE-DDS Client
  - Entity creation
  - Middleware related memory
  - Entity Names
- Message memory
  - Sequence types
  - Compound types
  - Sequences of compound types

micro-ROS Client provides full control over the memory usage during build, configuration and runtime. This is one of the most important requisites in order to fit in low resources system and to guarantee a hard real-time behaviour.

Memory management in micro-ROS Client can be configured at multiple levels so the user can use different mechanisms for fitting its requirements within the micro-ROS environment.

In general, micro-ROS by default will use:
- Static allocated memory in **build time**
- A bounded stack memory consumption
- A bounded amount of **dynamic memory** during entity creation and destruction (configuration time)
- **Zero dynamic memory** during runtime

During a micro-ROS Client application development the user is able to configure the memory at multiple levels. Along the following sections, those levels are analyzed in detail.

Allocators

As in the ROS 2 stack, micro-ROS dynamic memory allocators can be customized at runtime. By default those allocators relies on the libc implementation of malloc, calloc, realloc and free functions. However, in some platforms those functions are not available or not encouraged to be used and they can be replaced by platform specific functions. One example of this situation is FreeRTOS allocators.

An example on how to set custom allocators at runtime when using rcutils API is:

```c
allocator = rcutils_get_zero_initialized_allocator();
allocator.allocate = custom_allocate;
allocator.deallocate = custom_deallocate;
allocator.reallocate = custom_reallocate;
allocator.zero_allocate = custom_zero_allocate;
rcutils_set_default_allocator(&allocator);
```

A reference implementation of those allocators is:
void * custom_allocate(size_t size, void * state)
{
    // Allocate and return a memory chunk of `size` bytes.
}

void custom_deallocate(void * pointer, void * state)
{
    // Deallocate memory chunk pointed by `pointer`.
}

void * custom_reallocate(void * pointer, size_t size, void * state)
{
    // Reallocate memory chunk pointed by `pointer` to `size` bytes.
}

void * custom_zero_allocate(size_t number_of_elements, size_t size_of_element, void * state)
{
    // Allocate and return a memory chunk of `number_of_elements * size_of_element` bytes,
    // filled with zeros.
}

One example implementation of the most basic allocator that targets platforms where no libc allocators are available is:

static uint8_t heap[HEAP_SIZE];
static size_t current_pointer = 0;

void free_all_heap()
{
    current_pointer = 0;
}

void assert_position()
{
    if (current_pointer >= sizeof(heap)) {
        // Handle memory error
        while(1){};
    }
}

#define SYSTEM_ALIGNMENT 4

size_t align_size(size_t size)
{
    if (size % SYSTEM_ALIGNMENT != 0) {
        size += SYSTEM_ALIGNMENT - (size % SYSTEM_ALIGNMENT);
    }
    return size;
}

void * custom_allocate(size_t size, void * state)
{
size = align_size(size);
size_t p = current_pointer;
current_pointer += size;
assert_position();
return (void *) &heap[p];
}

void custom_deallocate(void * pointer, void * state)
{
    (void) state;
    (void) pointer;
}

void * custom_reallocate(void * pointer, size_t size, void * state)
{
    size = align_size(size);
    size_t p = current_pointer;
current_pointer += size;
    // Careful! pointer may have less than size memory, garbage can be copied!
    memcpy(&heap[p], pointer, size);
    assert_position();
    return (void *) &heap[p];
}

void * custom_zero_allocate(size_t number_of_elements, size_t size_of_element, void * state)
{
    size_t size = number_of_elements * size_of_element;
    size = align_size(size);
    size_t p = current_pointer;
current_pointer += size;
    memset(&heap[p], 0, size);
    assert_position();
    return (void *) &heap[p];
}

Note: This is a naive implementation of an allocator that does not allows memory deallocation. User shall be aware of using free_all_heap() when the micro-ROS entities are no longer required.
Middleware memory

By default micro-ROS uses an RMW based on eProsima Micro XRCE-DDS Client. This RMW implementation is known as rmw_microxrcedds and its main purpose in terms of memory is to avoid dynamic memory allocation and allowing the user to configure the memory usage during build time.

eProsima Micro XRCE-DDS Client

In the lower level of the middleware layers, the user can configure the maximum transfer unit of eProsima Micro XRCE-DDS Client by means of setting one of the following flags for microxrcedds_client package in the colcon.meta file:

- `UCLIENT_UDP_TRANSPORT_MTU`: Maximum transfer unit for UDP transport. Default value: 512 bytes.
- `UCLIENT_TCP_TRANSPORT_MTU`: Maximum transfer unit for TCP transport. Default value: 512 bytes.
- `UCLIENT_SERIAL_TRANSPORT_MTU`: Maximum transfer unit for serial transport. Default value: 512 bytes.
- `UCLIENT_CUSTOM_TRANSPORT_MTU`: Maximum transfer unit for custom transport. Default value: 512 bytes.

**Note:** Note that although many micro-ROS ports use UDP or Serial transport, most of them are implemented over Custom transport API. Therefore, the maximum transfer unit shall be set using `UCLIENT_CUSTOM_TRANSPORT_MTU`.

The rest of configuration parameters at middleware level are located in rmw_microxrcedds package configuration. The following parameters can be configured when building the micro-ROS Client library via colcon.meta file:

Entity creation

By means of the following build flags, the user can configure the maximum number of entities that can be created during the micro-ROS Client execution:

- `RMW_UXRCE_MAX_SESSIONS`: Maximum number of XRCE-DDS sessions. Default value: 1.
- `RMW_UXRCE_MAX_SUBSCRIPTIONS`: Maximum number of micro-ROS subscriptions. Default value: 4.
- `RMW_UXRCE_MAX_SERVICES`: Maximum number of micro-ROS services. Default value: 4.
- `RMW_UXRCE_MAX_GUARD_CONDITION`: Maximum number of micro-ROS guard conditions (used for timers among other things). Default value: 4.
- `RMW_UXRCE_MAX_TOPICS`: Maximum number of micro-ROS topics. Default value: 4. If set to -1 the value will be calculated as `RMW_UXRCE_MAX_TOPICS = RMW_UXRCE_MAX_PUBLISHERS + RMW_UXRCE_MAX_SUBSCRIPTIONS + RMW_UXRCE_MAX_NODES`.

In the default configuration, micro-ROS Client will not be able to create more entities than the ones specified above. If dynamic memory usage is allowed, by means of the following flag, the user can allow on-demand entity creating using dynamic memory when required. This dynamic memory usage uses micro-ROS allocators.

- `RMW_UXRCE_ALLOW_DYNAMIC_ALLOCATIONS`: Allow dynamic memory allocations when creating micro-ROS entities. Default value: OFF.
Middleware related memory

By means of the following build flags, the user can configure the behavior of communication buffers:

- **RMW_UXRCE_STREAM_HISTORY**: Maximum number of buffers of size UCLIENT_[XXX]_TRANSPORT_MTU that the XRCE-DDS layer is provided with. **It shall be power of 2**. Default value: 4.

- **RMW_UXRCE_STREAM_HISTORY_INPUT**: Override for RMW_UXRCE_STREAM_HISTORY for input buffers. Default value: RMW_UXRCE_STREAM_HISTORY.

- **RMW_UXRCE_STREAM_HISTORY_OUTPUT**: Override for RMW_UXRCE_STREAM_HISTORY for output buffers. Default value: RMW_UXRCE_STREAM_HISTORY.

- **RMW_UXRCE_MAX_HISTORY**: Maximum number of slots for storing incoming data between wait() and take() operations at RMW layer. It size is UCLIENT_[XXX]_TRANSPORT_MTU * RMW_UXRCE_STREAM_HISTORY[_INPUT] Default value: 8.

---

**Note:** When using Best Effort entities, the maximum serialized size of a topic shall fit in UCLIENT_[XXX]_TRANSPORT_MTU both for outgoing and incoming data.

When using Reliable entities, the maximum serialized size of a topic shall fit in UCLIENT_[XXX]_TRANSPORT_MTU * RMW_UXRCE_STREAM_HISTORY both for outgoing and incoming data.

When receiving data, and all RMW_UXRCE_MAX_HISTORY are occupied, the reception entity behavior is determined by History QoS.

---

Entity Names

Regarding the name of entities, topics and types, the user can configure the maximum length of the name by means of the following flags:

- **RMW_UXRCE_NODE_NAME_MAX_LENGTH**: Maximum number of characters for a node name. Default value: 60.

- **RMW_UXRCE_TOPIC_NAME_MAX_LENGTH**: Maximum number of characters for a topic name. Default value: 60.

- **RMW_UXRCE_TYPE_NAME_MAX_LENGTH**: Maximum number of characters for a type name. Default value: 100.

---

Message memory

Message memory handling is an important part of the micro-ROS Client memory handling due to the fact that **micro-ROS Client does not initialized by default the type memory**. This means that the user must initialize the type memory before using it. This consideration needs to be taken into account both for outgoing and incoming messages.

---

**Note:** micro-ROS provides an API for initializing the type memory that can be found in **micro-ROS Types Utilities**. This section provides an explanation of micro-ROS type memory handling in the case that this API is not available or not used for some reason.

A message type, both used for topics or services, is composed defined in a .msg, .srv or .action file. Each one of those files will describer members of the type that shall be one of the following types:

- **Basic type**: integers, floats, booleans, etc.

- **Compound type**: another ROS 2 defined type.

- **Array type**: fixed size arrays of basic or compound types.
• **Sequence type**: variable size sequences of basic or compound types.

For example, the following .msg can be described as follows:

```plaintext
# MyType.msg
std_msgs/Header header
int32[] values
float64 duration
int8[10] coefficients
string name
```

- the member `duration` is a **basic type** member.
- the member `values` is a **sequence type** member because it has a unbounded sequence of int32, in this case.
- the member `coefficients` is an **array type** member because it has a bounded sequence of 10 units of int8, in this case.
- the member `header` is an **compound type** member because it refers to type described in the same or other ROS 2 package.
- the member `name` is an **string type** member and should be understood as a char[] (sequence type member).

When dealing with the micro-ROS types support the developer needs to take into account how this message is going to be handled in the C99 API of micro-ROS. In general, the micro-ROS types support will create a C99 **struct** representation of the message:

```c
typedef struct mypackage__msg__MyType
{
    std_msgs__msg__Header header;
    rosidl_runtime_c__int32__Sequence values;
    double duration;
    int8 coefficients[10];
    rosidl_runtime_c__String name;  // equal to rosidl_runtime_c__char__Sequence
} mypackage__msg__MyType;
```

When in an application instances a variable of this type, for example `mypackage__msg__MyType mymsg;`, it is ensured that:

- `mymsg.coefficients` has a C array of int8.
- `mymsg.duration` is a double member.

But no memory is guaranteed to be allocated automatically for other members.

## Sequence types

A **sequence type member** is an especial type member that hosts a pointer `data`, a `size` and a `capacity` value. The pointer should have memory for storing up to `capacity` values and `size` member shows how many element are currently in the sequence. Usually in micro-ROS, the user is in charge of assigning memory and values to this sequence members.

In the case of the previous example `MyType.msg`, the `values` sequence member is represented in C99 as this struct:

```c
typedef struct rosidl_runtime_c__int32__Sequence
{
    int32_t* data;    /* The pointer to an array of int32 */
    size_t size;      /* The number of valid items in data */
} rosidl_runtime_c__int32__Sequence;
```
In that sense, a developer that instantiate a `mypackage__msg__MyType` `mymsg;` variable, should ensure that `mymsg.values.data` has memory for storing up to `mymsg.values.capacity` values and `mymsg.values.size` shows how many element are currently in the sequence, as shown in the following example:

```c
mypackage__msg__MyType mymsg;

// mymsg.values.data is NULL or garbage now
// mymsg.values.size is 0 or garbage now
// mymsg.values.capacity is 0 or garbage now

// Assigning dynamic memory to the sequence
mymsg.values.capacity = 100;
mymsg.values.data = (int32_t*) malloc(mymsg.values.capacity * sizeof(int32_t));
mymsg.values.size = 0;

// Assigning static memory to the sequence
static int32_t memory[100];
mymsg.values.capacity = 100;
mymsg.values.data = memory;
mymsg.values.size = 0;

// Filling some data
for(int32_t i = 0; i < 3; i++){
    mymsg.values.data[i] = i;
    mymsg.values.size++;
}
```

**Compound types**

When dealing with a compound type, the user should recursively inspect the types in order to determine how to handle each internal member.

In the former `MyType.msg` example, the `header` member has the following structure:

```c
typedef struct std_msgs__msg__Header
{
    builtin_interfaces__msg__Time stamp;
    rosidl_runtime_c__String frame_id;
} std_msgs__msg__Header;
```

It is important to note that `rosidl_runtime_c__String` is equivalent to `rosidl_runtime_c__char__Sequence`. On its side, `builtin_interfaces__msg__Time` looks like:

```c
typedef struct builtin_interfaces__msg__Time
{
    int32_t sec;
    uint32_t nanosec;
} builtin_interfaces__msg__Time;
```
Given that, in order to initialize the header member of MyType.msg the following code is required:

```c
mypackage__msg__MyType mymsg;

// Assigning dynamic memory to the frame_id char sequence
mymsg.header.frame_id.capacity = 100;
mymsg.header.frame_id.data = (char*) malloc(mymsg.values.capacity * sizeof(char));
mymsg.header.frame_id.size = 0;

// Assigning value to the frame_id char sequence
strcpy(mymsg.header.frame_id.data, "Hello World");
mymsg.header.frame_id.size = strlen(mymsg.header.frame_id.data);

// Assigning value to other members
mymsg.stamp.sec = 10;
mymsg.stamp.nanosec = 20;
```

### Sequences of compound types

Users should take into account that sequence type member of compound type member are also valid ROS 2 type. For example, a complex .msg can be described as follows:

```c
# MyComplexType.msg
std_msgs/Header[] multiheaders
int32[] values
float64 duration
int8[10] coefficients
string name
```

In this case, the generated structure will be:

```c
typedef struct mypackage__msg__MyComplexType
{
    std_msgs__msg__Header__Sequence multiheaders;
    rosidl_runtime_c__int32__Sequence values;
    double duration;
    int8 coefficients[10];
    rosidl_runtime_c__String name; // equal to rosidl_runtime_c__char__Sequence
} mypackage__msg__MyComplexType;
```

In this case multiheaders is a **sequence type of compound type member**. It shall be handled correctly and recursively by the user, as in the following example:

```c
mypackage__msg__MyComplexType mymsg;

// Init the multiheaders sequence
mymsg.multiheaders.capacity = 10;
mymsg.multiheaders.data = (std_msgs__msg__Header*) malloc(mymsg.values.capacity * sizeof(std_msgs__msg__Header));
mymsg.multiheaders.size = 0;

// Filling some data
for(int32_t i = 0; i < 3; i++){
(continues on next page)
mymsg.values.data[i] = i;

// Add memory to this sequence element frame_id
mymsg.multiheaders.data[i].frame_id.capacity = 100;
mymsg.multiheaders.data[i].frame_id.data = (char*) malloc(mymsg.multiheaders.data[i].frame_id.capacity * sizeof(char));
mymsg.multiheaders.data[i].frame_id.size = 0;

// Assigning value to the frame_id char sequence
strcpy(mymsg.multiheaders.data[i].frame_id.data, "Hello World");
mymsg.multiheaders.data[i].frame_id.size = strlen(mymsg.multiheaders.data[i].frame_id.data) + 1;

// Assigning value to other members
mymsg.multiheaders.data[i].stamp.sec = 10;
mymsg.multiheaders.data[i].stamp.nanosec = 20;

mymsg.multiheaders.size++;}

4.13.6 micro-ROS and Discovery Server

- micro-ROS Agent configuration
- ROS 2 tools with Discover Server

micro-ROS can be use with Fast DDS Discovery Server as discovery protocol. Fast DDS Discovery Server can be useful for several reasons, some of the most important:

- An scenario where a considerable amount of participants and endpoints share the same domain.
- A network where multicast communication is not available.
- A network where multicast does not work properly, such as WiFi networks.
- DDS isolation needs to be ensured between different data spaces but DDS domains cannot be used.

Note: Further information about Discover Server setup.

Note: Further information about how to use Discover Server.
micro-ROS Agent configuration

micro-ROS Agent uses Fast DDS as middleware, so it can be configured using Fast DDS XML configuration files. Also, micro-ROS Agent can be configured using environment variables.

As any Fast DDS application, micro-ROS Agent can be configured in two ways:

- Environment variable.
- Configuration XML

ROS 2 tools with Discover Server

In order to be able to use ROS 2 tools to interface with micro-ROS entities created in a micro-ROS Agent with Discovery Server enabled an XML profile shall be loaded in the ROS environment.

The following example provides an example XML for configuring ROS 2 Daemon as a SuperClient:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<dds>
  <profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="super_client_profile" is_default_profile="true">
      <rtps>
        <builtin>
          <discovery_config>
            <discoveryProtocol>SUPER_CLIENT</discoveryProtocol>
            <discoveryServersList>
              <RemoteServer prefix="44.53.00.5f.45.50.52.4f.53.49.4d.41">
                <RemoteServer prefix="44.53.00.5f.45.50.52.4f.53.49.4d.41"/>
              </RemoteServer>
            </discoveryServersList>
          </discovery_config>
        </builtin>
      </rtps>
    </participant>
  </profiles>
</dds>
```

In order to use the previous XML file, the environment variable FASTRTPS_DEFAULT_PROFILES_FILE shall be set to the path of the XML file:

```bash
export FASTRTPS_DEFAULT_PROFILES_FILE=[PATH_TO_XML_FILE]
```

4.14 Vulcanexus Use Cases

4.14.1 Mapping Webots environment with ROSbot 2R teleoperation

- Background
- Prerequisites
- ROS 2 Packages
- Environment review
### Background

This document explains how to map a Webots environment with the teleoperation of a ROSbot 2R robot from Husarion. In this demo we will use the simulation of ROSbot 2R - an autonomous mobile robot by Husarion, designed for learning ROS and for research and development purposes. It is an affordable platform that can serve as a base for a variety of robotic applications, including inspection robots and custom service robots. The robot features a solid aluminum frame and is equipped with a Raspberry Pi 4 with 4GB of RAM, distance sensors, an RPLIDAR A2 laser scanner, and an RGB-D Orbbec Astra camera.

Webots is an open-source three-dimensional mobile robot simulator. It is the simulator selected for simulating the Husarion environment and the ROSbot. It is based in Qt, a physics engine (ODE fork) and an OpenGL 3.3 rendering engine. Please, refer to Webots for more information.

RViz2 is a 3D visualization tool developed for ROS 2. It allows the display of ROS 2 Topics and the extension of its capabilities through the development of plugins.

rosbot-mapping is the GitHub repository from Husarion where you can find the Docker compose and configuration used in this demo. All its contents are explained below.

### Prerequisites

This tutorial covers the first steps of setting up a ROSbot 2R simulation in Webots. For this task, we will use the Docker Compose from the repository mentioned above. Therefore, it is necessary to have Docker and Docker Compose installed. If you do not have them installed, please follow Install Docker on Ubuntu and Getting Docker Compose installation guides.

High-performance computing is needed for the simulation. It is necessary to process such a large amount of data and to perform complex calculations at high speed. This requires to use NVIDIA Container Runtime. Ensure that you have NVIDIA GPU and NVIDIA Container Toolkit installed, otherwise, follow the installation steps here.

Add the rosbot-mapping GitHub repository into your workspace directory with the following command:
git clone https://github.com/husarion/rosbot-mapping

ROS 2 Packages

The ROS 2 Packages involved in the demo are the following:

- **webots_ros2_husarion** It is a ROS 2 package prepared to start Webots with Husarion environment.
- **slam_toolbox** It is a ROS 2 package that provides a set of tools for 2D Simultaneous Localization and Mapping (SLAM).
- **nav2_map_server** It is a ROS 2 package from Navigation that provides maps built with the information of the sensors, in this case, the RPLIDAR A2.
- **teleop_twist_keyboard** It is a ROS 2 package that provides a node which takes keypresses from the keyboard and publishes them as Twist messages.

Environment review

The Docker Compose used for the simulation is `compose.sim.webots.yaml`. You can find it [here](https://github.com/husarion/rosbot-mapping).

The Docker Compose launches the following containers:

- **rviz**: is the container that is responsible for starting up RViz2 with the appropriate configurations.
- **rosbot**: start the environment and the robot in Webots.
- **mapping**: will be responsible for locating the robot and creating a map using the odometry and LIDAR data received from the sensors.
- **map-saver**: is responsible for the storage of the map that was previously created.

The following lines apply to all containers and mean that everything is running in host.

```yaml
x-net-config:
  &net-config
  network_mode: host
```

(continues on next page)
Below are the GPU configurations that apply to RViz2 and Webots, as they are graphical applications.

```yaml
x-gpu-config:
  &gpu-config
  runtime: nvidia
  environment:
    - DISPLAY=${DISPLAY:?err}
    - NVIDIA_VISIBLE_DEVICES=all
    - NVIDIA_DRIVER_CAPABILITIES=all
```

Then start the container definition.

The first thing you can see is the rviz container, which is started with the configuration file found here. The plugins used are defined in the configuration file. Among them, for example, we find the plugin from the navigation stack 2D Goal Pose, which allows the user to send a target by setting a desired pose for the robot to achieve. For the purposes of this tutorial, however, we need only pay attention to the Displays and familiarize ourselves with the information they provide. Between them you can see the display of the map created in static_map and the information from the LIDAR in scan.

```yaml
services:

  rviz:
    image: husarion/rviz2:vulcanexus-humble-11.2.5-20230308
    <<: [ *net-config, *gpu-config ]
    container_name: rviz
    volumes:
      - /tmp/.X11-unix:/tmp/.X11-unix:rw
      - ./config/rosbot.rviz:/root/.rviz2/default.rviz
```

The following service defines the rosbot container. It launches Webots with the robot and environment of choice. It also starts the nodes responsible for publishing the robot's state, as robot_localisation or robot_state_publisher.

```yaml
rosbot:
  image: husarion/webots:vulcanexus-humble-2023.0.1-20230301
  <<: [ *net-config, *gpu-config ]
  volumes:
    - /tmp/.X11-unix:/tmp/.X11-unix:rw
  command: ros2 launch webots_ros2_husarion robot_launch.py robot_name:=rosbot
```

Next, the mapping container will start up the slam_toolbox node with the configuration file, which can be found here. use_sim_time set to True define that must use Webots clock.

```yaml
mapping:
  image: husarion/slam-toolbox:vulcanexus-humble-2.6.4-20230228
  <<: *net-config
  volumes:
    - ./config/slam_toolbox_webots.yaml:/slam_params.yaml
  command: >
    ros2 launch slam_toolbox online_sync_launch.py
```

Finally, the map-server container is responsible for saving the map created by the previous container every 5 seconds. It will store the map in your ./maps directory.

```
map-saver:
  image: husarion/nav2-map-server:vulcanexus-humble-1.1.5-20230228
  <<: *net-config
  volumes:
    - ./maps:/maps
  command: bash -c "while true; do ros2 run nav2_map_server map_saver_cli --free 0.15 --fmt png -f /maps/map; sleep 5; done"
```

**Execution**

First, it is necessary to launch the docker compose `compose.sim.webots.yaml` that will activate the containers rviz, rosbot, mapping and map-saver.

Start the containers in a new terminal:

```
xhost +local:docker
docker compose -f compose.sim.webots.yaml up
```

In order to teleoperate the ROSbot with the keyboard, launch the teleoperation node, `teleop_twist_keyboard`, inside the rviz docker container previously created. Use the commands below in a second terminal:

```
docker exec -it rviz bash
ros2 run teleop_twist_keyboard teleop_twist_keyboard
```
Now you can move around the environment with the keyboard and create a map!

4.15 Supported platforms

Vulcanexus ROS 2 all-in-one tool set, is officially available in the platforms specified in the table below.

<table>
<thead>
<tr>
<th>Vulcanexus Version</th>
<th>Architecture</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Galactic Gamble v1</td>
<td>x86_64</td>
<td>Ubuntu Focal</td>
</tr>
<tr>
<td>Humble Hierro v2</td>
<td>x86_64 arm64</td>
<td>Ubuntu Jammy</td>
</tr>
<tr>
<td>Iron Imagination v3</td>
<td>x86_64 arm64</td>
<td>Ubuntu Jammy</td>
</tr>
</tbody>
</table>

However, as ROS 2 is officially supported in the platforms stated in the REP 2000 specification, building Vulcanexus for these platforms is expected to succeed. Other platforms not mentioned in the REP 2000 specification may also build successfully and be used.

4.16 Vulcanexus Releases

Vulcanexus maintains several releases with different support cycles. Each year, a new Vulcanexus major version is released. This major versions have a code name composed of an adjective and the name of a volcano, both starting with the same letter, the first of them being Galactic Gamble (v1.0.0). Within the support period of any version, there can be both minor and patch releases that either add new functionalities in an ABI compatible way, or fix possible issues. Every other year, a long term release (LTS) is released, the first of them being the H version (May 2022). In between, LTSs a short term release is released which will receive support for a shorter period of time. The following table outlines the Vulcanexus releases and their support cycles:
4.16.1 Iron Imagination

Iron Imagination (v3.0.0)

This version ships the following packages:

<table>
<thead>
<tr>
<th>Package name</th>
<th>Package version</th>
<th>Since Vulcanexus version</th>
</tr>
</thead>
<tbody>
<tr>
<td>eProsima Fast CDR</td>
<td>v1.1.0</td>
<td>v3.0.0</td>
</tr>
<tr>
<td>eProsima Fast DDS</td>
<td>v2.11.1</td>
<td>v3.0.0</td>
</tr>
<tr>
<td>eProsima Fast DDS Gen</td>
<td>v2.5.1</td>
<td>v3.0.0</td>
</tr>
<tr>
<td>foonathan_memory_vendor</td>
<td>v1.3.1</td>
<td>v3.0.0</td>
</tr>
<tr>
<td>micro-ROS Agent</td>
<td>v4.0.3</td>
<td>v3.0.0</td>
</tr>
<tr>
<td>micro-ROS Setup</td>
<td>v4.1.0</td>
<td>v3.0.0</td>
</tr>
<tr>
<td>ROS 2 Monitor</td>
<td>v1.3.0</td>
<td>v3.0.0</td>
</tr>
<tr>
<td>ROS 2 Router</td>
<td>v2.0.0</td>
<td>v3.0.0</td>
</tr>
<tr>
<td>ROS 2 Shapes Demo</td>
<td>v2.11.1</td>
<td>v3.0.0</td>
</tr>
<tr>
<td>ROS 2 Statistics Backend</td>
<td>v0.11.0</td>
<td>v3.0.0</td>
</tr>
<tr>
<td>Vulcanexus Fast DDS RMW</td>
<td>vulcanexus-iron</td>
<td>v3.0.0</td>
</tr>
<tr>
<td>eProsima Dev Utils</td>
<td>v0.4.0</td>
<td>v3.0.0</td>
</tr>
<tr>
<td>eProsima DDS Pipe</td>
<td>v0.2.0</td>
<td>v3.0.0</td>
</tr>
<tr>
<td>eProsima DDS Record &amp; Replay</td>
<td>v0.2.0</td>
<td>v3.0.0</td>
</tr>
<tr>
<td>eProsima Fast DDS Spy</td>
<td>v0.2.0</td>
<td>v3.0.0</td>
</tr>
<tr>
<td>eProsima Fast DDS QoS Profiles Manager</td>
<td>v0.1.0</td>
<td>v3.0.0</td>
</tr>
</tbody>
</table>
### 4.16.2 Humble Hierro

**Humble Hierro (v2.3.0)**

This version ships the following packages:

<table>
<thead>
<tr>
<th>Package name</th>
<th>Package version</th>
<th>Since Vulcanexus version</th>
</tr>
</thead>
<tbody>
<tr>
<td>eProsima Fast CDR</td>
<td>v1.1.0</td>
<td>v2.3.0</td>
</tr>
<tr>
<td>eProsima Fast DDS</td>
<td>v2.11.1</td>
<td>v2.3.0</td>
</tr>
<tr>
<td>eProsima Fast DDS Gen</td>
<td>v2.5.1</td>
<td>v2.3.0</td>
</tr>
<tr>
<td>foonathan_memory_vendor</td>
<td>v1.3.1</td>
<td>v2.3.0</td>
</tr>
<tr>
<td>micro-ROS Agent</td>
<td>v3.0.5</td>
<td>v2.3.0</td>
</tr>
<tr>
<td>micro-ROS Setup</td>
<td>v3.1.3</td>
<td>v2.3.0</td>
</tr>
<tr>
<td>ROS 2 Monitor</td>
<td>v1.5.0</td>
<td>v2.3.0</td>
</tr>
<tr>
<td>ROS 2 Router</td>
<td>v2.0.0</td>
<td>v2.3.0</td>
</tr>
<tr>
<td>ROS 2 Shapes Demo</td>
<td>v2.11.1</td>
<td>v2.3.0</td>
</tr>
<tr>
<td>ROS 2 Statistics Backend</td>
<td>v0.11.0</td>
<td>v2.3.0</td>
</tr>
<tr>
<td>Vulcanexus Fast DDS RMW</td>
<td>vulcanexus-humble</td>
<td>v2.0.3</td>
</tr>
<tr>
<td>eProsima Dev Utils</td>
<td>v0.4.0</td>
<td>v2.3.0</td>
</tr>
<tr>
<td>eProsima DDS Pipe</td>
<td>v0.2.0</td>
<td>v2.3.0</td>
</tr>
<tr>
<td>eProsima DDS Record &amp; Replay</td>
<td>v0.2.0</td>
<td>v2.3.0</td>
</tr>
<tr>
<td>eProsima Fast DDS Spy</td>
<td>v0.2.0</td>
<td>v2.3.0</td>
</tr>
<tr>
<td>eProsima Fast DDS QoS Profiles Manager</td>
<td>v0.1.0</td>
<td>v2.3.0</td>
</tr>
</tbody>
</table>

**Humble Hierro previous versions**

**Humble Hierro (v2.2.0)**

This version ships the following packages:

<table>
<thead>
<tr>
<th>Package name</th>
<th>Package version</th>
<th>Since Vulcanexus version</th>
</tr>
</thead>
<tbody>
<tr>
<td>eProsima Fast CDR</td>
<td>v1.0.27</td>
<td>v2.2.0</td>
</tr>
<tr>
<td>eProsima Fast DDS</td>
<td>v2.10.1</td>
<td>v2.2.0</td>
</tr>
<tr>
<td>eProsima Fast DDS Gen</td>
<td>v2.4.0</td>
<td>v2.2.0</td>
</tr>
<tr>
<td>foonathan_memory_vendor</td>
<td>v1.3.0</td>
<td>v2.2.0</td>
</tr>
<tr>
<td>micro-ROS Agent</td>
<td>v3.0.4</td>
<td>v2.0.4</td>
</tr>
<tr>
<td>micro-ROS Setup</td>
<td>v3.1.2</td>
<td>v2.0.4</td>
</tr>
<tr>
<td>ROS 2 Monitor</td>
<td>v1.4.0</td>
<td>v2.2.0</td>
</tr>
<tr>
<td>ROS 2 Router</td>
<td>v1.2.0</td>
<td>v2.2.0</td>
</tr>
<tr>
<td>ROS 2 Shapes Demo</td>
<td>v2.10.1</td>
<td>v2.2.0</td>
</tr>
<tr>
<td>ROS 2 Statistics Backend</td>
<td>v0.10.0</td>
<td>v2.2.0</td>
</tr>
<tr>
<td>Vulcanexus Fast DDS RMW</td>
<td>vulcanexus-humble</td>
<td>v2.0.3</td>
</tr>
<tr>
<td>eProsima Dev Utils</td>
<td>v0.3.0</td>
<td>v2.2.0</td>
</tr>
</tbody>
</table>
Humble Hierro (v2.1.1)

This version ships the following packages:

<table>
<thead>
<tr>
<th>Package name</th>
<th>Package version</th>
<th>Since Vulcanexus version</th>
</tr>
</thead>
<tbody>
<tr>
<td>eProsima Fast CDR</td>
<td>v1.0.26</td>
<td>v2.1.0</td>
</tr>
<tr>
<td>eProsima Fast DDS</td>
<td>v2.9.1</td>
<td>v2.1.0</td>
</tr>
<tr>
<td>eProsima Fast DDS Gen</td>
<td>v2.3.0</td>
<td>v2.1.0</td>
</tr>
<tr>
<td>foonathan_memory_vendor</td>
<td>v1.2.2</td>
<td>v2.1.0</td>
</tr>
<tr>
<td>micro-ROS Agent</td>
<td>v3.0.4</td>
<td>v2.0.4</td>
</tr>
<tr>
<td>micro-ROS Setup</td>
<td>v3.1.2</td>
<td>v2.0.4</td>
</tr>
<tr>
<td>ROS 2 Monitor</td>
<td>v1.3.0</td>
<td>v2.1.0</td>
</tr>
<tr>
<td>ROS 2 Router</td>
<td>v1.1.0</td>
<td>v2.1.0</td>
</tr>
<tr>
<td>ROS 2 Shapes Demo</td>
<td>v2.9.1</td>
<td>v2.1.0</td>
</tr>
<tr>
<td>ROS 2 Statistics Backend</td>
<td>v0.8.0</td>
<td>v2.0.5</td>
</tr>
<tr>
<td>Vulcanexus Fast DDS RMW</td>
<td>vulcanexus-humble</td>
<td>v2.0.3</td>
</tr>
<tr>
<td>eProsima Dev Utils</td>
<td>v0.2.0</td>
<td>v2.1.0</td>
</tr>
</tbody>
</table>

Humble Hierro (v2.1.0)

This version ships the following packages:

<table>
<thead>
<tr>
<th>Package name</th>
<th>Package version</th>
<th>Since Vulcanexus version</th>
</tr>
</thead>
<tbody>
<tr>
<td>eProsima Fast CDR</td>
<td>v1.0.26</td>
<td>v2.1.0</td>
</tr>
<tr>
<td>eProsima Fast DDS</td>
<td>v2.9.0</td>
<td>v2.1.0</td>
</tr>
<tr>
<td>eProsima Fast DDS Gen</td>
<td>v2.3.0</td>
<td>v2.1.0</td>
</tr>
<tr>
<td>foonathan_memory_vendor</td>
<td>v1.2.2</td>
<td>v2.1.0</td>
</tr>
<tr>
<td>micro-ROS Agent</td>
<td>v3.0.4</td>
<td>v2.0.4</td>
</tr>
<tr>
<td>micro-ROS Setup</td>
<td>v3.1.2</td>
<td>v2.0.4</td>
</tr>
<tr>
<td>ROS 2 Monitor</td>
<td>v1.3.0</td>
<td>v2.1.0</td>
</tr>
<tr>
<td>ROS 2 Router</td>
<td>v1.1.0</td>
<td>v2.1.0</td>
</tr>
<tr>
<td>ROS 2 Shapes Demo</td>
<td>v2.9.1</td>
<td>v2.1.0</td>
</tr>
<tr>
<td>ROS 2 Statistics Backend</td>
<td>v0.8.0</td>
<td>v2.0.5</td>
</tr>
<tr>
<td>Vulcanexus Fast DDS RMW</td>
<td>vulcanexus-humble</td>
<td>v2.0.3</td>
</tr>
<tr>
<td>eProsima Dev Utils</td>
<td>v0.2.0</td>
<td>v2.1.0</td>
</tr>
</tbody>
</table>
**Humble Hierro (v2.0.5)**

This version ships the following packages:

<table>
<thead>
<tr>
<th>Package name</th>
<th>Package version</th>
<th>Since Vulcanexus version</th>
</tr>
</thead>
<tbody>
<tr>
<td>eProsima Fast CDR</td>
<td>v1.0.25</td>
<td>v2.0.4</td>
</tr>
<tr>
<td>eProsima Fast DDS</td>
<td>v2.8.1</td>
<td>v2.0.5</td>
</tr>
<tr>
<td>eProsima Fast DDS Gen</td>
<td>v2.2.0</td>
<td>v2.0.4</td>
</tr>
<tr>
<td>foonathan_memory_vendor</td>
<td>v1.2.1</td>
<td>v2.0.0</td>
</tr>
<tr>
<td>micro-ROS Agent</td>
<td>v3.0.4</td>
<td>v2.0.4</td>
</tr>
<tr>
<td>micro-ROS Setup</td>
<td>v3.1.2</td>
<td>v2.0.4</td>
</tr>
<tr>
<td>ROS 2 Monitor</td>
<td>v1.2.1</td>
<td>v2.0.4</td>
</tr>
<tr>
<td>ROS 2 Router</td>
<td>v1.0.0</td>
<td>v2.0.4</td>
</tr>
<tr>
<td>ROS 2 Shapes Demo</td>
<td>v2.8.1</td>
<td>v2.0.5</td>
</tr>
<tr>
<td>ROS 2 Statistics Backend</td>
<td>v0.8.0</td>
<td>v2.0.5</td>
</tr>
<tr>
<td>Vulcanexus Fast DDS RMW</td>
<td>vulcanexus-humble</td>
<td>v2.0.3</td>
</tr>
<tr>
<td>eProsima Dev Utils</td>
<td>v0.1.0</td>
<td>v2.0.4</td>
</tr>
</tbody>
</table>

**Humble Hierro (v2.0.4)**

This version ships the following packages:

<table>
<thead>
<tr>
<th>Package name</th>
<th>Package version</th>
<th>Since Vulcanexus version</th>
</tr>
</thead>
<tbody>
<tr>
<td>eProsima Fast CDR</td>
<td>v1.0.25</td>
<td>v2.0.4</td>
</tr>
<tr>
<td>eProsima Fast DDS</td>
<td>v2.8.0</td>
<td>v2.0.4</td>
</tr>
<tr>
<td>eProsima Fast DDS Gen</td>
<td>v2.2.0</td>
<td>v2.0.4</td>
</tr>
<tr>
<td>foonathan_memory_vendor</td>
<td>v1.2.1</td>
<td>v2.0.0</td>
</tr>
<tr>
<td>micro-ROS Agent</td>
<td>v3.0.4</td>
<td>v2.0.4</td>
</tr>
<tr>
<td>micro-ROS Setup</td>
<td>v3.1.2</td>
<td>v2.0.4</td>
</tr>
<tr>
<td>ROS 2 Monitor</td>
<td>v1.2.1</td>
<td>v2.0.4</td>
</tr>
<tr>
<td>ROS 2 Router</td>
<td>v1.0.0</td>
<td>v2.0.4</td>
</tr>
<tr>
<td>ROS 2 Shapes Demo</td>
<td>v2.8.0</td>
<td>v2.0.4</td>
</tr>
<tr>
<td>ROS 2 Statistics Backend</td>
<td>v0.7.1</td>
<td>v2.0.4</td>
</tr>
<tr>
<td>Vulcanexus Fast DDS RMW</td>
<td>vulcanexus-humble</td>
<td>v2.0.3</td>
</tr>
<tr>
<td>eProsima Dev Utils</td>
<td>v0.1.0</td>
<td>v2.0.4</td>
</tr>
</tbody>
</table>
Humble Hierro (v2.0.3)

This version ships the following packages:

<table>
<thead>
<tr>
<th>Package name</th>
<th>Package version</th>
<th>Since Vulcanexus version</th>
</tr>
</thead>
<tbody>
<tr>
<td>eProsima Fast CDR</td>
<td>v1.0.24</td>
<td>v2.0.0</td>
</tr>
<tr>
<td>eProsima Fast DDS</td>
<td>v2.7.1</td>
<td>v2.0.3</td>
</tr>
<tr>
<td>eProsima Fast DDS Gen</td>
<td>v2.1.3</td>
<td>v2.0.2</td>
</tr>
<tr>
<td>foonathan_memory_vendor</td>
<td>v1.2.1</td>
<td>v2.0.0</td>
</tr>
<tr>
<td>micro-ROS Agent</td>
<td>v3.0.3</td>
<td>v2.0.1</td>
</tr>
<tr>
<td>micro-ROS Setup</td>
<td>v3.1.0</td>
<td>v2.0.0</td>
</tr>
<tr>
<td>ROS 2 Monitor</td>
<td>v1.2.0</td>
<td>v2.0.2</td>
</tr>
<tr>
<td>ROS 2 Router</td>
<td>v0.4.0</td>
<td>v2.0.2</td>
</tr>
<tr>
<td>ROS 2 Shapes Demo</td>
<td>v2.7.1</td>
<td>v2.0.3</td>
</tr>
<tr>
<td>ROS 2 Statistics Backend</td>
<td>v0.7.0</td>
<td>v2.0.2</td>
</tr>
<tr>
<td>Vulcanexus Fast DDS RMW</td>
<td>Use Fast DDS WaitSets</td>
<td>v2.0.3</td>
</tr>
</tbody>
</table>

Humble Hierro (v2.0.2)

This version ships the following packages:

<table>
<thead>
<tr>
<th>Package name</th>
<th>Package version</th>
<th>Since Vulcanexus version</th>
</tr>
</thead>
<tbody>
<tr>
<td>eProsima Fast CDR</td>
<td>v1.0.24</td>
<td>v2.0.0</td>
</tr>
<tr>
<td>eProsima Fast DDS</td>
<td>v2.7.0</td>
<td>v2.0.2</td>
</tr>
<tr>
<td>eProsima Fast DDS Gen</td>
<td>v2.1.3</td>
<td>v2.0.2</td>
</tr>
<tr>
<td>foonathan_memory_vendor</td>
<td>v1.2.1</td>
<td>v2.0.0</td>
</tr>
<tr>
<td>micro-ROS Agent</td>
<td>v3.0.3</td>
<td>v2.0.1</td>
</tr>
<tr>
<td>micro-ROS Setup</td>
<td>v3.1.0</td>
<td>v2.0.0</td>
</tr>
<tr>
<td>ROS 2 Monitor</td>
<td>v1.2.0</td>
<td>v2.0.2</td>
</tr>
<tr>
<td>ROS 2 Router</td>
<td>v0.4.0</td>
<td>v2.0.2</td>
</tr>
<tr>
<td>ROS 2 Shapes Demo</td>
<td>v2.7.0</td>
<td>v2.0.2</td>
</tr>
<tr>
<td>ROS 2 Statistics Backend</td>
<td>v0.7.0</td>
<td>v2.0.2</td>
</tr>
<tr>
<td>Vulcanexus Fast DDS RMW</td>
<td>v6.2.1 improved with PKCS #11 support.</td>
<td>v2.0.0</td>
</tr>
</tbody>
</table>
Humble Hierro (v2.0.1)

This version ships the following packages:

<table>
<thead>
<tr>
<th>Package name</th>
<th>Package version</th>
<th>Since Vulcanexus version</th>
</tr>
</thead>
<tbody>
<tr>
<td>eProsima Fast CDR</td>
<td>v1.0.24</td>
<td>v2.0.0</td>
</tr>
<tr>
<td>eProsima Fast DDS</td>
<td>v2.6.1</td>
<td>v2.0.1</td>
</tr>
<tr>
<td>eProsima Fast DDS Gen</td>
<td>v2.1.2</td>
<td>v2.0.0</td>
</tr>
<tr>
<td>foonathan_memory_vendor</td>
<td>v1.2.1</td>
<td>v2.0.0</td>
</tr>
<tr>
<td>micro-ROS Agent</td>
<td>v3.0.3</td>
<td>v2.0.1</td>
</tr>
<tr>
<td>micro-ROS Setup</td>
<td>v3.1.0</td>
<td>v2.0.0</td>
</tr>
<tr>
<td>ROS 2 Monitor</td>
<td>v1.1.0</td>
<td>v2.0.0</td>
</tr>
<tr>
<td>ROS 2 Router</td>
<td>v0.3.0</td>
<td>v2.0.0</td>
</tr>
<tr>
<td>ROS 2 Shapes Demo</td>
<td>v2.6.1</td>
<td>v2.0.0</td>
</tr>
<tr>
<td>ROS 2 Statistics Backend</td>
<td>v0.6.0</td>
<td>v2.0.0</td>
</tr>
<tr>
<td>Vulcanexus Fast DDS RMW</td>
<td>v6.2.1 improved with PKCS #11 support</td>
<td>v2.0.0</td>
</tr>
</tbody>
</table>

This release also adds the following *binary package distributions*:

- Vulcanexx Humble Simulation: vulcanexx-humble-simulation

Humble Hierro (v2.0.0)

Initial release extending ROS 2 Humble Hawksbill including the following package versions:

<table>
<thead>
<tr>
<th>Package name</th>
<th>Package version</th>
<th>Since Vulcanexus version</th>
</tr>
</thead>
<tbody>
<tr>
<td>eProsima Fast CDR</td>
<td>v1.0.24</td>
<td>v2.0.0</td>
</tr>
<tr>
<td>eProsima Fast DDS</td>
<td>v2.6.0</td>
<td>v2.0.0</td>
</tr>
<tr>
<td>eProsima Fast DDS Gen</td>
<td>v2.1.2</td>
<td>v2.0.0</td>
</tr>
<tr>
<td>foonathan_memory_vendor</td>
<td>v1.2.1</td>
<td>v2.0.0</td>
</tr>
<tr>
<td>micro-ROS Agent</td>
<td>v3.0.2</td>
<td>v2.0.0</td>
</tr>
<tr>
<td>micro-ROS Setup</td>
<td>v3.1.0</td>
<td>v2.0.0</td>
</tr>
<tr>
<td>ROS 2 Monitor</td>
<td>v1.1.0</td>
<td>v2.0.0</td>
</tr>
<tr>
<td>ROS 2 Router</td>
<td>v0.3.0</td>
<td>v2.0.0</td>
</tr>
<tr>
<td>ROS 2 Shapes Demo</td>
<td>v2.6.1</td>
<td>v2.0.0</td>
</tr>
<tr>
<td>ROS 2 Statistics Backend</td>
<td>v0.6.0</td>
<td>v2.0.0</td>
</tr>
<tr>
<td>Vulcanexus Fast DDS RMW</td>
<td>v6.2.1 improved with PKCS #11 support</td>
<td>v2.0.0</td>
</tr>
</tbody>
</table>

This release also includes the following *binary package distributions*:

- Vulcanexx Humble Base: vulcanexx-humble-base
- Vulcanexx Humble Cloud: vulcanexx-humble-cloud
- Vulcanexx Humble Core: vulcanexx-humble-core
- Vulcanexx Humble Desktop: vulcanexx-humble-desktop
- Vulcanexx Humble Micro: vulcanexx-humble-micro
- Vulcanexx Humble Tools: vulcanexx-humble-tools
4.16.3 Galactic Gamble

Galactic Gamble (v1.0.3)

This version ships the following packages:

<table>
<thead>
<tr>
<th>Package name</th>
<th>Package version</th>
<th>Since Vulcanexus version</th>
</tr>
</thead>
<tbody>
<tr>
<td>eProsima Fast CDR</td>
<td>v1.0.25</td>
<td>v1.0.3</td>
</tr>
<tr>
<td>eProsima Fast DDS</td>
<td>v2.8.0</td>
<td>v1.0.3</td>
</tr>
<tr>
<td>eProsima Fast DDS Gen</td>
<td>v2.2.0</td>
<td>v1.0.3</td>
</tr>
<tr>
<td>foonathan_memory_vendor</td>
<td>v1.2.1</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>micro-ROS Agent</td>
<td>v2.0.4</td>
<td>v1.0.3</td>
</tr>
<tr>
<td>micro-ROS Setup</td>
<td>v2.1.1</td>
<td>v1.0.3</td>
</tr>
<tr>
<td>ROS 2 Monitor</td>
<td>v1.2.1</td>
<td>v1.0.3</td>
</tr>
<tr>
<td>ROS 2 Router</td>
<td>v1.0.0</td>
<td>v1.0.3</td>
</tr>
<tr>
<td>ROS 2 Shapes Demo</td>
<td>v2.8.0</td>
<td>v1.0.3</td>
</tr>
<tr>
<td>ROS 2 Statistics Backend</td>
<td>v0.7.1</td>
<td>v1.0.3</td>
</tr>
<tr>
<td>Vulcanexus Fast DDS RMW</td>
<td>vulcanexus-galactic</td>
<td>v1.0.3</td>
</tr>
<tr>
<td>eProsima Dev Utils</td>
<td>v0.1.0</td>
<td>v1.0.3</td>
</tr>
</tbody>
</table>

Galactic Gamble previous versions

Galactic Gamble (v1.0.2)

This version ships the following packages:

<table>
<thead>
<tr>
<th>Package name</th>
<th>Package version</th>
<th>Since Vulcanexus version</th>
</tr>
</thead>
<tbody>
<tr>
<td>eProsima Fast CDR</td>
<td>v1.0.24</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>eProsima Fast DDS</td>
<td>v2.7.0</td>
<td>v1.0.2</td>
</tr>
<tr>
<td>eProsima Fast DDS Gen</td>
<td>v2.1.3</td>
<td>v1.0.2</td>
</tr>
<tr>
<td>foonathan_memory_vendor</td>
<td>v1.2.1</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>micro-ROS Agent</td>
<td>v2.0.3</td>
<td>v1.0.1</td>
</tr>
<tr>
<td>micro-ROS Setup</td>
<td>v2.1.0</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>ROS 2 Monitor</td>
<td>v1.2.0</td>
<td>v1.0.2</td>
</tr>
<tr>
<td>ROS 2 Router</td>
<td>v0.4.0</td>
<td>v1.0.2</td>
</tr>
<tr>
<td>ROS 2 Shapes Demo</td>
<td>v2.7.0</td>
<td>v1.0.2</td>
</tr>
<tr>
<td>ROS 2 Statistics Backend</td>
<td>v0.7.0</td>
<td>v1.0.2</td>
</tr>
<tr>
<td>Vulcanexus Fast DDS RMW</td>
<td>v5.0.2 improved with PKCS #11 support.</td>
<td>v1.0.0</td>
</tr>
</tbody>
</table>

Galactic Gamble (v1.0.1)

This version ships the following packages:
This release also adds the following *binary package distributions*:

- Vulcanexus Galactic Simulation: vulcanexus-galactic-simulation

**Galactic Gamble (v1.0.0)**

Initial release extending ROS 2 Galactic Geochelone including the following package versions:

<table>
<thead>
<tr>
<th>Package name</th>
<th>Package version</th>
<th>Since Vulcanexus version</th>
</tr>
</thead>
<tbody>
<tr>
<td>eProsima Fast CDR</td>
<td>v1.0.24</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>eProsima Fast DDS</td>
<td>v2.6.1</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>eProsima Fast DDS Gen</td>
<td>v2.1.2</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>loonathan_memory_vendor</td>
<td>v1.2.1</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>micro-ROS Agent</td>
<td>v2.0.3</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>micro-ROS Setup</td>
<td>v2.1.0</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>ROS 2 Monitor</td>
<td>v1.1.0</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>ROS 2 Router</td>
<td>v0.3.0</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>ROS 2 Shapes Demo</td>
<td>v2.6.1</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>ROS 2 Statistics Backend</td>
<td>v0.6.0</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>Vulcanexus Fast DDS RMW</td>
<td>v5.0.2 improved with PKCS #11 support.</td>
<td>v1.0.0</td>
</tr>
</tbody>
</table>

This release also includes the following *binary package distributions*:

- Vulcanexus Galactic Base: vulcanexus-galactic-base
- Vulcanexus Galactic Cloud: vulcanexus-galactic-cloud
- Vulcanexus Galactic Core: vulcanexus-galactic-core
- Vulcanexus Galactic Desktop: vulcanexus-galactic-desktop
- Vulcanexus Galactic Micro: vulcanexus-galactic-micro
- Vulcanexus Galactic Tools: vulcanexus-galactic-tools

Table 7: Vulcanexus versions

<table>
<thead>
<tr>
<th>Name</th>
<th>Latest Version</th>
<th>Release Date</th>
<th>EOL Date</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Iron Imagination</em></td>
<td>v2.3.0</td>
<td>May 2022</td>
<td>May 2027</td>
</tr>
<tr>
<td><em>Humble Hierro</em></td>
<td>v2.3.0</td>
<td>May 2022</td>
<td>May 2027</td>
</tr>
<tr>
<td><em>Galactic Gamble</em></td>
<td>v1.0.4</td>
<td>May 2022</td>
<td>November 2022</td>
</tr>
</tbody>
</table>
4.17 Glossary

CA  Certificate Authority

**Diffie-Hellman  Diffie-Hellman key exchange** Diffie-Hellman key exchange is a method of securely exchanging cryptographic keys over a public channel and was one of the first public-key protocols.

LAN  Local Area Network

RSA  Rivest-Shamir-Adleman cryptosystem

**Public-key cryptography** **Public-key cryptography** Public-key cryptography, or asymmetric cryptography, is a cryptographic system that uses pairs of keys. Each pair consists of a public key (which may be known to others) and a private key (which may not be known by anyone except the owner).

TCP  Transmission Control Protocol

TLS  Transport Layer Security Transport Layer Security is a cryptographic protocol designed to provide communications security over a computer network.

WAN  Wide Area Network
INDEX

A
ament CMake package, 1302
ament package, 1302
ament Python package, 1302
API, 1402

C
CA, 1703
client_library, 1402
CMake package, 1302

D
Diffie-Hellman, 1703

L
LAN, 1703

P
package, 1402
package.xml, 1302
Public-key cryptography, 1703
Python package, 1302

R
rclcpp, 1402
REP, 1402
repository, 1402
RSA, 1703

T
TCP, 1703
TLS, 1703

V
VCS, 1402

W
WAN, 1703