



INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES
DE MONTERREY
CAMPUS GUADALAJARA

Curso de ROS

Apuntes de clase

Impartido por:

Dr. Eduardo de Jesús Dávila Meza

para el bloque integrador de:

Robótica y Sistemas Inteligentes

para estudiantes de:

Ingeniería en Robótica y Sistemas Digitales

de la Escuela de:

Ingeniería y Ciencias

Tecnológico de Monterrey, Campus Guadalajara. Zapopan, Jalisco. Marzo 2025.

Contents

Contents

iii

1	ROS 2 Environment Setup and Development Tools	1
1.1	Introduction to ROS 2	1
1.1.1	Basic Concepts	1
1.1.2	Some ROS Distributions	2
1.1.3	ROS Distribution for the Course	2
1.2	Installation of Ubuntu 22 and ROS 2 Humble Hawksbill	3
1.2.1	Installing Ubuntu 22.04.5 LTS (Jammy Jellyfish)	3
1.2.2	Installing ROS 2 Humble Hawksbill	3
1.3	Try Some Examples	5
1.3.1	Talker-Listener	5
1.3.2	Turtlesim: Publish and Move a Turtle	5
1.4	Configuring the Development Environment	6
1.4.1	Installing Visual Studio Code	6
1.4.2	Installing Recommended VS Code Extensions	6
1.4.3	Installing Terminator for Multiple Terminals	7
1.5	colcon Configuration	7
1.5.1	Installing colcon	7
1.5.2	Enabling colcon Argument Completion	7
1.6	Practice Assignment: Running ROS 2 Examples	8
1.6.1	Examples to Try	8
1.6.2	Submission Instructions	8
2	ROS 2 Workspace, Package, and Node Development	11
2.1	Fundamental Concepts of ROS 2	11
2.1.1	Workspace	11
2.1.2	Packages	11
2.1.3	Nodes	12
2.1.4	Topics	12
2.2	Workspace Creation and Setup	13
2.2.1	Creating the Workspace	13
2.2.2	Building the Workspace	14
2.2.3	Sourcing the Workspace Environment	14
2.3	Creating a ROS 2 Package for C++ Nodes	14
2.3.1	Navigating to the src Directory	14
2.3.2	Creating the C++ Package	14

2.3.3	Building the Package	15
2.3.4	Examining the Package Contents	15
2.3.5	Detailed Examination of Key Package Files	15
2.3.6	Ensuring Consistency Between <code>package.xml</code> and <code>CMakeLists.txt</code>	16
2.4	Creating a ROS 2 Package for Python Nodes	16
2.4.1	Navigating to the <code>src</code> Directory	16
2.4.2	Creating the Python Package	16
2.4.3	Building the Package	17
2.4.4	Examining the Package Contents	17
2.4.5	Detailed Examination of Key Package Files	17
2.4.6	Ensuring Consistency Between <code>package.xml</code> and <code>setup.py</code>	18
2.5	Creating a ROS 2 Publisher Node with C++	18
2.5.1	Setting Up the C++ Node	18
2.5.2	Editing the C++ Node	19
2.5.3	Integrating the Publisher Node into the Package	22
2.5.4	Running the Publisher Node as a Standalone Executable (Optional)	23
2.5.5	Running the ROS 2 Publisher Node with ROS 2	24
2.5.6	Summary of Key Identifiers	24
2.6	Creating a ROS 2 Subscriber Node with C++	24
2.6.1	Setting Up the C++ Node	24
2.6.2	Editing the C++ Node	25
2.6.3	Integrating the Subscriber Node into the Package	27
2.6.4	Running the Subscriber Node as a Standalone Executable (Optional)	28
2.6.5	Running the ROS 2 Subscriber Node with ROS 2	29
2.6.6	Summary of Key Identifiers	29
2.7	Creating a ROS 2 Publisher Node with Python	29
2.7.1	Setting Up the Python Node	29
2.7.2	Editing the Python Node	30
2.7.3	Running the Publisher Node Standalone (Optional)	33
2.7.4	Integrating the Publisher Node into the Package	33
2.7.5	Running the ROS 2 Publisher Node with ROS 2	34
2.7.6	Summary of Key Identifiers	34
2.8	Creating a ROS 2 Subscriber Node with Python	34
2.8.1	Setting Up the Python Node	34
2.8.2	Editing the Python Node	35
2.8.3	Breaking Down the Subscriber Node Code	36
2.8.4	Running the Subscriber Node Standalone (Optional)	37
2.8.5	Integrating the Subscriber Node into the Package	38
2.8.6	Running the ROS 2 Subscriber Node with ROS 2	39
2.8.7	Summary of Key Identifiers	39
2.9	Practice Assignment: Running ROS 2 Nodes	39
2.9.1	Executing ROS 2 Publishers and Subscribers	40
2.9.2	Submission Instructions	41



3	ROS 2 Service and Client	43
3.1	Introduction to ROS 2 Services and Clients	43
3.1.1	Concepts: Service and Client	43
3.1.2	Prerequisites	43
3.2	Creating a ROS 2 Service Node in C++	44
3.2.1	Setting Up the C++ Service Node	44
3.2.2	Editing the C++ Service Node	44
3.2.3	Integrating the C++ Service Node into the Package	47
3.2.4	Building, Sourcing, and Running the C++ Service Node	47
3.2.5	Testing the C++ Service	48
3.2.6	Summary of C++ Service Node Key Identifiers	48
3.3	Creating a ROS 2 Client Node in C++	48
3.3.1	Setting Up the C++ Client Node	48
3.3.2	Editing the C++ Client Node	48
3.3.3	Integrating the C++ Client Node into the Package	53
3.3.4	Building, Sourcing, and Running the C++ Client Node	54
3.3.5	Summary of C++ Client Node Key Identifiers	54
3.4	Creating a ROS 2 Service Node in Python	54
3.4.1	Setting Up the Python Service Node	54
3.4.2	Editing the Python Service Node	54
3.4.3	Integrating the Python Service Node into the Package	57
3.4.4	Building, Sourcing, and Running the Python Service Node	57
3.4.5	Testing the Python Service	57
3.4.6	Summary of Python Service Node Key Identifiers	57
3.5	Creating a ROS 2 Client Node in Python	58
3.5.1	Setting Up the Python Client Node	58
3.5.2	Editing the Python Client Node	58
3.5.3	Integrating the Python Client Node into the Package	61
3.5.4	Building, Sourcing, and Running the Python Client Node	62
3.5.5	Summary of Python Client Node Key Identifiers	62
3.6	Practice Assignment: Running ROS 2 Services and Clients	62
3.6.1	Executing ROS 2 Services and Clients	62
3.6.2	Submission Instructions	64
4	ROS 2 Custom msg and srv Files	65
4.1	Introduction to ROS 2 Services and Clients	65
4.1.1	Prerequisites	65
4.2	Creating the .msg and .srv files	66
4.2.1	msg definition	66
4.2.2	srv definition	66
4.3	CMakeLists.txt	67
4.4	package.xml	67
4.5	Build the s4_custom_interface package	68
4.6	Confirm msg and srv creation	68
4.7	Test the new interfaces	69



4.7.1	Implementing <code>HardwareStatus.msg</code> in <code>Sphere.msg</code> with pub/sub nodes	69
4.7.2	Implementing <code>AddThreeInts.srv</code> in service/client nodes	69
4.7.3	<code>CMakeLists.txt</code>	69
4.7.4	<code>setup.py</code>	70
4.7.5	<code>package.xml</code>	70
4.7.6	Building the Packages and Running the Nodes	70
4.8	Practice Assignment: Custom msg and srv Files	71
4.8.1	Executing ROS 2 Publishers, Subscribers, Services and Clients	72
4.8.2	Visualizing the ROS Graph with <code>rqt_graph</code>	73
4.8.3	Submission Instructions	73
5	Parameters and YAML and Launch files (Image acquisition)	75
A	Essential ROS 2 Command Reference	77
A.1	System Package Management and Updates in Ubuntu	77
A.2	ROS 2 Environment Setup	78
A.3	Workspace Creation and Building	78
A.4	Package Creation and Editing in C++ and Python	79
A.5	Dev & Debug Essentials: CLI/GUI Commands	80
B	Using the <code>geometry_msgs</code> Package in C++ and Python	81
B.1	Importing the Package	81
B.1.1	In C++	81
B.1.2	In Python	81
B.1.3	Including <code>geometry_msgs</code> in Your Package Dependencies	82
B.2	Using Message Definitions	82
B.2.1	Point	82
B.2.2	Quaternion	83
B.2.3	Pose	83
B.3	Additional Message Types	84

CHAPTER

1

ROS 2 Environment Setup and Development Tools

Author: Dr. Eduardo de Jesús Dávila Meza.

 [EduardoDavila-AI-PhD](#)

1.1 Introduction to ROS 2

1.1.1 Basic Concepts

The [Robot Operating System \(ROS\)](#) is not an actual operating system but a flexible *framework* for writing robot software. It provides a collection of software libraries, tools, and conventions to simplify the task of creating complex and robust 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. Some of its main advantages include:

- **Multi-language:** While ROS supports multiple programming languages, the primary supported languages are C++ and Python. Support for other languages like Java exists but is less common.
- **Free and open-source:** ROS 1 and ROS 2 are available on Ubuntu, with ROS 2 also supporting Windows and macOS. However, the availability and stability can vary across different versions and platforms.
- **Support:** Since ROS was started in 2007, a lot has changed in the robotics and ROS community. The ROS 2 project began in 2015 to address the evolving needs of the robotics community, building upon the strengths of ROS 1 and introducing improvements for better performance, security, and support for real-time systems.

1.1.2 Some ROS Distributions

ROS is released as distributions (or "distros"), with several versions supported concurrently. Some releases come with long-term support (LTS), meaning they are more stable and have undergone extensive testing, while other distributions are newer, have shorter lifetimes, and support more recent platforms and package versions. Generally, a new ROS distro is released every year on [World Turtle Day](#), with LTS releases appearing in even-numbered years. ROS is available for different versions of Ubuntu and other operating systems. Some of the notable distributions include:

- **Indigo Igloo (ROS 1):** Ubuntu 14.04 (Trusty Tahr)
- **Kinetic Kame (ROS 1):** Ubuntu 16.04 (Xenial Xerus)
- **Melodic Morenia (ROS 1):** Ubuntu 18.04 (Bionic Beaver)
- **Noetic Ninjemys (ROS 1):** Ubuntu 20.04 (Focal Fossa)
- **Foxy Fitzroy (ROS 2):** Ubuntu 20.04 (Focal Fossa)
- **Humble Hawksbill (ROS 2):** Ubuntu 22.04 (Jammy Jellyfish)
- **Jazzy Jalisco (ROS 2):** Ubuntu 24.04 (Noble Numbat)

For more details, see the [ROS Distributions list](#). Currently, the Noetic Ninjemys, Humble Hawksbill, and Jazzy Jalisco distributions are actively supported.

1.1.3 ROS Distribution for the Course

For this course, we will use **ROS 2 Humble Hawksbill**, a long-term support (LTS) release that offers enhanced performance, security, and real-time capabilities. Its logo, shown in Figure 1.1, reflects its distinctive branding. ROS 2 Humble Hawksbill is compatible with Ubuntu 22.04 (Jammy Jellyfish) and Windows 10, making it a versatile choice for both simulation and deployment on physical hardware. This distribution will serve as the foundation for all class exercises and, in particular, for project development.



Figure 1.1: ROS 2 Humble Hawksbill Logo.

1.2 Installation of Ubuntu 22 and ROS 2 Humble Hawksbill

1.2.1 Installing Ubuntu 22.04.5 LTS (Jammy Jellyfish)

1. **Download the Universal USB Installer**

Visit the official [Universal USB Installer website](#) to download the tool. Follow the instructions provided on the website or refer to a YouTube tutorial for creating a bootable USB.

2. **Download the Ubuntu ISO**

Download the Ubuntu 22.04.5 LTS (Jammy Jellyfish) ISO from the official [Ubuntu releases page](#).

3. **Create a Bootable USB Drive**

Use the Universal USB Installer tool with the downloaded ISO to create a bootable USB stick. Follow the on-screen prompts to properly set up the drive.

4. **Install Ubuntu on Your Machine**

Boot your computer from the USB drive. Follow the Ubuntu installation wizard to install Ubuntu 22.04.5 LTS and configure your system settings as needed.

1.2.2 Installing ROS 2 Humble Hawksbill

Recommended Method: Using Debian Packages

The official ROS 2 Humble Hawksbill documentation recommends installing from deb packages for a stable and straightforward setup. Follow the installation instructions on the official [ROS 2 Humble Hawksbill Installation Guide](#). This method installs pre-built binaries and generally covers all core dependencies needed for running ROS 2.

Instructions:

Set locale

Make sure you have a locale which supports UTF-8. The following settings are tested, although any UTF-8 supported locale should work.

```
$ 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
```

Setup Sources

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 ROS 2 Packages

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 to 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. Installing 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): This includes ROS, RViz, demos, and tutorials.

```
$ sudo apt install ros-humble-desktop
```

Development Tools: Compilers and other tools to build ROS packages.

```
$ sudo apt install ros-dev-tools
```

Environment Setup – Sourcing the Setup Script

Set up your environment by sourcing the following file (replace **.bash** with your shell if not using bash):

```
$ source /opt/ros/humble/setup.bash
```

Alternative: Building from Source

If you require customizations or intend to develop complex packages, you can also install ROS 2 Humble Hawksbill from source. Note that building from source may require additional dependency installations (similar to ROS1). For most beginners and for the purposes of this course, the deb package installation is recommended.



Sourcing the Setup Script

After installing ROS 2, source the setup script to ensure your environment is correctly configured:

```
$ source /opt/ros/humble/setup.bash
```

Optionally, add the sourcing command to your shell's initialization file (e.g., `.bashrc`) so that the ROS environment variables are automatically loaded every time a new terminal session is opened (replace `.bash` with your shell if not using bash).

```
$ echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

1.3 Try Some Examples

1.3.1 Talker-Listener

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

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

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

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

You should see the talker publishing messages and the listener confirming reception, verifying that both the C++ and Python APIs are working properly.

1.3.2 Turtlesim: Publish and Move a Turtle

Another example you can try is the `turtlesim` package, which demonstrates basic ROS 2 communication using a simulated turtle.

First, ensure that the package is installed:

```
$ sudo apt install ros-humble-turtlesim
```

Then, in a new terminal, launch the turtlesim node:

```
$ ros2 run turtlesim turtlesim_node
```

Next, open another terminal, use the `teleop` node to control the turtle with your keyboard:

```
$ ros2 run turtlesim turtle_teleop_key
```

Now, pressing the arrow keys will move the turtle in the corresponding direction. This example demonstrates publishing velocity commands to a topic that the turtlesim node subscribes to, allowing interaction with the simulated environment.



1.4 Configuring the Development Environment

1.4.1 Installing Visual Studio Code

Visual Studio Code (VS Code) is a versatile code editor that supports various programming languages and development environments. To install it on Ubuntu 22.04, follow these steps:

1. Using the Ubuntu Software Center

- (a) Open the *Ubuntu Software* application from the applications menu.
- (b) In the search bar, type *Visual Studio Code*.
- (c) Locate *Visual Studio Code* in the search results and click *Install*.

2. Using the Official .deb Package

- (a) Visit the official VS Code download page: <https://code.visualstudio.com/>.
- (b) Click on the *.deb* package suitable for Debian/Ubuntu.
- (c) Once downloaded, open a terminal and navigate to the directory containing the downloaded file.
- (d) Install the package using:

```
$ sudo apt install ./<file>.deb
```

Replace *<file>* with the actual filename. This method ensures that VS Code is added to your system repositories and receives updates automatically.

1.4.2 Installing Recommended VS Code Extensions

Enhance your development experience by installing the following VS Code extensions:

- **Python** - Provides rich support for Python, including features such as IntelliSense, linting, and debugging.
- **C++** - Offers C++ IntelliSense, debugging, and code browsing.
- **CMake** - Simplifies working with CMake projects.
- **CMake Tools** - Provides CMake project integration.
- **ROS** - Adds support for Robot Operating System (ROS) development.
- **ROS2** - Facilitates ROS 2 development with code snippets and other utilities.
- **XML** - Enhances XML editing capabilities.
- **XML Tools** - Offers additional functionalities for XML files.
- **Indent-Rainbow** - Highlights indentation levels with different colors.
- **vscode-icons** - Adds file icons for better visual identification.
- **Error Lens** - Displays inline error messages in the code editor.

To install these extensions:

1. Open VS Code.
2. Navigate to the *Extensions* view by clicking on the square icon in the sidebar or pressing **Ctrl+Shift+X**.
3. Search for each extension by name and click *Install*.



1.4.3 Installing Terminator for Multiple Terminals

Terminator is a terminal emulator that allows splitting the window into multiple terminals, facilitating simultaneous operations. To install Terminator:

1. Open a terminal.
2. Update the package list:

```
$ sudo apt update
```

3. Install Terminator:

```
$ sudo apt install terminator
```

4. Launch Terminator from the applications menu, by typing **terminator** in the terminal, or by pressing **Ctrl+Alt+T**.

With these tools and extensions, your development environment will be well-equipped for ROS 2 projects.

1.5 colcon Configuration

colcon is the command-line tool used in ROS 2 to build sets of packages in a workspace. It automatically detects packages (via **package.xml**), resolves dependencies, and builds them (often in parallel) to simplify the development process. **colcon** also supports options like **--symlink-install** for faster iterative development.

1.5.1 Installing colcon

Update your package list and ensure that the common **colcon** extensions are installed. Although these packages are usually installed as part of the **ros-humble-desktop** and **ros-dev-tools** installations, it is good practice to verify their presence by running the following commands:

```
$ sudo apt update
$ sudo apt install python3-colcon-common-extensions
```

1.5.2 Enabling colcon Argument Completion

To simplify command usage with **colcon**, add the argument completion script to your shell initialization file (e.g., **.bashrc**). Execute the following commands:

```
$ echo "source /usr/share/colcon_argcomplete/hook/colcon_argcomplete.bash" >> ~/.bashrc
$ source ~/.bashrc
```

You can verify that the sourcing command was added by viewing your **.bashrc** file:

```
$ cat ~/.bashrc
```



1.6 Practice Assignment: Running ROS 2 Examples

In this assignment, you will run the ROS 2 examples presented in Section 1.3 on your computer and capture evidence of successful execution using a screenshot. Submit your evidence in the designated assignment area on Canvas (within the ROS 2 module) as a PNG file. The filename must follow this format:

FirstnameLastname_evidence_wXsY.png,

where **wX** and **sY** indicate the week and session numbers respectively, for example:

EduardoDavila_evidence_w1s1.png.

1.6.1 Examples to Try

Talker-Listener

After installing **ros-humble-desktop** (and optionally **Terminator**), try the following:

- In Terminal 1, source the setup file and run a C++ talker:

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

- In Terminal 2, source the setup file and run a Python listener:

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

You should observe that the talker publishes messages and the listener confirms their reception.

Turtlesim: Publish and Move a Turtle

Another example is provided by the **turtlesim** package:

- In Terminal 3, launch the turtlesim node:

```
$ ros2 run turtlesim turtlesim_node
```

- In Terminal 4, run the teleoperation node to control the turtle:

```
$ ros2 run turtlesim turtle_teleop_key
```

Use the arrow keys to move the turtle. This example demonstrates publishing velocity commands to control the simulated turtle.

1.6.2 Submission Instructions

1. Run the examples as described above.
2. Capture a **screenshot showing the terminal output** where the examples are running successfully (see Figure 1.2 for an example).
3. Save the screenshot in PNG format.



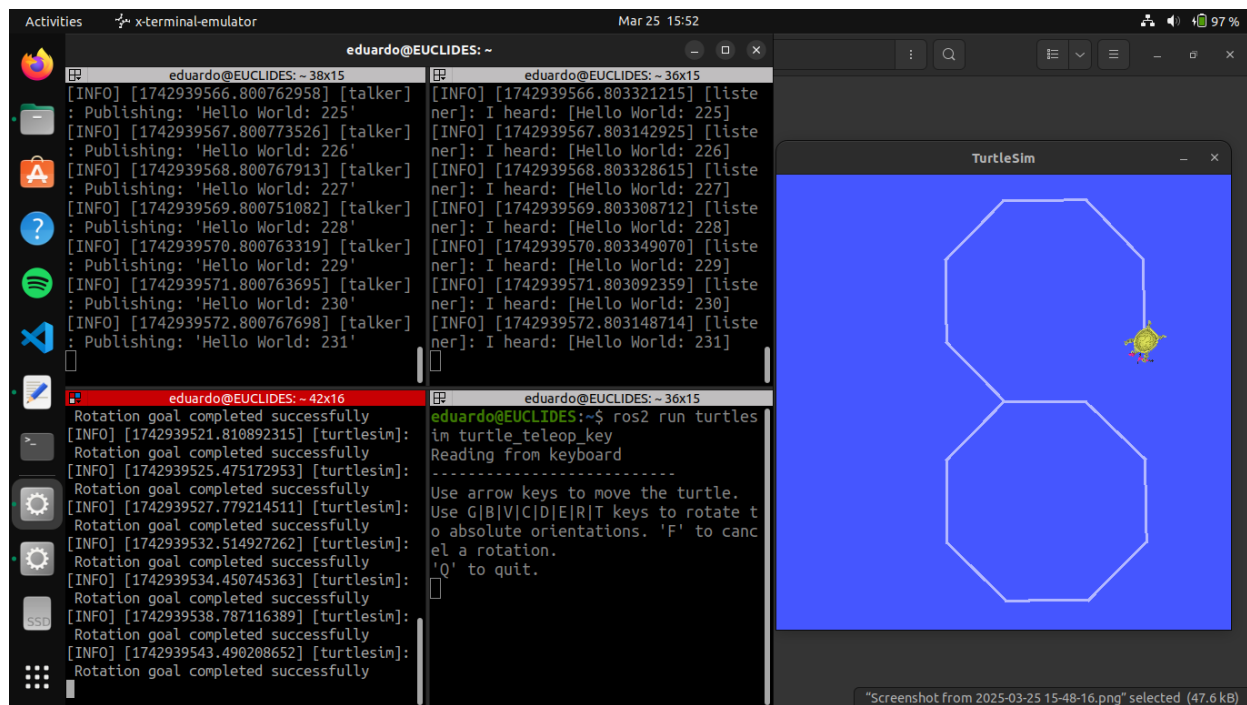


Figure 1.2: Example screenshot showing terminal outputs from both the Talker-Listener example and the TurtleSim node example in action.

4. Name the file according to the format: **FirstnameLastname_evidence_wXsY.png** (replace **X** with the week number and **Y** with the session number).
5. Submit your screenshot file as instructed by the course guidelines.

Note: Your machine's username and device name must be visible in the screenshot to verify the authenticity of the submission, as shown in Figure 1.2. Evidence that appears copied, unclear, or altered will be considered invalid and may result in a score of 0 for this assignment.

CHAPTER

ROS 2 Workspace, Package, and Node Development

Author: Dr. Eduardo de Jesús Dávila Meza.

 [EduardoDavila-AI-PhD](#)

2.1 Fundamental Concepts of ROS 2

ROS 2 is a [middleware](#) based on a strongly-typed, anonymous publish/subscribe mechanism that enables message passing between different processes. At the core of any ROS 2 system is the [ROS graph](#), which represents the network of nodes and the connections through which they communicate.

This section introduces the fundamental concepts necessary to understand the basics of ROS 2.

2.1.1 Workspace

A [ROS 2 workspace](#) is a directory that contains one or more ROS 2 packages. It serves as the working environment for building, developing, and testing ROS 2 applications.

Best practices recommend creating a separate directory for each workspace, with a meaningful name that reflects its purpose. Inside the workspace, it is common to include a [src](#) folder to store ROS 2 packages, ensuring an organized structure.

2.1.2 Packages

A [ROS 2 package](#) is the fundamental unit of software organization in ROS 2. It provides a structured way to manage code, facilitating its distribution, installation, and reuse. A package may contain:

- Nodes
- Libraries
- Configuration files

- Launch files
- Other resources necessary for execution

A single workspace can contain multiple packages, even if they use different build types (e.g., CMake, Python). However, packages cannot be nested within each other.

ROS 2 uses [ament](#) as its build system and [colcon](#) as its build tool. Packages can be created using either CMake or Python, each with minimum required contents, as detailed in Sections [2.3](#) and [2.4](#), respectively.

2.1.3 Nodes

A [ROS 2 node](#) is a fundamental component that represents a single process performing computation. Nodes communicate with each other through:

- [Topics](#) for continuous data exchange.
- [Services](#) for request-response interactions.
- [Actions](#) for long-running tasks with feedback.

Each node is part of the [ROS graph](#) and can communicate with other nodes within the same process, across different processes, or even across multiple machines. Nodes are designed to be modular, meaning that each node should focus on a single logical task.

A node can simultaneously act as:

- A [publisher](#) for sending messages.
- A [subscriber](#) for receiving messages.
- A [service client](#) for requesting a computation.
- A [service server](#) for providing a computation.
- An [action client](#) for initiating long-running tasks with feedback.
- An [action server](#) for executing long-running tasks with periodic feedback.

Connections between nodes are established dynamically, allowing for a modular and scalable system design.

2.1.4 Topics

[Topics](#) are a core mechanism in ROS 2 for message-based communication between nodes. They are used for continuous data exchange, such as transmitting sensor readings, robot state updates, or other real-time information.

ROS 2 follows a [publish/subscribe](#) model for topics, meaning:

- [Publishers](#) send data to a topic.
- [Subscribers](#) receive data from a topic.
- Multiple publishers and subscribers can exist on the same topic.
- Communication is [anonymous](#), so subscribers can receive messages without knowing which publisher transmitted the data.



Publish/Subscribe Model

The publish/subscribe system allows for flexible and decoupled communication:

- Publishers and subscribers communicate via a shared **topic name**.
- Multiple publishers and subscribers can exist on a topic simultaneously.
- When a publisher sends data, all subscribers of that topic receive it.

This architecture is often compared to a **bus system** in electrical engineering, where multiple devices can share a common communication channel.

Anonymous Communication

ROS 2 implements **anonymous communication**, meaning:

- A subscriber does not need to know which publisher sent a message.
- Publishers and subscribers can be dynamically replaced without affecting the system.
- Debugging and monitoring tools (e.g., **ros2 bag record**) can subscribe to topics without interrupting existing communication.

Strongly-Typed Messages

ROS 2 enforces **strong typing** in its publish/subscribe system to ensure data consistency and integrity. This guarantees:

1. **Strict Data Types:** Each field in a message adheres to a predefined data type.

```
uint32 field1
string field2
```

In this example, **field1** must always be an unsigned 32-bit integer, and **field2** must always be a string.

2. **Well-Defined Semantics:** Message contents adhere to clear conventions. For example, an IMU message includes a 3-dimensional angular velocity vector, where each component is explicitly defined in radians per second. This ensures consistency and prevents misinterpretation of the transmitted data.

2.2 Workspace Creation and Setup

A ROS 2 workspace is a directory that contains one or more ROS 2 packages, as described in Section 2.1. When you build your workspace using **colcon**, it automatically creates the **build**, **install**, and **log** directories. This structure helps manage dependencies and package configurations efficiently, ensuring a clean and organized development environment.

2.2.1 Creating the Workspace

After installing ROS 2, set up your workspace by creating a directory (e.g., **~/ros2_ws**) and adding a **src** folder:



```
$ mkdir -p ros2_ws/src
```

The best practice is to create and organize your packages inside the `src` folder to keep the workspace structure clean and maintainable.

2.2.2 Building the Workspace

Navigate to the workspace directory:

```
$ cd ros2_ws
```

Build the workspace using `colcon`:

```
$ colcon build
```

For development purposes, you might consider using the `--symlink-install` option. This allows changes in source files to take effect immediately without requiring a full rebuild:

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

If the build completes successfully, you will see the following confirmation message: `'colcon build' successful`, and the `build`, `install`, and `log` directories will be created within your workspace.

2.2.3 Sourcing the Workspace Environment

To overlay your workspace on top of the system installation and load the environment variables for your newly built packages, run:

```
$ source ./install/setup.bash
```

Note: Unlike the ROS 2 and `colcon` setup scripts, this sourcing command is not permanently added to your shell's initialization file. For learning purposes, you must run it in each new terminal session where you work with your ROS 2 workspace.

For more details on `colcon` and workspace management, refer to the [ROS 2 Colcon](#) tutorial.

2.3 Creating a ROS 2 Package for C++ Nodes

2.3.1 Navigating to the `src` Directory

Begin by opening a terminal and navigating to the `src` directory within your ROS 2 workspace (e.g., `~/ros2_ws`):

```
$ cd ~/ros2_ws/src
```

2.3.2 Creating the C++ Package

Use the `ros2 pkg create` command to create a new C++ package. Replace `my_cpp_package` with your desired package name and provide an appropriate description:



```
$ ros2 pkg create my_cpp_package --build-type ament_cmake --dependencies rclcpp std_msgs  
↳ --license Apache-2.0 --description "Your package description here"
```

Let's break down the components of this command:

- **my_cpp_package**: Specifies the name of the new package.
- **--build-type ament_cmake**: Indicates that the package uses C++ and will be built with **ament_cmake**.
- **--dependencies rclcpp std_msgs**: Lists the package dependencies, in this case, **rclcpp** (the ROS 2 C++ API) and **std_msgs** (standard message definitions).
- **--license Apache-2.0**: Sets the license type for the package.
- **--description "Your package description here"**: Provides a brief description of the package.

2.3.3 Building the Package

After creating the package, navigate back to the root of your workspace and build the package using **colcon**:

```
$ cd ~/ros2_ws  
$ colcon build --packages-select my_cpp_package
```

Alternatively, to build all packages in the workspace:

```
$ colcon build
```

2.3.4 Examining the Package Contents

Navigate to the newly created package directory, **my_cpp_package**, and list its contents:

```
$ cd src/my_cpp_package  
$ ls
```

You should see the following key files and directories:

- **CMakeLists.txt**: Contains instructions for building the code within the package using CMake.
- **include/**: Directory containing the public header files for the package.
- **package.xml**: Defines metadata about the package, including its name, version, description, licenses, maintainers, authors, and dependencies.
- **src/**: Directory containing the source code files for the package.

2.3.5 Detailed Examination of Key Package Files

package.xml

The **package.xml** file contains essential metadata about the ROS 2 package. Key elements include:

- **<name>**: Specifies the package's name.
- **<version>**: Indicates the current version of the package.
- **<description>**: Provides a brief overview of the package's purpose.
- **<maintainer>**: Contains contact information for the package maintainer.



- `<license>`: Details the licensing information.
- `<depend>`: Lists build-time and runtime dependencies required by the package.

To examine the `package.xml` file, use the following command:

```
$ code package.xml
```

Ensure that the dependencies listed here match those specified during package creation.

CMakeLists.txt

The `CMakeLists.txt` file contains instructions for building the package using CMake. It specifies details such as the minimum required CMake version, project name, dependencies, include directories, source files, and targets to be built.

To inspect the `CMakeLists.txt` file, execute:

```
$ code CMakeLists.txt
```

2.3.6 Ensuring Consistency Between `package.xml` and `CMakeLists.txt`

It is crucial that the information in `package.xml` and `CMakeLists.txt` is consistent. Discrepancies between these files can lead to build or runtime issues. Ensure that details like the package name, version, description, maintainer, license, and dependencies align across both files.

By following these steps, you have created a C++-based ROS 2 package, built it, and examined its structure and configuration files. This foundational setup is essential for developing and organizing your ROS 2 C++ nodes effectively.

For more details on package creation and management, refer to the official [Creating Your First ROS 2 Package](#) tutorial.

2.4 Creating a ROS 2 Package for Python Nodes

2.4.1 Navigating to the `src` Directory

Begin by opening a terminal and navigating to the `src` directory within your ROS 2 workspace (e.g., `~/ros2_ws`):

```
$ cd ~/ros2_ws/src
```

2.4.2 Creating the Python Package

Use the `ros2 pkg create` command to create a new Python package. Replace `my_py_package` with your desired package name and provide an appropriate description:

```
$ ros2 pkg create my_py_package --build-type ament_python --dependencies rclpy std_msgs  
↪ --license Apache-2.0 --description "Your package description here"
```



Let's break down the components of this command:

- `my_py_package`: Specifies the name of the new package.
- `-build-type ament_python`: Indicates that the package uses Python and will be built with `ament_python`.
- `-dependencies rclpy std_msgs`: Lists the package dependencies, in this case, `rclpy` (the ROS 2 Python API) and `std_msgs` (standard message definitions).
- `-license Apache-2.0`: Sets the license type for the package.
- `-description "Your package description here"`: Provides a brief description of the package.

2.4.3 Building the Package

After creating the package, navigate back to the root of your workspace and build the package using `colcon`:

```
$ cd ~/ros2_ws
$ colcon build --packages-select my_py_package
```

Alternatively, to build all packages in the workspace:

```
$ colcon build
```

2.4.4 Examining the Package Contents

Navigate to the newly created package directory, `my_py_package`, and list its contents:

```
$ cd src/my_py_package
$ ls
```

You should see the following key files and directories:

- `my_py_package/`: Directory with the same name as your package, used by ROS 2 tools to find your package, contains `__init__.py`.
- `package.xml`: Defines metadata about the package, including its name, version, description, licenses, maintainers, authors, and dependencies.
- `setup.py`: Script for installing and setting up the package using Python's `setuptools`.
- `setup.cfg`: Configuration file for `setuptools`.
- `resource/`: Directory containing resource files.
- `test/`: Directory designated for test files.

2.4.5 Detailed Examination of Key Package Files

`package.xml`

The `package.xml` file contains essential metadata about the ROS 2 package. Key elements include:

- `<name>`: Specifies the package's name.
- `<version>`: Indicates the current version of the package.
- `<description>`: Provides a brief overview of the package's purpose.
- `<maintainer>`: Contains contact information for the package maintainer.



- `<license>`: Details the licensing information.
- `<depend>`: Lists build-time and runtime dependencies required by the package.

To examine the `package.xml` file, use the following command:

```
$ code package.xml
```

Ensure that the dependencies listed here match those specified during package creation.

`setup.py`

The `setup.py` script utilizes `setuptools` to configure how the package is installed and structured.. It specifies details such as the package name, version, author information, license, and included packages or modules.

To inspect the `setup.py` file, execute:

```
$ code setup.py
```

2.4.6 Ensuring Consistency Between `package.xml` and `setup.py`

It is crucial that the information in `package.xml` and `setup.py` is consistent. Inconsistencies may cause build failures or unexpected runtime behavior. Ensure that details like the package name, version, description, maintainer, license, and dependencies align across both files.

By following these steps, you have created a Python-based ROS 2 package, built it, and examined its structure and configuration files. This foundational setup is essential for developing and organizing your ROS 2 Python nodes effectively.

For more details on package creation and management, refer to the official [Creating a ROS 2 Package](#) tutorial.

2.5 Creating a ROS 2 Publisher Node with C++

2.5.1 Setting Up the C++ Node

Navigating to the Package Directory

Open a terminal and navigate to your C++ package directory within your ROS 2 workspace (e.g., `~/ros2_ws`):

```
$ cd ros2_ws/src/my_cpp_package/src
```

Replace `my_cpp_package` with your actual package name.

Creating the C++ Node File

Create a new C++ file for your publisher node, for example, `publisher_cpp.cpp`:

```
$ touch publisher_cpp.cpp
```


Examine the folder contents to verify the new file has been created:

```
$ ls
```

You should now see the `publisher_cpp.cpp` file.

2.5.2 Editing the C++ Node

Opening the Workspace in VS Code

Navigate back to the root of your workspace:

```
$ cd ../../..
```

Then, open the workspace in Visual Studio Code:

```
$ code .
```

In VS Code, locate and open the `publisher_cpp.cpp` file for editing.

Implementing the Node Code

Below is an example implementation of a simple ROS 2 publisher node in C++:

```
#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 CppPublisher : public rclcpp::Node
{
public:
    CppPublisher()
    : Node("cpp_publisher"), count_(0)
    {
        RCLCPP_INFO(this->get_logger(), "C++ Publisher node has been started");
        publisher_ = this->create_publisher<std_msgs::msg::String>("cpp_topic", 10);
        timer_ = this->create_wall_timer(
            500ms, std::bind(&CppPublisher::timer_callback, this));
    }
};
```



```

private:
    void timer_callback()
    {
        auto message = std_msgs::msg::String();
        message.data = "Hello, this is Eduardo from the C++ Publisher: " +
        ↪ 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<CppPublisher>());
    rclcpp::shutdown();
    return 0;
}

```

This code defines a node that publishes a "Hello, World" message along with a counter to the topic `publisher_topic` every 0.5 seconds.

Breaking Down the Publisher Node Code

We now examine each part of the code in detail to understand how the ROS 2 C++ publisher node works.

1. Includes and Namespace

```

#include <chrono>
#include <functional>
#include <memory>
#include <string>

#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

using namespace std::chrono_literals;

```

- `<chrono>` and `<memory>` are used for time management and smart pointers.
- `<functional>` and `<string>` support function binding and string operations.
- `rclcpp/rclcpp.hpp` provides the ROS 2 C++ API.
- `std_msgs/msg/string.hpp` defines the standard String message type.
- The `using namespace std::chrono_literals;` statement allows the use of time literals (e.g., `500ms`) for time durations.

2. Defining the Node Class

```
class CppPublisher : public rclcpp::Node
{
public:
    CppPublisher()
    : Node("cpp_publisher"), count_(0)
    {
        RCLCPP_INFO(this->get_logger(), "C++ Publisher node has been started");
        publisher_ = this->create_publisher<std_msgs::msg::String>("cpp_topic", 10);
        timer_ = this->create_wall_timer(
            500ms, std::bind(&CppPublisher::timer_callback, this));
    }
};
```

- The `CppPublisher` class inherits from `rclcpp::Node` to create a new ROS 2 node.
- **public:** This access specifier indicates that all members declared after it (until another access specifier is encountered) are publicly accessible. This means the constructor and any public methods can be accessed from outside the class.
- The constructor `CppPublisher()` initializes the node with the name `"cpp_publisher"` and sets the counter `count_` to 0.
- `RCLCPP_INFO(this->get_logger(), "C++ Publisher node has been started");` logs an informational message.
- A publisher is created to send `std_msgs::msg::String` messages on the `"publisher_topic"` topic with a queue size of 10.
- A wall timer is set up to call the `timer_callback` method every 500 milliseconds.

3. Timer Callback Method

```
private:
    void timer_callback()
    {
        auto message = std_msgs::msg::String();
        message.data = "Hello, this is Eduardo from the C++ Publisher: " +
        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_;
};
```

- **private:** This access specifier indicates that all members declared after it are accessible only within the class itself. This encapsulation prevents external access to these members, ensuring they can only be modified or called by member functions of the class.
- The `timer_callback` method is invoked periodically by the timer. It creates a new message, sets its `data` field to include the current counter value, logs the message, and publishes it on the topic `publisher_topic`.
- `rclcpp::TimerBase::SharedPtr timer_;` A shared pointer to a timer object that triggers the callback at regular intervals.



- `rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;` A shared pointer to a publisher that sends messages of type `std_msgs::msg::String`.
- `size_t count_;` A counter used to generate a unique message each time the callback is invoked.

4. The Main Function

```
int main(int argc, char * argv[])
{
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<CppPublisher>());
    rclcpp::shutdown();
    return 0;
}
```

- `rclcpp::init(argc, argv)` initializes ROS 2 communication by parsing command-line arguments and setting up necessary middleware.
- `rclcpp::spin(std::make_shared<CppPublisher>())`: Creates an instance of the `CppPublisher` node and enters a loop, keeping the node active to process callbacks (such as those from the timer).
- `rclcpp::shutdown()`: Gracefully shuts down ROS 2, releasing all resources when the node stops.
- `return 0;`: Returns 0 to indicate successful program termination.

This detailed breakdown explains each component of the ROS 2 C++ publisher node. For further details and additional context, refer to the [Creating ROS 2 Nodes \(C++\)](#) tutorial.

2.5.3 Integrating the Publisher Node into the Package

Modifying `CMakeLists.txt`

To enable ROS 2 to build and run your C++ node, modify the `CMakeLists.txt` file of your package. Open `CMakeLists.txt` in VS Code and add or verify the following lines:

```
# find dependencies
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)

add_executable(node_publisher_cpp src/publisher_cpp.cpp)
ament_target_dependencies(node_publisher_cpp rclcpp std_msgs)

install(TARGETS
    node_publisher_cpp
    DESTINATION lib/${PROJECT_NAME})
```

Breaking down each line:

- `find_package(ament_cmake REQUIRED)` locates the ament build system.
- `find_package(rclcpp REQUIRED)` and `find_package(std_msgs REQUIRED)` ensure that the required dependencies are found.



- `add_executable(node_publisher_cpp src/publisher_cpp.cpp)` creates an executable from your source file.
- `ament_target_dependencies(node_publisher_cpp rclcpp std_msgs)` links the required libraries.
- The `install` command specifies where the executable should be installed.

Building the Package

After saving the changes to `CMakeLists.txt`, build your package. In the current terminal session, from the root of the workspace, compile with:

```
$ colcon build
```

Alternatively, to build only the C++ package:

```
$ colcon build --packages-select my_cpp_package
```

For development purposes, you might consider using the `--symlink-install` option so that changes in the source files are immediately reflected without requiring a full rebuild:

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

or, more specifically:

```
$ colcon build --packages-select my_cpp_package --symlink-install
```

If the build completes successfully, you will see the following confirmation message: `'colcon build' successful`.

2.5.4 Running the Publisher Node as a Standalone Executable (Optional)

You can run your C++ publisher node outside of the ROS 2 environment. Open a new terminal (or use an additional session in Terminator) and navigate to the location where the executable was installed. This could be either:

```
$ cd build/my_cpp_package
```

or:

```
$ cd install/my_cpp_package/lib/my_cpp_package
```

as the `install` command specified, in the `CMakeLists.txt` file, where the executable should be installed. Then, run the executable:

```
$ ./node_publisher_cpp
```

as the `add_executable` command specified, in the `CMakeLists.txt` file, how the executable should be named.

Sourcing the Environment

Before running the node, in the first (or current) terminal session from the root of the workspace, source the workspace's setup file to load the necessary environment variables:

```
$ source ./install/setup.bash
```

This step ensures that ROS 2 and the built packages are properly configured.



2.5.5 Running the ROS 2 Publisher Node with ROS 2

Execute your C++ publisher node using the `ros2 run` command:

```
$ ros2 run my_cpp_package node_publisher_cpp
```

At this point, if everything is set up correctly, your node should be actively publishing messages to the topic `publisher_topic`.

2.5.6 Summary of Key Identifiers

At the end of the creation and configuration of the ROS 2 C++ publisher node, it is important to distinguish between the following identifiers used:

1. `publisher_cpp.cpp`: The filename of your C++ source file.
2. `cpp_publisher`: The name of the node as defined in the constructor of your C++ class.
3. `node_publisher_cpp`: The name of the executable specified in `CMakeLists.txt` under the `add_executable` command.

Ensuring these identifiers are correctly set and consistently used is crucial for the proper functioning of your ROS 2 C++ nodes.

For more details on node creation and setup in C++, refer to the official [Creating ROS 2 Nodes \(C++\)](#) tutorial.

2.6 Creating a ROS 2 Subscriber Node with C++

2.6.1 Setting Up the C++ Node

Navigating to the Package Directory

Open a terminal and navigate to your C++ package directory within your ROS 2 workspace (e.g., `~/ros2_ws`):

```
$ cd ros2_ws/src/my_cpp_package/src
```

Replace `my_cpp_package` with your actual package name.

Creating the C++ Node File

Create a new C++ file for your subscriber node, for example, `subscriber_cpp.cpp`:

```
$ touch subscriber_cpp.cpp
```

Then, list the folder contents to verify the file has been created:

```
$ ls
```

You should now see the `subscriber_cpp.cpp` file.



2.6.2 Editing the C++ Node

Opening the Workspace in VS Code

Navigate back to the root of your workspace:

```
$ cd ../../..
```

Then, open the workspace in Visual Studio Code:

```
$ code .
```

In VS Code, locate and open the `subscriber_cpp.cpp` file for editing.

Implementing the Node Code

Below is an example implementation of a simple ROS 2 subscriber node in C++:

```
#include <memory>
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

using std::placeholders::_1;

class CppSubscriber : public rclcpp::Node
{
public:
    CppSubscriber()
    : Node("cpp_subscriber")
    {
        RCLCPP_INFO(this->get_logger(), "C++ Subscriber node has been started");
        subscription_ = this->create_subscription<std_msgs::msg::String>(
            "py_topic", 10, std::bind(&CppSubscriber::topic_callback, this, _1));
    }

private:
    void topic_callback(const std_msgs::msg::String::SharedPtr 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<CppSubscriber>());
    rclcpp::shutdown();
    return 0;
}
```



This code defines a node that subscribes to the topic `publisher_topic` and logs the messages received.

Breaking Down the Subscriber Node Code

1. Includes and Namespace

```
#include <memory>
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

using std::placeholders::_1;
```

- `<memory>` is used for smart pointers.
- `rclcpp/rclcpp.hpp` provides the ROS 2 C++ API.
- `std_msgs/msg/string.hpp` defines the standard String message type.
- `using std::placeholders::_1;` simplifies binding the callback function.

2. Defining the Node Class

```
class CppSubscriber : public rclcpp::Node
{
public:
    CppSubscriber()
    : Node("cpp_subscriber")
    {
        RCLCPP_INFO(this->get_logger(), "C++ Subscriber node has been started");
        subscription_ = this->create_subscription<std_msgs::msg::String>(
            "py_topic", 10, std::bind(&CppSubscriber::topic_callback, this, _1));
    }
}
```

- The `CppSubscriber` class inherits from `rclcpp::Node` to create a new ROS 2 node.
- `public:` makes the constructor accessible outside the class.
- The constructor `CppSubscriber()` initializes the node with the name `"cpp_subscriber"` and creates a subscription to the `"publisher_topic"` topic with a queue size of 10.
- The subscription's callback is bound using `std::bind`.

3. Listener Callback Method

```
private:
    void topic_callback(const std_msgs::msg::String::SharedPtr msg) const
    {
        RCLCPP_INFO(this->get_logger(), "I heard: '%s'", msg->data.c_str());
    }
    rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
};
```

- `private:` restricts access to the callback and the subscription pointer, encapsulating internal details.
- The `topic_callback` method is invoked when a new message is received. It logs the content of the message.



4. The Main Function

```
int main(int argc, char * argv[])
{
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<CppSubscriber>());
    rclcpp::shutdown();
    return 0;
}
```

- `rclcpp::init(argc, argv)` initializes ROS 2 communication by parsing command-line arguments and setting up necessary middleware.
- `rclcpp::spin`: Keeps the node active and processes callbacks.
- `rclcpp::shutdown()`: Gracefully shuts down ROS 2, cleaning up all resources when the node stops.
- `return 0`: Returns 0 to indicate successful program termination.

This detailed breakdown explains each component of the ROS 2 C++ subscriber node. For further details and additional context, refer to the [Creating ROS 2 Nodes \(C++\)](#) tutorial.

2.6.3 Integrating the Subscriber Node into the Package

Modifying `CMakeLists.txt`

To build and run your C++ subscriber node, modify the `CMakeLists.txt` file of your package. Open `CMakeLists.txt` in VS Code and add or verify the following lines, keeping any other existing ROS 2 node setup (e.g., `node_publisher_cpp`):

```
# find dependencies
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)

add_executable(node_publisher_cpp src/publisher_cpp.cpp)
ament_target_dependencies(node_publisher_cpp rclcpp std_msgs)

add_executable(node_subscriber_cpp src/subscriber_cpp.cpp)
ament_target_dependencies(node_subscriber_cpp rclcpp std_msgs)

install(TARGETS
    node_publisher_cpp
    node_subscriber_cpp
    DESTINATION lib/${PROJECT_NAME})
```

Breaking down each line:

- `find_package(ament_cmake REQUIRED)` locates the ament build system.
- `find_package(rclcpp REQUIRED)` and `find_package(std_msgs REQUIRED)` ensure that the required dependencies are found.
- `add_executable(node_subscriber_cpp src/subscriber_cpp.cpp)` creates an executable from your source file.



- `ament_target_dependencies(node_subscriber_cpp rclcpp std_msgs)` links the required libraries.
- The `install` command specifies where the executable should be installed.

Building the Package

After updating `CMakeLists.txt`, build your package. In the current terminal session from the root of your workspace, compile with:

```
$ colcon build
```

Alternatively, to build only the C++ package:

```
$ colcon build --packages-select my_cpp_package
```

For development, you might consider using the `--symlink-install` option so that changes in the source files are immediately reflected without requiring a full rebuild:

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

or, more specifically:

```
$ colcon build --packages-select my_cpp_package --symlink-install
```

If the build completes successfully, you will see the following confirmation message: `'colcon build' successful`.

2.6.4 Running the Subscriber Node as a Standalone Executable (Optional)

You can run your C++ subscriber node outside of the ROS 2 environment. Open a new terminal (or use an additional session in Terminator) and navigate to the directory where the executable was installed. This could be either:

```
$ cd build/my_cpp_package
```

or:

```
$ cd install/my_cpp_package/lib/my_cpp_package
```

as the `install` command specified, in the `CMakeLists.txt` file, where the executable should be installed. Then, run the executable:

```
$ ./node_subscriber_cpp
```

as the `add_executable` command specified, in the `CMakeLists.txt` file, how the executable should be named.

Sourcing the Environment

Before running the node with ROS 2, in the first (or current) terminal session from the root of your workspace, source the workspace's setup file to load the necessary environment variables:

```
$ source ./install/setup.bash
```

This ensures that ROS 2 and your built packages are properly configured.



2.6.5 Running the ROS 2 Subscriber Node with ROS 2

Execute your C++ subscriber node using the `ros2 run` command:

```
$ ros2 run my_cpp_package node_subscriber_cpp
```

At this point, if everything is set up correctly, your node should be actively receiving and logging messages from the topic `publisher_topic`.

2.6.6 Summary of Key Identifiers

At the end of the creation and configuration of the ROS 2 C++ subscriber node, it is important to distinguish between the following identifiers used:

1. `subscriber_cpp.cpp`: The filename of your C++ source file.
2. `cpp_subscriber`: The name of the node as defined in the constructor of your C++ class.
3. `node_subscriber_cpp`: The name of the executable specified in `CMakeLists.txt` under the `add_executable` command.

Ensuring these identifiers are correctly set and consistently used is crucial for the proper functioning of your ROS 2 C++ nodes.

For more details on node creation and setup in C++, refer to the official [Creating ROS 2 Nodes \(C++\)](#) tutorial.

2.7 Creating a ROS 2 Publisher Node with Python

2.7.1 Setting Up the Python Node

Navigating to the Package Directory

Open a terminal and navigate to your Python package directory within the ROS 2 workspace (e.g., `~/ros2_ws`):

```
$ cd ros2_ws/src/my_py_package/my_py_package
```

Replace `my_py_package` with your actual package name.

Creating the Python Node File

Create a new Python file for your publisher node, for example, `publisher_py.py`:

```
$ touch publisher_py.py
```

Examine the folder contents to verify the new file has been created:

```
$ ls
```

You should now see both the `__init__.py` and `publisher_py.py` files.



Making the Python File Executable

Ensure the new Python file is executable:

```
$ chmod +x publisher_py.py
```

Examine the folder contents again to confirm that `publisher_py.py` now appears in a different color, indicating that it is executable.

2.7.2 Editing the Python Node

Opening the Workspace in VS Code

Navigate back to the root of your workspace:

```
$ cd ../../..
```

Then, open the workspace in Visual Studio Code:

```
$ code .
```

In VS Code, locate and open the `publisher_py.py` file for editing.

Implementing the Node Code

Below is an example implementation of a simple ROS 2 publisher node in Python:

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class PyPublisher(Node):

    def __init__(self):
        super().__init__('py_publisher')
        self.get_logger().info("Python Publisher node has been started")
        self.publisher_ = self.create_publisher(String, 'py_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, this is Eduardo from the Python Publisher: %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)

    py_publisher = PyPublisher()

    rclpy.spin(py_publisher)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    py_publisher.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

This script defines a node that publishes a "Hello, World" message along with a counter to the topic named `publisher_topic` every 0.5 seconds.

Breaking Down the Publisher Node Code

We now examine each part of the code in detail to understand how the ROS 2 Python publisher node works.

1. Shebang and Imports

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from std_msgs.msg import String
```

- The shebang line (`#!/usr/bin/env python3`) ensures that the script is executed with Python 3.
- `rclpy` is the ROS 2 Python client library, which provides the tools needed to write ROS 2 nodes.
- `Node` is the base class for creating ROS 2 nodes.
- `String` is the message type imported from the standard messages package.

2. Defining the Node Class

```
class PyPublisher(Node):

    def __init__(self):
        super().__init__('py_publisher')
        self.get_logger().info("Python Publisher node has been started")
        self.publisher_ = self.create_publisher(String, 'py_topic', 10)
        timer_period = 0.5 # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.i = 0
```

- The `PyPublisher` class inherits from `Node` to create a new ROS 2 node.
- In the `__init__` method:
 - `super().__init__('py_publisher')` initializes the node with the name `py_publisher`.



- `self.get_logger().info("Python Publisher node has been started")` logs an informational message when the node starts.
- `self.create_publisher(String, 'publisher_topic', 10)` creates a publisher that will publish messages of type `String` to the topic `publisher_topic` with a queue size of 10.
- `self.create_timer(timer_period, self.timer_callback)` sets up a timer to call the `timer_callback` method every 0.5 seconds.
- `self.i = 0` initializes a counter for use in the published messages.

3. Timer Callback Method

```
def timer_callback(self):
    msg = String()
    msg.data = 'Hello, this is Eduardo from the Python Publisher: %d' % self.i
    self.publisher_.publish(msg)
    self.get_logger().info('Publishing: "%s"' % msg.data)
    self.i += 1
```

- The `timer_callback` method is called periodically by the timer.
- A new `String` message is created, and its `data` field is set to include the current counter value.
- The message is published to `publisher_topic`.
- An informational log displays the published message.
- The counter `self.i` is incremented for the next callback.

4. The Main Function

```
def main(args=None):
    rclpy.init(args=args)

    py_publisher = PyPublisher()

    rclpy.spin(py_publisher)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    py_publisher.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

- `rclpy.init(args=args)` initializes ROS 2 communication by parsing command-line arguments and setting up necessary middleware.
- An instance of the `PyPublisher` node is created.
- `rclpy.spin(py_publisher)` keeps the node active, processing callbacks until a shutdown signal is received.
- Once the node is stopped, it is explicitly destroyed and `rclpy.shutdown()` is called to clean up resources.

This detailed breakdown explains each component of the ROS 2 Python publisher node. For further details and additional context, refer to the [Creating ROS 2 Nodes \(Python\)](#) tutorial.



2.7.3 Running the Publisher Node Standalone (Optional)

After editing the file, you can run your Python publisher node out of ROS 2 environment. For this, open a new terminal (or use an additional session in Terminator) and navigate to the location of the Python file:

```
$ cd src/my_py_package/my_py_package
```

Then, test the Python file by running it with one of the following commands:

```
$ ./publisher_py.py
```

to run it as an executable, or

```
$ python3 publisher_py.py
```

to run it as a Python script.

2.7.4 Integrating the Publisher Node into the Package

Modifying `setup.py`

To enable ROS 2 to recognize and execute your publisher node, you need to specify it in the `setup.py` file of your package. Open `setup.py` in VS Code and locate the `entry_points` field. Then, add your node as follows:

```
entry_points={
    'console_scripts': [
        'node_publisher_py = my_py_package.publisher_py:main',
    ],
},
```

Breaking down the components of this specification:

- `node_publisher_py`: Specifies the command-line executable name that you will use to run the node.
- `my_py_package`: Indicates the name of your package.
- `publisher_py`: Specifies the Python file (without the `.py` extension) containing the node.
- `main`: Refers to the function that will be executed when the node runs.

Building the Package

After saving the changes to `setup.py`, build your package. In the first terminal session, from the root of the workspace, compile with:

```
$ colcon build
```

Alternatively, to build only the Python package:

```
$ colcon build --packages-select my_py_package
```

For development purposes, consider using the `--symlink-install` option so that changes in the source files are immediately reflected without requiring a full rebuild:

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



or, more specifically:

```
$ colcon build --packages-select my_py_package --symlink-install
```

If the build completes successfully, you will see the following confirmation message: `'colcon build' successful`.

Sourcing the Environment

Before running the node, in the first (or current) terminal session from the root of the workspace, source the workspace's setup file to load the necessary environment variables:

```
$ source ./install/setup.bash
```

This step ensures that ROS 2 and the built packages are properly configured.

2.7.5 Running the ROS 2 Publisher Node with ROS 2

Execute your Python publisher node using the `ros2 run` command:

```
$ ros2 run my_py_package node_publisher_py
```

At this point, your node should be actively publishing messages to the topic `publisher_topic`.

2.7.6 Summary of Key Identifiers

At the end of the creation and configuration of the ROS 2 publisher node, it is important to distinguish between the different identifiers used:

1. `publisher_py.py`: The filename of your Python script.
2. `py_publisher`: The name of the node as defined in the `super().__init__()` call within your script.
3. `node_publisher_py`: The command-line executable name specified in `setup.py` under the `entry_points` field.

Ensuring these identifiers are correctly set and consistently used is crucial for the proper functioning of your ROS 2 Python nodes.

For more details on node creation and setup in Python, refer to the official [Creating ROS 2 Nodes \(Python\)](#) tutorial.

2.8 Creating a ROS 2 Subscriber Node with Python

2.8.1 Setting Up the Python Node

Navigating to the Package Directory

Open a terminal and navigate to your Python package directory within the ROS 2 workspace (e.g., `~/ros2_ws`):

```
$ cd ros2_ws/src/my_py_package/my_py_package
```

Replace `my_py_package` with your actual package name.



Creating the Python Node File

Create a new Python file for your subscriber node, for example, `subscriber_py.py`:

```
$ touch subscriber_py.py
```

Examine the folder contents to verify the new file has been created:

```
$ ls
```

You should now see both the `__init__.py` and `subscriber_py.py` files.

Making the Python File Executable

Ensure the new Python file is executable:

```
$ chmod +x subscriber_py.py
```

Check the folder contents again to confirm that `subscriber_py.py` now appears in a different color, indicating that it is executable.

2.8.2 Editing the Python Node

Opening the Workspace in VS Code

Navigate back to the root of your workspace:

```
$ cd ../../..
```

Then, open the workspace in Visual Studio Code:

```
$ code .
```

In VS Code, locate and open the `subscriber_py.py` file for editing.

Implementing the Node Code

Below is an example implementation of a simple ROS 2 subscriber node in Python:

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class PySubscriber(Node):

    def __init__(self):
        super().__init__('py_subscriber')
        self.get_logger().info("Python Subscriber node has been started")
        self.subscription = self.create_subscription(
            String,
            'cpp_topic',
            self.listener_callback,
            10)
        self.subscription  # 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)

    py_subscriber = PySubscriber()

    rclpy.spin(py_subscriber)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    py_subscriber.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()

```

This script defines a node that subscribes to the topic `publisher_topic` and logs the messages received.

2.8.3 Breaking Down the Subscriber Node Code

1. Shebang and Imports

```

#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

```

- The shebang ensures the script runs with Python 3.
- `rclpy` is the ROS 2 Python client library.
- `Node` is the base class for creating ROS 2 nodes.
- `String` is the message type imported from the standard messages package.

2. Defining the Node Class

```

class PySubscriber(Node):

    def __init__(self):
        super().__init__('py_subscriber')
        self.get_logger().info("Python Subscriber node has been started")
        self.subscription = self.create_subscription(
            String,
            'cpp_topic',
            self.listener_callback,
            10)
        self.subscription # prevent unused variable warning

```

- The `PySubscriber` class inherits from `Node` to create a new ROS 2 node.
- The node is initialized with the name `py_subscriber`.



- `self.get_logger().info("Python Subscriber node has been started")` logs an informational message when the node starts.
- `self.create_subscription` creates a subscription to the topic `publisher_topic` for messages of type `String` with a queue size of 10.
- `self.listener_callback` is set as the callback function to be invoked upon receiving a message.

3. Listener Callback Method

```
def listener_callback(self, msg):  
    self.get_logger().info('I heard: "%s"' % msg.data)
```

- `listener_callback` is executed every time a message is received.
- An informational log displays the data of the received message.

4. The Main Function

```
def main(args=None):  
    rclpy.init(args=args)  
  
    py_subscriber = PySubscriber()  
  
    rclpy.spin(py_subscriber)  
  
    # Destroy the node explicitly  
    # (optional - otherwise it will be done automatically  
    # when the garbage collector destroys the node object)  
    py_subscriber.destroy_node()  
    rclpy.shutdown()  
  
if __name__ == '__main__':  
    main()
```

- `rclpy.init(args=args)` initializes ROS 2 communication by parsing command-line arguments and setting up necessary middleware.
- An instance of `PySubscriber` node is created.
- `rclpy.spin(py_subscriber)` keeps the node active, processing incoming messages until shutdown.
- Finally, the node is destroyed and `rclpy.shutdown()` cleans up resources.

This detailed breakdown explains each component of the ROS 2 Python subscriber node. For further details and additional context, refer to the [Creating ROS 2 Nodes \(Python\)](#) tutorial.

2.8.4 Running the Subscriber Node Standalone (Optional)

After editing the file, you can run your Python subscriber node out of ROS 2 environment. For this, open a new terminal session (or use an additional session in Terminator) and navigate to the location of the Python file:

```
$ cd src/my_py_package/my_py_package
```

Then, test the Python file by running it with one of the following commands:



```
$ ./subscriber_py.py
```

to run it as an executable, or

```
$ python3 subscriber_py.py
```

to run it as a Python script.

2.8.5 Integrating the Subscriber Node into the Package

Modifying `setup.py`

To allow ROS 2 to recognize and execute your subscriber node, specify it in the `setup.py` file of your package. Open `setup.py` in VS Code and locate the `entry_points` field. Then, add your node entry immediately after any existing ROS 2 node entries (e.g., `node_publisher_py`):

```
entry_points={
    'console_scripts': [
        'node_publisher_py = my_py_package.publisher_py:main',
        'node_subscriber_py = my_py_package.subscriber_py:main',
    ],
}
```

Breaking down the components of this specification:

- `node_subscriber_py`: Specifies the command-line executable name that you will use to run the subscriber node.
- `my_py_package`: Indicates the name of your package.
- `subscriber_py`: Specifies the Python file (without the `.py` extension) that contains the node.
- `main`: Refers to the function that will be executed when the node is run.

Building the Package

After updating `setup.py`, build your package. In the first terminal session, from the root of the workspace, compile with:

```
$ colcon build
```

Alternatively, to build only the Python package:

```
$ colcon build --packages-select my_py_package
```

For development, you might consider using the `--symlink-install` option so that changes in the source files are immediately reflected without requiring a full rebuild:

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

or, more specifically:

```
$ colcon build --packages-select my_py_package --symlink-install
```

If the build completes successfully, you will see the following confirmation message: `'colcon build' successful`.



Sourcing the Environment

Before running the node, in the first (or current) terminal session from the root of the workspace, source the workspace's setup file to load the necessary environment variables:

```
$ source ./install/setup.bash
```

This step ensures that ROS 2 and the built packages are properly configured.

2.8.6 Running the ROS 2 Subscriber Node with ROS 2

Execute your subscriber node using the `ros2 run` command:

```
$ ros2 run my_py_package node_subscriber_py
```

At this point, if your publisher node is running, your subscriber node should be actively receiving and logging messages published to the topic `publisher_topic`.

2.8.7 Summary of Key Identifiers

It is important to distinguish between the following identifiers in your ROS 2 subscriber node:

1. `subscriber_py.py`: The filename of your Python script.
2. `py_subscriber`: The name of the node as defined in the `super().__init__()` call within your script.
3. `node_subscriber_py`: The command-line executable name specified in `setup.py` under the `entry_points` field.

Ensuring that these identifiers are correctly set and consistently used is crucial for the proper functioning of your ROS 2 Python nodes.

For more details on creating ROS 2 nodes in Python, refer to the official [Creating ROS 2 Nodes \(Python\)](#) tutorial.

2.9 Practice Assignment: Running ROS 2 Nodes

In this assignment, you will run ROS 2 publisher and subscriber nodes implemented in both C++ and Python, as described in Sections 2.5, 2.6, 2.7, and 2.8. You will then capture evidence of successful execution using a screenshot and submit it in the designated assignment area on Canvas (within the ROS 2 module) as a PNG file. The filename must follow this format:

`FirstnameLastname_evidence_wXsY.png`,

where `wX` and `sY` indicate the corresponding week and session numbers. For example, for week 1, session 2, the correct filename would be:

`EduardoDavila_evidence_w1s2.png`.



2.9.1 Executing ROS 2 Publishers and Subscribers

Running the Python Publisher and Subscriber

After implementing and saving your Python nodes and updating `package.xml` and `setup.py` files, build your package and run the nodes. In two separate terminal sessions (or using the **Terminator** emulator), execute the following commands from the root of your workspace:

- In **Terminal 1**, build your package, source the setup file, and run the Python publisher node:

```
$ colcon build --packages-select my_py_package
$ source install/setup.bash
$ ros2 run my_py_package node_publisher_py
```

- In **Terminal 2**, source the setup file and run the Python subscriber node:

```
$ source install/setup.bash
$ ros2 run my_py_package node_subscriber_py
```

You should observe that the `py_publisher` node sends a personalized message (e.g., displaying your name), and the `py_subscriber` node awaits the reception of the messages.

Running the C++ Publisher and Subscriber

After implementing and saving your C++ nodes and updating `package.xml` and `CMakeLists.txt` files, build your package and run the nodes. In two additional terminal sessions (or another two sessions in **Terminator**), execute the following commands from the root of your workspace:

- In **Terminal 3**, build your package, source the setup file, and run the C++ publisher node:

```
$ colcon build --packages-select my_cpp_package
$ source install/setup.bash
$ ros2 run my_cpp_package node_publisher_cpp
```

- In **Terminal 4**, source the setup file and run the C++ subscriber node:

```
$ source install/setup.bash
$ ros2 run my_cpp_package node_subscriber_cpp
```

Here, you should observe that the `py_publisher` node publishes your personalized message (e.g., displaying your name), whereas the `cpp_subscriber` node confirms its reception. Additionally, the `cpp_publisher` node publishes your other personalized message (e.g., displaying your name), whereas the `py_subscriber` node confirms its reception.

Visualizing the ROS Graph with `rqt_graph`

To ensure that your publisher and subscriber nodes are correctly connected, use the `rqt_graph` tool. Then, in **Terminal 5**, execute the following command (without the need to enter the workspace directory):

```
$ ros2 run rqt_graph rqt_graph
```



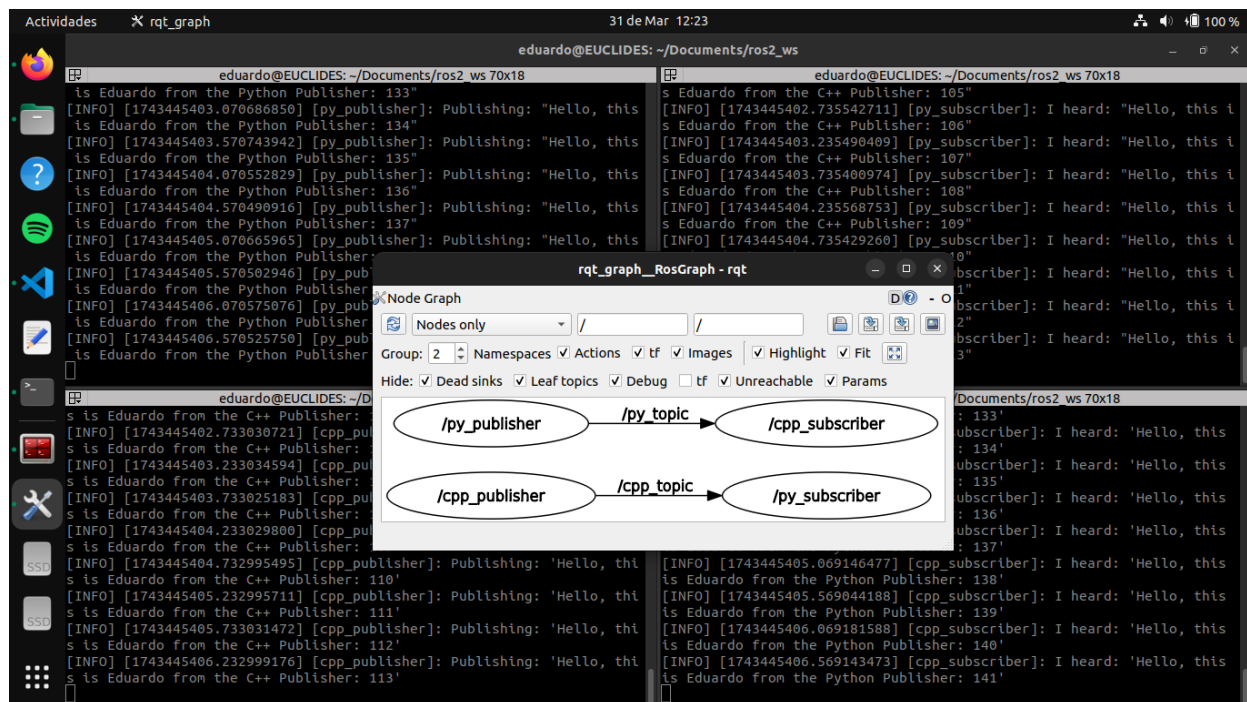


Figure 2.1: Example screenshot showing terminal outputs from the `py_publisher`, `py_subscriber`, `cpp_publisher`, and `cpp_subscriber` nodes in action.

or simply:

```
$ rqt_graph
```

This visualization should show the topics connecting your publisher and subscriber nodes as described above.

2.9.2 Submission Instructions

1. Run the ROS 2 nodes as described above.
2. Capture a **screenshot showing the terminal outputs** from all four nodes (Python and C++ publishers and subscribers) along with the visualization of the ROS graph using the `rqt_graph` tool. See Figure 2.1 for reference.
3. Save the screenshot as a PNG file.
4. Name the file following this format: `FirstnameLastname_evidence_wXsY.png` (replace `X` with the week number and `Y` with the session number).
5. Submit your file as instructed by the course guidelines on Canvas.

Note: Your machine's username and device name must be visible in the screenshot to verify the authenticity of the submission, as shown in Figure 2.1. Any submission that appears copied, unclear, or altered will be considered invalid and may result in a score of 0 for this assignment.



CHAPTER

3

ROS 2 Service and Client

Author: Dr. Eduardo de Jesús Dávila Meza.

 [EduardoDavila-AI-PhD](#)

✿ **Abstract** - This chapter introduces ROS 2 services and clients. Unlike topics, which support one-to-many communication, services allow for a synchronous or asynchronous request-response interaction between nodes. A service consists of two parts: a [server](#), which provides a functionality, and a [client](#), which calls that functionality.

3.1 Introduction to ROS 2 Services and Clients

3.1.1 Concepts: Service and Client

Service/Server: A ROS 2 service provides a mechanism for a node to offer a specific functionality that other nodes can call. The service operates on a request/response model, where the server waits for a request and then sends back a response.

Client: A ROS 2 client calls a service provided by a server. This interaction can be synchronous, meaning the client blocks until the response is received (or a timeout occurs). However, in our examples, we use asynchronous service call APIs: `async_send_request()` in C++ and `call_async()` in Python. These asynchronous APIs allow the client node to continue processing other tasks while waiting for the service response, which is generally recommended over synchronous calls. (For more details, refer to the guide on [Synchronous vs. Asynchronous Clients](#).)

3.1.2 Prerequisites

Before you begin, ensure that you are familiar with workspace and package creation, as described in Sections [2.2](#), [2.3](#), and [2.4](#). Open a terminal and navigate to the `src` directory of your ROS 2

workspace (e.g., `~/ros2_ws`):

```
$ cd ros2_ws/src
```

Replace `ros2_ws` with the actual name of your workspace.

Next, create the following packages in your workspace:

- C++ package for the service and client nodes:

```
$ ros2 pkg create s3_cpp_srvcli --build-type ament_cmake --dependencies rclcpp  
→ example_interfaces --license Apache-2.0 --description "Your package description  
→ here"
```

- Python package for the service and client nodes:

```
$ ros2 pkg create s3_py_srvcli --build-type ament_python --dependencies rclpy  
→ example_interfaces --license Apache-2.0 --description "Your package description  
→ here"
```

You may replace `s3_cpp_srvcli` and `s3_py_srvcli` with package names of your preference.

3.2 Creating a ROS 2 Service Node in C++

3.2.1 Setting Up the C++ Service Node

Open a terminal and navigate to the `src` directory of your C++ package (e.g., `s3_cpp_srvcli`). Then, create a new C++ file for the service node, (e.g., `server.cpp`):

```
$ cd ros2_ws/src/s3_cpp_srvcli/src  
$ touch server.cpp
```

3.2.2 Editing the C++ Service Node

Open your workspace in VS Code:

```
$ cd ~/ros2_ws  
$ code .
```

In VS Code, locate and open `server.cpp` and implement your service node code (code to be provided later).

Implementing the C++ Service Node Code

Below is an example implementation of a simple ROS 2 service node in C++:



```

#include "rclcpp/rclcpp.hpp"
#include "example_interfaces/srv/add_two_ints.hpp"

class CppService : public rclcpp::Node
{
public:
    CppService()
    : Node("cpp_server_node")
    {
        RCLCPP_INFO(this->get_logger(), "C++ Server node has been started");
        service_ = this->create_service<example_interfaces::srv::AddTwoInts>(
            "cpp_add_two_ints_service",
            std::bind(&CppService::AddTwoInts_callback, this, std::placeholders::_1,
                std::placeholders::_2)
        );
        RCLCPP_INFO(this->get_logger(), "Service 'cpp_add_two_ints_service' is ready
        to receive requests");
    }

private:
    void AddTwoInts_callback(
        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(this->get_logger(), "Received request:\na: %ld, b: %ld",
            request->a, request->b);
        RCLCPP_INFO(this->get_logger(), "Sending back response: sum = %ld",
            response->sum);
    }

    rclcpp::Service<example_interfaces::srv::AddTwoInts>::SharedPtr service_;
};

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);
    std::shared_ptr<rclcpp::Node> node = std::make_shared<CppService>();
    rclcpp::spin(node);
    rclcpp::shutdown();
    return 0;
}

```

This code defines a server node that offers a service to add two integers using the service name `cpp_add_two_ints_service`.

Breaking Down the Service Node Code

We now examine each part of the code in detail to understand how the C++ service node works.

1. Includes and Setup

```

#include "rclcpp/rclcpp.hpp"
#include "example_interfaces/srv/add_two_ints.hpp"

```



- `rclcpp/rclcpp.hpp` provides the ROS 2 C++ API.
- `example_interfaces/srv/add_two_ints.hpp` defines the `AddTwoInts` service message type.

2. Defining the Node Class

```
class CppService : public rclcpp::Node
{
public:
    CppService()
    : Node("cpp_server_node")
    {
        RCLCPP_INFO(this->get_logger(), "C++ Server node has been started");
        service_ = this->create_service<example_interfaces::srv::AddTwoInts>(
            "cpp_add_two_ints_service",
            std::bind(&CppService::AddTwoInts_callback, this, std::placeholders::_1,
                std::placeholders::_2)
        );
        RCLCPP_INFO(this->get_logger(), "Service 'cpp_add_two_ints_service' is ready
        to receive requests");
    }
};
```

- The `CppService` class inherits from `rclcpp::Node`, creating a new ROS 2 node.
- **public:** This access specifier indicates that all members declared after it (until another access specifier is encountered) are publicly accessible.
- The `CppService()` constructor initializes the node with the name `"cpp_server_node"`.
- `RCLCPP_INFO(this->get_logger(), "C++ Server node has been started")` logs an informational message when the node starts.
- The node registers a service named `"cpp_add_two_ints_service"` using the `AddTwoInts` service type. The callback function, `AddTwoInts_callback`, is bound to handle incoming requests using `std::bind()`, ensuring that the correct instance and parameters are forwarded. Additionally, `std::placeholders::_1` and `_2` are used to represent the request and response objects passed to the callback at runtime.
- `RCLCPP_INFO(this->get_logger(), "Service 'cpp_add_two_ints_service' is ready to receive requests")` logs an informational message to indicate service is ready.

3. Request Callback Method

```
private:
    void AddTwoInts_callback(
        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(this->get_logger(), "Received request:\na: %ld, b: %ld",
        request->a, request->b);
        RCLCPP_INFO(this->get_logger(), "Sending back response: sum = %ld",
        response->sum);
    }

    rclcpp::Service<example_interfaces::srv::AddTwoInts>::SharedPtr service_;
};
```



- **private:** This access specifier indicates that all members declared after it are accessible only within the class itself.
- The **AddTwoInts_callback** method is invoked when the service receives a request. It calculates the sum of **a** and **b**, logs the request and the response, and then sends back the result.
- **service_:** A shared pointer to the service object that handles incoming requests.

4. The Main Function

```
int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);
    std::shared_ptr<rclcpp::Node> node = std::make_shared<CppService>();
    rclcpp::spin(node);
    rclcpp::shutdown();
    return 0;
}
```

- **rclcpp::init(argc, argv)** initializes ROS 2 communication by parsing command-line arguments and setting up the necessary middleware.
- **node = std::make_shared<CppService>()** creates an instance of the **CppService** node.
- **rclcpp::spin(node)** keeps the node active, processing callbacks (incoming requests) until a shutdown signal is received.
- **rclcpp::shutdown()** gracefully shuts down ROS 2, releasing all allocated resources.
- **return 0;** indicates successful termination of the program.

This detailed breakdown explains each component of the ROS 2 C++ service node. For further details, refer to the [Creating a ROS 2 Service \(C++\)](#) tutorial.

3.2.3 Integrating the C++ Service Node into the Package

Modify **CMakeLists.txt** of your C++ package to add an executable for the service node:

```
# Find dependencies
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(example_interfaces REQUIRED)

add_executable(cpp_server_exe src/server.cpp)
ament_target_dependencies(cpp_server_exe rclcpp example_interfaces)

install(TARGETS
  cpp_server_exe
  DESTINATION lib/${PROJECT_NAME}
)
```

3.2.4 Building, Sourcing, and Running the C++ Service Node

From the root of your workspace, build the package:

```
$ colcon build --packages-select s3_cpp_srvcli
```



Then, source the workspace:

```
$ source install/setup.bash
```

Finally, run the service server node:

```
$ ros2 run s3_cpp_srvcli cpp_server_exe
```

If everything is set up correctly, your node should be running and ready to receive requests.

3.2.5 Testing the C++ Service

Open a new terminal, test the service using the following command:

```
$ ros2 service call /cpp_add_two_ints_service example_interfaces/srv/AddTwoInts "{a: 2,  
→ b: 3}"
```

Alternatively, use `rqt_service_caller`:

```
$ ros2 run rqt_service_caller rqt_service_caller
```

and request the service on `/cpp_add_two_ints_service` by entering the values `a: 5, b: 7`.

3.2.6 Summary of C++ Service Node Key Identifiers

- `server.cpp`: Name of the source file for the C++ service node.
- `cpp_server_node`: Name of the C++ service node as defined in the constructor of the C++ class.
- `cpp_server_exe`: Name of the executable defined in `CMakeLists.txt`.
- `/cpp_add_two_ints_service`: Name of the service provided by the C++ server node.

3.3 Creating a ROS 2 Client Node in C++

3.3.1 Setting Up the C++ Client Node

Open a terminal and navigate to the `src` directory of your C++ package (e.g., `s3_cpp_srvcli`). Then, create a new C++ file for the client node, (e.g., `client.cpp`):

```
$ cd ros2_ws/src/s3_cpp_srvcli/src  
$ touch client.cpp
```

3.3.2 Editing the C++ Client Node

Open your workspace in VS Code:

```
$ cd ~/ros2_ws  
$ code .
```

In VS Code, locate and open `client.cpp` and implement your client node code (code to be provided later).



Implementing the C++ Client Node Code

Below is an example implementation of a simple ROS 2 client node in C++ that uses the asynchronous service call API (`async_send_request()`).

```
#include "rclcpp/rclcpp.hpp"
#include "example_interfaces/srv/add_two_ints.hpp"

using namespace std::chrono_literals;

class CppClientAsync : public rclcpp::Node
{
public:
    CppClientAsync(int64_t a, int64_t b)
        : Node("cpp_client_async_node"), a_(a), b_(b)
    {
        RCLCPP_INFO(this->get_logger(), "C++ Client node has been started");
        client_ =
        ↪ this->create_client<example_interfaces::srv::AddTwoInts>("cpp_add_two_ints_service");

        while (!client_->wait_for_service(1s))
        {
            if (!rclcpp::ok())
            {
                RCLCPP_ERROR(this->get_logger(), "Interrupted while waiting for the
        ↪ service. Exiting.");
                return;
            }
            RCLCPP_INFO(this->get_logger(), "Waiting for service to become
        ↪ available...");
        }
        send_request();
    }

private:
    void send_request()
    {
        std::shared_ptr<example_interfaces::srv::AddTwoInts::Request> request =
        ↪ std::make_shared<example_interfaces::srv::AddTwoInts::Request>();
        request->a = a_;
        request->b = b_;

        // Asynchronously send the request
        rclcpp::Client<example_interfaces::srv::AddTwoInts>::FutureAndRequestId
        ↪ future_and_request_id = client_->async_send_request(request);
        std::shared_future<example_interfaces::srv::AddTwoInts::Response::SharedPtr>
        ↪ result = future_and_request_id.future.share();
    }
};
```



```

        // Spin until the future is complete
        if (rclcpp::spin_until_future_complete(this->get_node_base_interface(),
→ result) ==
            rclcpp::FutureReturnCode::SUCCESS)
        {
            RCLCPP_INFO(this->get_logger(), "Received response: %ld + %ld = %ld",
→ request->a, request->b, result.get()->sum);
        }
        else
        {
            RCLCPP_ERROR(this->get_logger(), "Failed to call service add_two_ints");
        }
    }

    rclcpp::Client<example_interfaces::srv::AddTwoInts>::SharedPtr client_;
    int64_t a_;
    int64_t b_;
};

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);

    if (argc != 3)
    {
        RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Usage: cpp_client_exe a b");
        return 1;
    }

    std::shared_ptr<rclcpp::Node> node = std::make_shared<CppClientAsync>(atoll(argv[1]),
→ atoll(argv[2]));
    rclcpp::shutdown();
    return 0;
}

```

This code defines a client node that asynchronously requests the service `cpp_add_two_ints_service`, using `async_send_request()`, to add two integers.

Breaking Down the Client Node Code

We now examine each part of the code in detail to understand how the ROS 2 C++ client node works.

1. Includes and Setup

```
#include "rclcpp/rclcpp.hpp"
#include "example_interfaces/srv/add_two_ints.hpp"
```

- `rclcpp/rclcpp.hpp` provides the ROS 2 C++ API.
- `example_interfaces/srv/add_two_ints.hpp` defines the `AddTwoInts` service message type.

2. Defining the Node Class

```
class CppClientAsync : public rclcpp::Node
{
public:
    CppClientAsync(int64_t a, int64_t b)
        : Node("cpp_client_async_node"), a_(a), b_(b)
    {
        RCLCPP_INFO(this->get_logger(), "C++ Client node has been started");
        client_ =
        ↪ this->create_client<example_interfaces::srv::AddTwoInts>("cpp_add_two_ints_service");
        // Wait for the service to become available
        while (!client_->wait_for_service(1s))
        {
            if (!rclcpp::ok())
            {
                RCLCPP_ERROR(this->get_logger(), "Interrupted while waiting for the
        ↪ service. Exiting.");
                return;
            }
            RCLCPP_INFO(this->get_logger(), "Waiting for service to become
        ↪ available...");
        }
        send_request();
    }
}
```

- The `CppClientAsync` class inherits from `rclcpp::Node`, creating a new ROS 2 node.
- `public`: This access specifier indicates that all members declared after it (until another access specifier is encountered) are publicly accessible.
- The `CppClientAsync()` constructor initializes the node with the name `"cpp_client_async_node"` and stores the input integers `a` and `b`.
- `RCLCPP_INFO(this->get_logger(), "C++ Client node has been started")` logs an informational message when the node starts.
- The node creates a client for the service `"cpp_add_two_ints_service"`.
- A loop waits for the service to become available, logging status messages. If the ROS system is interrupted during the wait, an error is logged and the function returns.
- Once the service is available, the `send_request()` function is called.

3. Sending the Service Request

```
private:
    void send_request()
    {
        std::shared_ptr<example_interfaces::srv::AddTwoInts::Request> request =
→ std::make_shared<example_interfaces::srv::AddTwoInts::Request>();
        request->a = a_;
        request->b = b_;
        rclcpp::Client<example_interfaces::srv::AddTwoInts>::FutureAndRequestId
→ future_and_request_id = client_->async_send_request(request);
        std::shared_future<example_interfaces::srv::AddTwoInts::Response::SharedPtr>
→ result = future_and_request_id.future.share();

        if (rclcpp::spin_until_future_complete(this->get_node_base_interface(),
→ result) ==
            rclcpp::FutureReturnCode::SUCCESS)
        {
            RCLCPP_INFO(this->get_logger(), "Received response: %ld + %ld = %ld",
→ request->a, request->b, result.get()->sum);
        }
        else
        {
            RCLCPP_ERROR(this->get_logger(), "Failed to call service add_two_ints");
        }
    }

    rclcpp::Client<example_interfaces::srv::AddTwoInts>::SharedPtr client_;
    int64_t a_;
    int64_t b_;
};
```

- **private:** This access specifier indicates that all members declared after it are accessible only within the class itself.
- The **send_request()** method creates a request, assigns values to its fields, and sends it asynchronously. It waits until the future completes and then logs the received response.
- **request->a** and **request->b** store the values to be added, taken from the class member variables.
- **async_send_request(request)** sends the request asynchronously and returns a **FutureAndRequestId** object, which contains both the future result and the internal request ID.
- **future.share()** transforms the unique future into a shared future, allowing multiple accesses if needed.
- **spin_until_future_complete()** blocks the current node until the future is completed or a timeout/error occurs. Although the request is sent asynchronously, this line synchronously waits for the response.
- **result.get()->sum** accesses the result of the service call once the future has completed successfully.
- **RCLCPP_INFO()** and **RCLCPP_ERROR()** are used to print informational or error messages to the console, including the input data and the result.
- **client_**, **a_**, and **b_**: Class members used to send the request and store the request parameters.



4. The Main Function

```
int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);
    if (argc != 3)
    {
        RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Usage: cpp_client_exe a b");
        return 1;
    }
    std::shared_ptr<rclcpp::Node> node = std::make_shared<CppClientAsync>(atoll(argv[1]),
    atoll(argv[2]));
    rclcpp::shutdown();
    return 0;
}
```

- `rclcpp::init(argc, argv)` initializes ROS 2 communication by parsing command-line arguments and setting up the necessary middleware.
- `argc != 3` ensures that exactly two numeric arguments are passed from the command line. If not, a usage message is printed and the program exits.
- `node = std::make_shared<CppClientAsync>()` creates an instance of the `CppClientAsync` node, passing the two command-line arguments as integers.
- After execution, `rclcpp::shutdown()` gracefully shuts down ROS 2, releasing all allocated resources.
- `return 0;` indicates successful termination of the program.

This detailed breakdown explains each component of the ROS 2 C++ client node. For further details, refer to the [Creating a ROS 2 Client \(C++\)](#) tutorial.

3.3.3 Integrating the C++ Client Node into the Package

Modify `CMakeLists.txt` of your C++ package to add an executable for the client node while preserving any existing node setup (e.g., `cpp_server_exe`):

```
# Find dependencies
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(example_interfaces REQUIRED)

add_executable(cpp_server_exe src/server.cpp)
ament_target_dependencies(cpp_server_exe rclcpp example_interfaces)

add_executable(cpp_client_exe src/client.cpp)
ament_target_dependencies(cpp_client_exe rclcpp example_interfaces)

install(TARGETS
  cpp_server_exe
  cpp_client_exe
  DESTINATION lib/${PROJECT_NAME})
```



3.3.4 Building, Sourcing, and Running the C++ Client Node

From the root of your workspace, build the package:

```
$ colcon build --packages-select s3_cpp_srvcli
```

Then, source the workspace:

```
$ source install/setup.bash
```

Finally, run the client node:

```
$ ros2 run s3_cpp_srvcli cpp_client_exe 11 13
```

If everything is set up correctly, your node should execute, send a request to the service, and wait for the response. If the service is not available, the client will wait until it becomes accessible before proceeding.

3.3.5 Summary of C++ Client Node Key Identifiers

- **client.cpp**: Name of the source file for the C++ client node.
- **cpp_client_async_node**: Name of the C++ client node as defined in the constructor of the C++ class.
- **cpp_client_exe**: Name of the executable defined in **CMakeLists.txt**.
- **/cpp_add_two_ints_service**: Name of the service that the C++ client node calls.

3.4 Creating a ROS 2 Service Node in Python

3.4.1 Setting Up the Python Service Node

Open a terminal and navigate to the **src** directory of your Python package (e.g., **s3_py_srvcli**). Then, create a new Python file for the service node (e.g., **server.py**):

```
$ cd ros2_ws/src/s3_py_srvcli/s3_py_srvcli
$ touch server.py
```

3.4.2 Editing the Python Service Node

Open your workspace in VS Code:

```
$ cd ~/ros2_ws
$ code .
```

In VS Code, locate and open **server.py** and implement your service node code (code to be provided later).

Implementing the Python Service Node Code

Below is an example implementation of a simple ROS 2 service node in Python:



```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from example_interfaces.srv import AddTwoInts

class PyService(Node):

    def __init__(self):
        super().__init__('py_server_node')
        self.get_logger().info("Python Server node has been started")
        self.srv = self.create_service(AddTwoInts, 'py_add_two_ints_service',
        → self.AddTwoInts_callback)
        self.get_logger().info("Service 'py_add_two_ints_service' is ready to receive
        → requests")

    def AddTwoInts_callback(self, request, response):
        response.sum = request.a + request.b
        self.get_logger().info('Received request:\na: %d, b: %d' % (request.a,
        → request.b))
        self.get_logger().info('Sending back response: sum = %d' % (response.sum))
        return response

def main(args=None):
    rclpy.init(args=args)
    py_service = PyService()
    rclpy.spin(py_service)
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

This code defines a server node that offers a service to add two integers using the service name `py_add_two_ints_service`.

Breaking Down the Service Node Code

We now examine each part of the code in detail to understand how the Python service node works.

1. Imports and Initialization

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from example_interfaces.srv import AddTwoInts
```

- The shebang line (`#!/usr/bin/env python3`) ensures that the script is executed with Python 3.
- `rclpy` provides the ROS 2 Python API.
- `Node` is the base class for creating ROS 2 nodes.
- `AddTwoInts` is the service message type imported from the `example_interfaces` package.



2. Defining the Node Class

```
class PyService(Node):

    def __init__(self):
        super().__init__('py_server_node')
        self.get_logger().info("Python Server node has been started")
        self.srv = self.create_service(AddTwoInts, 'py_add_two_ints_service',
        ↪ self.AddTwoInts_callback)
        self.get_logger().info("Service 'py_add_two_ints_service' is ready to receive
        ↪ requests")
```

- The `PyService` class inherits from `Node`, creating a new ROS 2 node.
- In the `__init__` method:
 - `super().__init__()` initializes the node with the name `py_server_node`.
 - `self.get_logger().info("Python Server node has been started")` logs an informational message when the node starts.
 - `self.create_service()` creates a service using the `AddTwoInts` service type with the service name `"py_add_two_ints_service"`. The callback function, `AddTwoInts_callback`, is set to handle incoming requests.
 - `self.get_logger().info("Service 'py_add_two_ints_service' is ready to receive requests")` logs an informational message to indicate service is ready.

3. Request Callback Method

```
def AddTwoInts_callback(self, request, response):
    response.sum = request.a + request.b
    self.get_logger().info('Received request:\na: %d, b: %d' % (request.a,
    ↪ request.b))
    self.get_logger().info('Sending back response: sum = %d' % (response.sum))
    return response
```

- The `AddTwoInts_callback` method is invoked when the service receives a request.
- It calculates the sum of `a` and `b`, logs the request and response, and returns the response.

4. The Main Function

```
def main(args=None):
    rclpy.init(args=args)
    py_service = PyService()
    rclpy.spin(py_service)
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

- `rclpy.init(args=args)` initializes ROS 2 communication by parsing command-line arguments and setting up the necessary middleware.
- An instance of the `PyService` node is created.



- `rcipy.spin(py_service)` keeps the node active, processing callbacks (incoming requests) until a shutdown signal is received.
- `rcipy.shutdown()` gracefully shuts down ROS 2, releasing all allocated resources.

This detailed breakdown explains each component of the ROS 2 Python service node. For further details, refer to the [Creating a ROS 2 Service \(Python\)](#) tutorial.

3.4.3 Integrating the Python Service Node into the Package

Edit the `setup.py` file of your Python package to add an entry point for the service node:

```
entry_points={
    'console_scripts': [
        'py_server_exe = s3_py_srvcli.server:main',
    ],
},
```

3.4.4 Building, Sourcing, and Running the Python Service Node

From the root of your workspace, build the package:

```
$ colcon build --packages-select s3_py_srvcli
```

Then, source the workspace:

```
$ source install/setup.bash
```

Finally, run the service node:

```
$ ros2 run s3_py_srvcli py_server_exe
```

If everything is set up correctly, your node should be running and ready to receive requests.

3.4.5 Testing the Python Service

Open a new terminal, test the service using the following command:

```
$ ros2 service call /py_add_two_ints_srv example_interfaces/srv/AddTwoInts "{a: -2, b: -3}"
```

Alternatively, use `rqt_service_caller`:

```
$ ros2 run rqt_service_caller rqt_service_caller
```

and request the service on `/py_add_two_ints_service` with values `a: -5, b: -7`.

3.4.6 Summary of Python Service Node Key Identifiers

- `server.py`: Name of the source file for the Python service node.
- `py_server_node`: Name of the Python service node as defined in the constructor of the Python class.
- `py_server_exe`: Name of executable defined in `setup.py`.
- `/py_add_two_ints_service`: The name of the service provided by the Python server node.



3.5 Creating a ROS 2 Client Node in Python

3.5.1 Setting Up the Python Client Node

Open a terminal and navigate to the `src` directory of your Python package (e.g., `s3_py_srvcli`). Then, create a new Python file for the service node (e.g., `client.py`):

```
$ cd ros2_ws/src/s3_py_srvcli/s3_py_srvcli
$ touch client.py
```

3.5.2 Editing the Python Client Node

Open your workspace in VS Code:

```
$ cd ~/ros2_ws
$ code .
```

In VS Code, locate and open `client.py` and implement your client node code (code to be provided later).

Implementing the Python Client Node Code

Below is an example implementation of a simple ROS 2 client node in Python that uses the asynchronous service call API (`call_async()`).

```
#!/usr/bin/env python3
import sys
import rclpy
from rclpy.node import Node
from example_interfaces.srv import AddTwoInts

class PyClientAsync(Node):

    def __init__(self):
        super().__init__('py_client_async_node')
        self.get_logger().info("Python Client node has been started")
        self.client = self.create_client(AddTwoInts, 'py_add_two_ints_service')

        # Wait for the service to become available
        while not self.client.wait_for_service(timeout_sec=1.0):
            if not rclpy.ok():
                self.get_logger().error('Interrupted while waiting for the service.
                ↪ Exiting.')
            return
        self.get_logger().info('Waiting for service to become available...')

        # Prepare a persistent request object
        self.req = AddTwoInts.Request()
```




```

def send_request(self, a, b):
    self.req.a = a
    self.req.b = b
    return self.client.call_async(self.req)

def main(args=None):
    rclpy.init(args=args)
    if len(sys.argv) != 3:
        print("Usage: py_client_exe a b")
        sys.exit(1)
    py_client = PyClientAsync()
    future = py_client.send_request(int(sys.argv[1]), int(sys.argv[2]))

    # Spin until the future is complete
    rclpy.spin_until_future_complete(py_client, future)
    result = future.result()
    if result is not None:
        py_client.get_logger().info(
            'Received response: %d + %d = %d' %
            (int(sys.argv[1]), int(sys.argv[2]), result.sum)
        )
    else:
        py_client.get_logger().error('Failed to call service add_two_ints')
    py_client.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()

```

This code defines a client node that asynchronously calls the service `py_add_two_ints_service`, using `call_async()`, to add two integers.

Breaking Down the Client Node Code

We now examine each part of the code in detail to understand how the ROS 2 Python client node works.

1. Imports and Initialization

```

#!/usr/bin/env python3
import sys
import rclpy
from rclpy.node import Node
from example_interfaces.srv import AddTwoInts

```

- The shebang line (`#!/usr/bin/env python3`) ensures that the script is executed with Python 3.
- `rclpy` provides the ROS 2 Python API.
- `Node` is the base class for creating ROS 2 nodes.
- `AddTwoInts` is the service message type imported from the `example_interfaces` package.



2. Defining the Node Class

```
class PyClientAsync(Node):

    def __init__(self):
        super().__init__('py_client_async_node')
        self.get_logger().info("Python Client node has been started")
        self.client = self.create_client(AddTwoInts, 'py_add_two_ints_service')

        # Wait for the service to become available
        while not self.client.wait_for_service(timeout_sec=1.0):
            if not rclpy.ok():
                self.get_logger().error('Interrupted while waiting for the service.
                ↪ Exiting.')
                return
            self.get_logger().info('Waiting for service to become available...')

        # Prepare a persistent request object
        self.req = AddTwoInts.Request()
```

- The `PyClientAsync` class inherits from `Node`, creating a new ROS 2 node.
- In the `__init__` method:
 - `super().__init__()` initializes the node with the name `py_client_async_node`.
 - `self.get_logger().info("Python Client node has been started")` logs an informational message when the node starts.
 - `self.create_client()` creates a client to call the `AddTwoInts` service type under the name `py_add_two_ints_service`.
 - A loop waits for the service to become available, logging status messages. If the ROS system is interrupted during the wait, an error is logged and the function returns.
 - Once the service is available, a persistent request object is initialized with `AddTwoInts.Request()`.

3. Sending the Service Request

```
def send_request(self, a, b):
    self.req.a = a
    self.req.b = b
    return self.client.call_async(self.req)
```

- The `send_request` method assigns the integers `a` and `b` to the corresponding fields of the previously created request object.
- Then, it uses `call_async()` to initiate a non-blocking service call, returning a future object that will eventually contain the result.
- This approach allows the node to continue running other tasks while waiting for the service response.



4. The Main Function

```
def main(args=None):
    rclpy.init(args=args)
    if len(sys.argv) != 3:
        print("Usage: py_client_exe a b")
        sys.exit(1)
    py_client = PyClientAsync()
    future = py_client.send_request(int(sys.argv[1]), int(sys.argv[2]))

    # Spin until the future is complete
    rclpy.spin_until_future_complete(py_client, future)
    result = future.result()
    if result is not None:
        py_client.get_logger().info(
            'Received response: %d + %d = %d' %
            (int(sys.argv[1]), int(sys.argv[2]), result.sum)
        )
    else:
        py_client.get_logger().error('Failed to call service add_two_ints')
    py_client.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

- `rclpy.init(args=args)` initializes ROS 2 communication by parsing command-line arguments and setting up the necessary middleware.
- `len(sys.argv) != 3` ensures that exactly two numeric arguments are passed from the command line. If not, a usage message is printed and the program exits.
- An instance of the `PyClientAsync` node is created.
- `rclpy.spin_until_future_complete()` processes callbacks until the service response is received, and the result is then logged.
- After execution, `rclpy.shutdown()` gracefully shuts down ROS 2, releasing all allocated resources.

This detailed breakdown explains each component of the ROS 2 Python client node. For further details, refer to the [Creating a ROS 2 Client \(Python\)](#) tutorial.

3.5.3 Integrating the Python Client Node into the Package

Edit the `setup.py` file of your Python package to add an entry point for the client node while preserving any existing node setup (e.g., `py_server_exe`):

```
entry_points={
    'console_scripts': [
        'py_server_exe = s3_py_srvcli.server:main',
        'py_client_exe = s3_py_srvcli.client:main',
    ],
}
```



3.5.4 Building, Sourcing, and Running the Python Client Node

From the root of your workspace, build the package:

```
$ colcon build --packages-select s3_py_srvcli
```

Then, source the workspace:

```
$ source install/setup.bash
```

Finally, run the client node:

```
$ ros2 run s3_py_srvcli py_client_exe -11 -13
```

If everything is set up correctly, your node should execute, send a request to the service, and wait for the response. If the service is not available, the client will wait until it becomes accessible before proceeding.

3.5.5 Summary of Python Client Node Key Identifiers

- **client.py**: Name of the source file for the Python client node.
- **py_client_node**: Name of the Python client node as defined in the constructor of the Python class.
- **py_client_exe**: Name of executable defined in **setup.py**.
- **/py_add_two_ints_service**: The name of the service that the Python client node calls.

3.6 Practice Assignment: Running ROS 2 Services and Clients

In this assignment, you will run the ROS 2 service and client nodes implemented in both C++ and Python, as described in Sections 3.2, 3.3, 3.4, and 3.5. Capture evidence of successful execution using a screenshot and submit it in the designated assignment area on Canvas (within the ROS 2 module) as a PNG file. The filename must follow this format:

FirstnameLastname_evidence_wXsY.png

where **wX** and **sY** correspond to the week and session numbers. For example, a correct filename would be:

EduardoDavila_evidence_w2s3.png.

3.6.1 Executing ROS 2 Services and Clients

Running the Python Service and Client

After implementing and saving your Python nodes, and updating the **package.xml** and **setup.py** files, build your package and run the nodes. In three separate terminal sessions (or using the **Terminator** emulator), execute the following commands from the root of your workspace:



- In **Terminal 1**, build your package, source the setup file, and launch the Python service node:

```
$ colcon build --packages-select s3_py_srvcli
$ source install/setup.bash
$ ros2 run s3_py_srvcli py_server_exe
```

- In **Terminal 2**, source the setup file and run the Python client node:

```
$ source install/setup.bash
$ ros2 run s3_py_srvcli py_client_exe -11 -13
```

- In **Terminal 3**, send a service request directly from the command line:

```
$ ros2 service call /py_add_two_ints_service example_interfaces/srv/AddTwoInts "{a:
→ -2, b: -3}"
```

You should observe that the `py_server_node` node processes service requests from both the `py_client_node` node and the command-line client in Terminal 3.

Running the C++ Service and Client

After implementing and saving your C++ nodes, and updating the `package.xml` and `CMakeLists.txt` files, build your package and run the nodes. In three additional terminal sessions (or three new sessions in **Terminator**), execute the following commands from the root of your workspace:

- In **Terminal 4**, build your package, source the setup file, and launch the C++ service node:

```
$ colcon build --packages-select s3_cpp_srvcli
$ source install/setup.bash
$ ros2 run s3_cpp_srvcli cpp_server_exe
```

- In **Terminal 5**, source the setup file and run the C++ client node:

```
$ source install/setup.bash
$ ros2 run s3_cpp_srvcli cpp_client_exe 11 13
```

- In **Terminal 6**, send a service request directly from the command line:

```
$ ros2 service call /cpp_add_two_ints_service example_interfaces/srv/AddTwoInts
→ "{a: 2, b: 3}"
```

You should observe that the `cpp_server_node` node processes service requests from both the `cpp_client_node` node and the command-line client in Terminal 6.

Requesting a Service Using `rqt_service_caller`

Use the `rqt_service_caller` tool to send service requests to both the Python and C++ servers. In **Terminal 7**, execute the following command (without needing to enter the workspace directory):

```
$ ros2 run rqt_service_caller rqt_service_caller
```

Then, request a service call for:

- `/py_add_two_ints_service` by entering the values `a: -5, b: -7`.



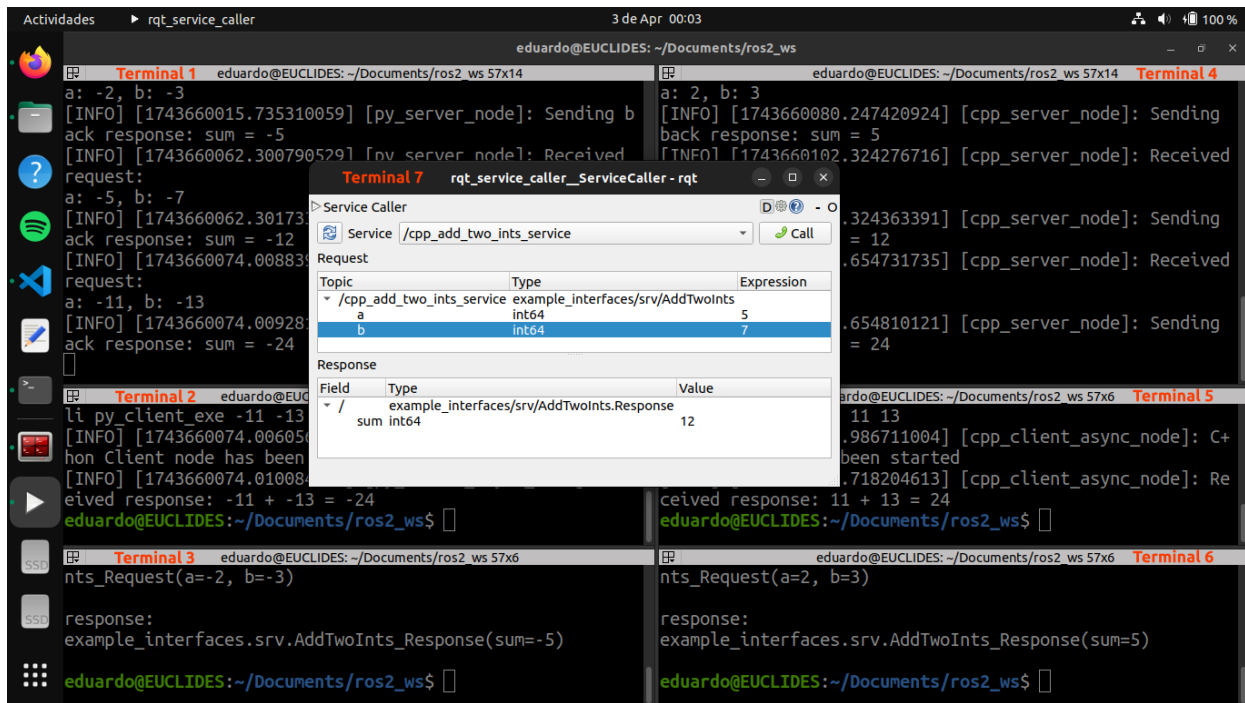


Figure 3.1: Example screenshot showing terminal outputs from the servers and clients in action.

- `/cpp_add_two_ints_service` by entering the values `a: 5, b: 7`.

You should observe that the `py_server_node` and `cpp_server_node` nodes successfully process the service requests sent via `rqt_service_caller`.

3.6.2 Submission Instructions

1. Run the ROS 2 service and client nodes as described above.
2. Capture a **screenshot showing the terminal outputs** from both the C++ and Python service and client nodes, along with the `rqt_service_caller` interface. See Figure 3.1 for reference.
3. Save the screenshot as a PNG file.
4. Name the file following this format: `FirstnameLastname_evidence_wXsY.png` (replace `X` with the week number and `Y` with the session number).
5. Submit the file as instructed by the course guidelines on Canvas.

Note: Your machine's username and hostname must be visible in the screenshot to verify the authenticity of your submission, as shown in Figure 3.1. Any submission that appears copied, unclear, or altered will be invalid and may receive a score of 0 for this assignment.

CHAPTER

4

ROS 2 Custom msg and srv Files

Author: Dr. Eduardo de Jesús Dávila Meza.

 [EduardoDavila-AI-PhD](#)

4.1 Introduction to ROS 2 Services and Clients

In previous chapters 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 is good practice to use predefined interface definitions, you will probably need to define your own messages and services sometimes as well. This chapter will introduce you to the simplest method of creating custom interface definitions. For more details, refer to the guide on [Creating Custom msg and srv Files](#).

4.1.1 Prerequisites

Before you begin, ensure that you are familiar with workspace and package creation, as described in Sections 2.2, 2.3, and 2.4. Additionally, this chapter also uses the source files created in the publisher/subscriber (C++ and Python) in Chapter 2 and service/client (C++ and Python) in Chapter 3 to try out the new custom messages.

Open a terminal and navigate to the `src` directory of your ROS 2 workspace (e.g., `~/ros2_ws`):

```
$ cd ros2_ws/src
```

Replace `ros2_ws` with the actual name of your workspace.

Next, create the following packages in your workspace:

- C++ package for the custom `.msg` and `.srv` files:

```
$ ros2 pkg create s4_custom_interface --build-type ament_cmake --license Apache-2.0
→ --description -"Your package description here"
```

- C++ package for the publisher/subscriber and service/client nodes:

```
$ ros2 pkg create s4_cpp_apps --build-type ament_cmake --dependencies rclcpp
→ s4_custom_interface --license Apache-2.0 --description -"Your package
→ description here"
```

- Python package for the publisher/subscriber and service/client nodes:

```
$ ros2 pkg create s4_py_apps --build-type ament_python --dependencies rclpy
→ s4_custom_interface --license Apache-2.0 --description -"Your package
→ description here"
```

You may replace *s4_custom_interface*, *s4_cpp_apps*, and *s4_py_apps* with package names of your preference.

The *.msg* and *.srv* files are required to be placed in directories called *msg* and *srv* respectively. Create the directories in *ros2_ws/src/s4_custom_interface*:

```
$ mkdir msg srv
```

4.2 Creating the *.msg* and *.srv* files

4.2.1 msg definition

In the *s4_custom_interface/msg* directory you just created, make a new file called *HardwareStatus.msg* with a few lines of code declaring its data structure:

```
int64 temperature
bool are_motors_up
string debug_message
builtin_interfaces/Time the_time
```

This is a custom message that transfers different message types, and uses a message from another message package (*builtin_interfaces/Time*).

Also in the *s4_custom_interface/msg* directory you just created, make a new file called *Sphere.msg* with the following content:

```
geometry_msgs/Point center
float64 radius
```

This custom message uses a message from another message package (*geometry_msgs/Point* in this case).

4.2.2 srv definition

Back in the *s4_custom_interface/srv* directory you just created, make a new file called *AddThreeInts.srv* with the following request and response structure:




```
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.

4.3 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**:

```
# Added manually
find_package(builtin_interfaces REQUIRED)
find_package(geometry_msgs REQUIRED)
find_package(rosidl_default_generators REQUIRED)

# Added manually
rosidl_generate_interfaces(${PROJECT_NAME}
  "msg/HardwareStatus.msg"
  "msg/Sphere.msg"
  "srv/AddThreeInts.srv"
  DEPENDENCIES builtin_interfaces geometry_msgs # Add packages that above messages
  ↪ depend on, in this case builtin_interfaces for HardwareStatus.msg, and geometry_msgs
  ↪ for Sphere.msg
)
```

4.4 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, **s4_custom_interface**, should be associated with, declared using the **<member_of_group>** tag.

Add the following lines within the **<package>** element of **package.xml**:

```
<!-- Added manually -->
<depend>builtin_interfaces</depend>
<depend>geometry_msgs</depend>

<test_depend>ament_lint_auto</test_depend>
<test_depend>ament_lint_common</test_depend>

<!-- Added manually -->
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

4.5 Build the *s4_custom_interface* 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:

```
$ colcon build --packages-select s4_custom_interface
```

Now the interfaces will be discoverable by other ROS 2 packages.

4.6 Confirm msg and srv creation

In a new terminal, run the following command from within your workspace (*/ros2_ws*) to source it:

```
$ source install/setup.bash
```

Now you can confirm that your interface creation worked by using the `ros2 interface show` command:

```
$ ros2 interface show s4_custom_interface/msg/HardwareStatus
```

should return:

```
int64 temperature
bool are_motors_up
string debug_message
builtin_interfaces/Time the_time
  int32 sec
  uint32 nanosec
```

And

```
$ ros2 interface show s4_custom_interface/msg/Sphere
```

should return:

```
geometry_msgs/Point center
  float64 x
  float64 y
  float64 z
float64 radius
```

And

```
$ ros2 interface show s4_custom_interface/srv/AddThreeInts
```

should return:

```
int64 a
int64 b
int64 c
---
int64 sum
```



4.7 Test the new interfaces

For this step you can use the `s4_cpp_apps` and `s4_py_apps` packages you created previously. A few simple modifications to the publisher/subscriber and service/client nodes, `CMakeLists.txt` and `package.xml` files will allow you to use your new interfaces.

4.7.1 Implementing `HardwareStatus.msg` in `Sphere.msg` with pub/sub nodes

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 in C++

Publisher in Python

Subscriber in C++

Subscriber in Python

4.7.2 Implementing `AddThreeInts.srv` in service/client nodes

Service in C++

Service in Python

Client in C++

Client in Python

4.7.3 `CMakeLists.txt`

Add the following lines (C++ package only):



```
# find dependencies
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(s4_custom_interface REQUIRED)

add_executable(cpp_publisher_exe src/publisher.cpp)
ament_target_dependencies(cpp_publisher_exe rclcpp s4_custom_interface)

add_executable(cpp_subscriber_exe src/subscriber.cpp)
ament_target_dependencies(cpp_subscriber_exe rclcpp s4_custom_interface)

add_executable(cpp_server_exe src/server.cpp)
ament_target_dependencies(cpp_server_exe rclcpp s4_custom_interface)

add_executable(cpp_client_exe src/client.cpp)
ament_target_dependencies(cpp_client_exe rclcpp s4_custom_interface)

install(TARGETS
  cpp_publisher_exe
  cpp_subscriber_exe
  cpp_server_exe
  cpp_client_exe
  DESTINATION lib/${PROJECT_NAME})
)
```

4.7.4 setup.py

Add the following lines (Python package only):

```
entry_points={
    'console_scripts': [
        "py_publisher_exe = s4_py_apps.publisher:main",
        "py_subscriber_exe = s4_py_apps.subscriber:main",
        "py_server_exe = s4_py_apps.server:main",
        "py_client_exe = s4_py_apps.client:main"
    ],
}
```

4.7.5 package.xml

Add the following line (for both C++ and Python packages):

```
<depend>s4_custom_interface</depend>
```

4.7.6 Building the Packages and Running the Nodes

After making the above edits and saving all the changes, build the packages:

```
$ colcon build --packages-select s4_cpp_apps
$ colcon build --packages-select s4_py_apps
```



Then open eight new terminals, source `ros2_ws` in each, and run:

- In **terminal 1**, run the C++ publisher node:

```
$ source install/setup.bash
$ ros2 run s4_cpp_apps cpp_publisher_exe
```

- In **terminal 2**, run the C++ subscriber node:

```
$ source install/setup.bash
$ ros2 run s4_cpp_apps cpp_subscriber_exe
```

- In **terminal 3**, run the C++ server node:

```
$ source install/setup.bash
$ ros2 run s4_cpp_apps cpp_server_exe
```

- In **terminal 4**, run the C++ client node:

```
$ source install/setup.bash
$ ros2 run s4_cpp_apps cpp_client_exe 2 3 5
```

- In **terminal 5**, run the Python publisher node:

```
$ source install/setup.bash
$ ros2 run s4_py_apps py_publisher_exe
```

- In **terminal 6**, run the Python subscriber node:

```
$ source install/setup.bash
$ ros2 run s4_py_apps py_subscriber_exe
```

- In **terminal 7**, run the Python server node:

```
$ source install/setup.bash
$ ros2 run s4_py_apps py_server_exe
```

- In **terminal 8**, run the Python client node:

```
$ source install/setup.bash
$ ros2 run s4_py_apps py_client_exe -2 -3 -5
```

Additionally, you can open two new terminals, within your workspace, to run the `rqt_graph` and `rqt_service_caller` tools to visualize the current ROS 2 graph and request services to the server nodes, respectively.

4.8 Practice Assignment: Custom msg and srv Files

In this assignment, you will run the ROS 2 the publisher/subscriber and service/client nodes implemented in both C++ and Python, as described in Sections 4.7.1 and 4.7.2. Capture evidence of successful execution using two screenshots and submit them in the designated assignment area on Canvas (within the ROS 2 module) as PNG files. The filenames must follow this format:

FirstnameLastname_evidence_wXsY_Z.png



where **wX** and **sY** correspond to the week and session numbers, and **Z** to the evidence number. For example, correct filenames would be:

EduardoDavila_evidence_w2s4_1.png,
EduardoDavila_evidence_w2s4_2.png.

4.8.1 Executing ROS 2 Publishers, Subscribers, Services and Clients

After editing and saving your C++ and Python nodes, and updating the **CMakeLists.txt**, **setup.py**, and **package.xml** files, build the packages:

```
$ colcon build --packages-select s4_cpp_apps  
$ colcon build --packages-select s4_py_apps
```

Then, in eight separate terminal sessions (or using the **Terminator** emulator), source **ros2_ws** in each, from the root of your workspace, and run the nodes:

- In **terminal 1**, run the C++ publisher node:

```
$ source install/setup.bash  
$ ros2 run s4_cpp_apps cpp_publisher_exe
```

- In **terminal 2**, run the C++ subscriber node:

```
$ source install/setup.bash  
$ ros2 run s4_cpp_apps cpp_subscriber_exe
```

- In **terminal 3**, run the C++ server node:

```
$ source install/setup.bash  
$ ros2 run s4_cpp_apps cpp_server_exe
```

- In **terminal 4**, run the C++ client node:

```
$ source install/setup.bash  
$ ros2 run s4_cpp_apps cpp_client_exe 2 3 5
```

- In **terminal 5**, run the Python publisher node:

```
$ source install/setup.bash  
$ ros2 run s4_py_apps py_publisher_exe
```

- In **terminal 6**, run the Python subscriber node:

```
$ source install/setup.bash  
$ ros2 run s4_py_apps py_subscriber_exe
```

- In **terminal 7**, run the Python server node:

```
$ source install/setup.bash  
$ ros2 run s4_py_apps py_server_exe
```

- In **terminal 8**, run the Python client node:

```
$ source install/setup.bash  
$ ros2 run s4_py_apps py_client_exe -2 -3 -5
```



You should observe that:

1. The `cpp_publisher` node publishes the custom `HardwareStatus` and `Sphere` messages.
2. The `cpp_subscriber` node receives the custom `HardwareStatus` message from the `py_publisher` node, and receives the `Sphere` message from the `cpp_publisher` node.
3. The `cpp_server` node processes the service request from the `cpp_client` node.
4. The `py_publisher` node publishes the custom `HardwareStatus` and `Sphere` messages.
5. The `py_subscriber` node receives the custom `HardwareStatus` message from the `cpp_publisher` node, and receives the `Sphere` message from the `py_publisher` node.
6. The `py_server` node processes the service request from the `py_client` node.

4.8.2 Visualizing the ROS Graph with `rqt_graph`

To ensure that your publisher, subscriber, service and client nodes are correctly connected, use the `rqt_graph` tool. Then, in **Terminal 9**, execute the following command (without the need to enter the workspace directory):

```
$ ros2 run rqt_graph rqt_graph
```

or simply:

```
$ rqt_graph
```

This visualization should show the topics connecting your publisher and subscriber nodes, along with the service nodes, as described above.

4.8.3 Submission Instructions

1. Run the ROS 2 service and client nodes as described above.
2. Capture a **screenshot showing the terminal outputs** from both the C++ and Python nodes (publishers, subscribers, services, and clients), as described in Section 4.8.1. See Figure 4.1 for reference.
3. Capture a **screenshot showing the ROS 2 graph** from both the C++ and Python nodes (publishers, subscribers, and services), as described in Section 4.8.2. See Figure 4.2 for reference.
4. Save the screenshots as PNG files.
5. Name the files following this format: `FirstnameLastname_evidence_wXsY_Z.png` (replace `X` with the week number, `Y` with the session number, and `Z` with the evidence number).
6. Submit the files as instructed by the course guidelines on Canvas.

Note: Your machine's username and hostname must be visible in the screenshot to verify the authenticity of your submission, as shown in Figure 3.1. Any submission that appears copied, unclear, or altered will be invalid and may receive a score of 0 for this assignment.

```

eduardo@EUCLIDES: ~/Documents/ros2_ws
sage: Eduardo's C++ system runs smoothly, Time: 1744231646.331550955
[INFO] [1744231646.331786592] [cpp_publisher_node]: Publishing Sphere: x: 1.000000, y: 2.000000, z: 3.000000, Radius: 4.000000
age: Eduardo's Python system running smoothly, Time: 1744231646.311976280
[INFO] [1744231646.332182766] [cpp_subscriber_node]: Received Sphere: x: 1.000000, y: 2.000000, z: 3.000000, Radius: 4.000000
[INFO] [1744231296.115117385] [cpp_server_node]: Received request: a: 2, b: 3, c: 5
[INFO] [1744231296.115149123] [cpp_server_node]: Sending back response: sum = 10
[INFO] [1744231295.555245980] [cpp_client_async_node]: Waiting for service to become available...
[INFO] [1744231296.115325323] [cpp_client_async_node]: Received response: 2 + 3 + 5 = 10
eduardo@EUCLIDES: ~/Documents/ros2_ws$

eduardo@EUCLIDES: ~/Documents/ros2_ws
age: Eduardo's Python system running smoothly, Time: 1744231646.311976280
[INFO] [1744231646.315843493] [py_publisher_node]: Publishing Sphere: x: 1.000000, y: 2.000000, z: 3.000000, Radius: 4.000000
us: 4.000000
[INFO] [1744231646.334103638] [py_subscriber_node]: Publishing Status: Temperature: 45, Motors Up: True, Debug Message: Eduardo's C++ system runs smoothly, Time: 1744231646.331550955
[INFO] [1744231301.966252648] [py_server_node]: Received request: a: -2, b: -3, c: -5
[INFO] [1744231301.966804050] [py_server_node]: Sending back response: sum = -10
[INFO] [1744231301.463743508] [py_client_async_node]: Waiting for service to become available...
[INFO] [1744231301.967695199] [py_client_async_node]: Received response: -2 + -3 + -5 = -10
eduardo@EUCLIDES: ~/Documents/ros2_ws$

```

Figure 4.1: Example screenshot showing terminal outputs from the publishers, subscribers, services, and clients in action.

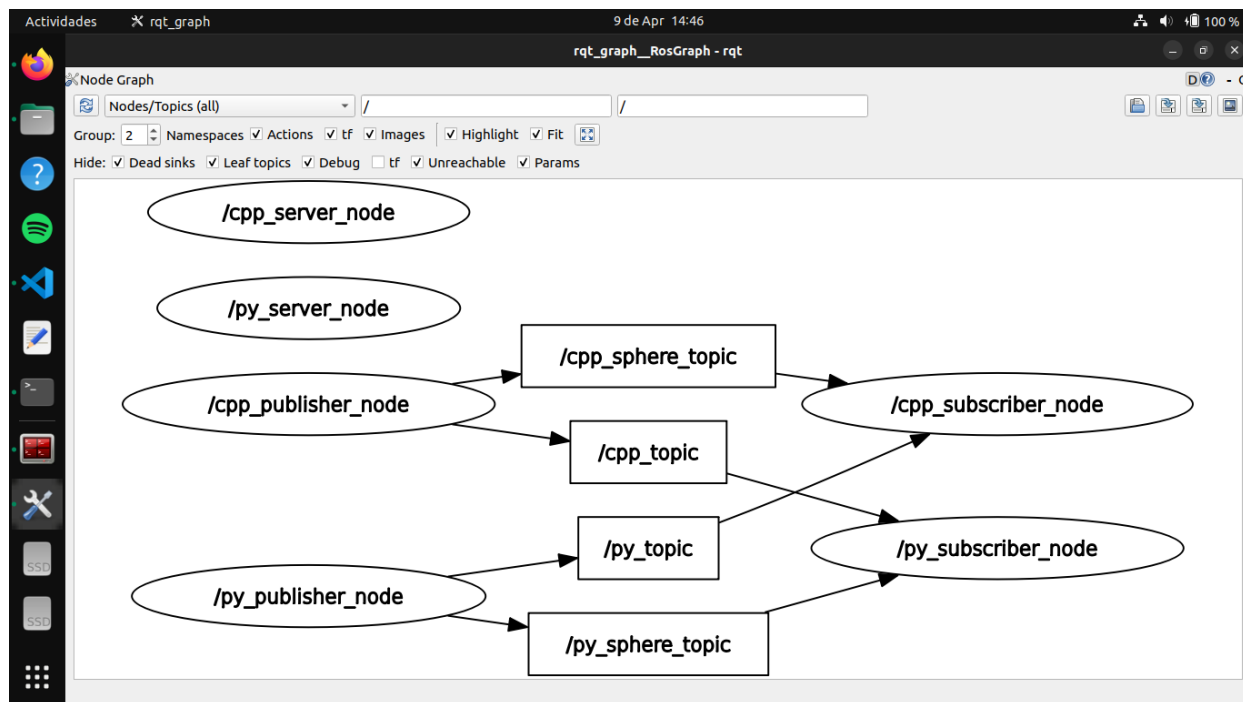


Figure 4.2: Example screenshot showing the ROS 2 graph from the publisher, subscriber, and service nodes.

CHAPTER

Parameters and YAML and Launch files (Image acquisition)

Author: Dr. Eduardo de Jesús Dávila Meza.

 [EduardoDavila-AI-PhD](#)

APPENDIX



Essential ROS 2 Command Reference

Author: Dr. Eduardo de Jesús Dávila Meza.

 [EduardoDavila-AI-PhD](#)

This appendix compiles the commands utilized throughout the course for creating and configuring workspaces, packages, nodes, and using integrated tools in ROS 2.

A.1 System Package Management and Updates in Ubuntu

Command	Description
<code>sudo apt update</code>	Updates the list of available packages and their versions.
<code>sudo apt upgrade</code>	Installs the latest versions of all installed packages.
<code>sudo apt install <package_name></code>	Installs the specified package.
<code>sudo apt install ./<file>.deb</code>	Installs a package from a local <code>.deb</code> file.
<code>sudo apt install ros-humble-desktop</code>	Installs the full desktop version of ROS 2 Humble.
<code>sudo apt install ros-dev-tools</code>	Installs development tools for ROS 2.
<code>sudo apt install ros-humble-turtlesim</code>	Installs the <code>turtlesim</code> package for ROS 2 Humble.
<code>sudo apt install python3-colcon-common-extensions</code>	Installs common extensions for <code>colcon</code> to facilitate package building.
<code>sudo apt install terminator</code>	Installs the Terminator terminal emulator.

Table A.I: Commands for system package management and updates in Ubuntu.

A.2 ROS 2 Environment Setup

Command	Description
<code>source /opt/ros/humble/setup.bash</code>	Sets up the environment for ROS 2 Humble in the current terminal session.
<code>source /usr/share/colcon_argcomplete/hook/colcon-argcomplete.bash</code>	Sets up the autocompletion for <code>colcon</code> in the current terminal session.
<code>echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc</code>	Adds the ROS 2 Humble environment setup to the <code>.bashrc</code> file for future terminal sessions.
<code>echo "source /usr/share/colcon_argcomplete/hook/colcon-argcomplete.bash" >> ~/.bashrc</code>	Enables autocompletion for <code>colcon</code> in the <code>.bashrc</code> file for future terminal sessions.
<code>source ~/.bashrc</code>	Reloads the <code>.bashrc</code> file to apply changes without restarting the terminal.
<code>cat ~/.bashrc</code>	Displays the contents of the <code>.bashrc</code> file.
<code>gedit ~/.bashrc</code>	Opens the <code>.bashrc</code> file in the <code>gedit</code> text editor.

Table A.II: Commands for ROS 2 environment setup.

A.3 Workspace Creation and Building

Command	Description
<code>mkdir -p <ws_name>/src</code>	Creates the workspace directory and its <code>src</code> subdirectory.
<code>cd <ws_name></code>	Navigates to the workspace directory.
<code>colcon build</code>	Builds all packages in the workspace.
<code>colcon build --symlink-install</code>	Builds packages using symbolic links, useful for development purposes.
<code>source ./install/setup.bash</code>	Sets up the environment for the built workspace in the current terminal session.
<code>code .</code>	Opens the current directory in Visual Studio Code.

Table A.III: Commands for workspace creation and building in ROS 2.



A.4 Package Creation and Editing in C++ and Python

Command	Description
<code>cd <ws_name>/src</code>	Navigates to the workspace's <code>src</code> directory.
<code>ros2 pkg create <package_name> --build-type ament_cmake --dependencies rclcpp std_msgs --license Apache-2.0 --description "Your package description here"</code>	Creates a new C++ package with specified dependencies, license, and description.
<code>ros2 pkg create <package_name> --build-type ament_python --dependencies rclpy std_msgs --license Apache-2.0 --description "Your package description here"</code>	Creates a new Python package with specified dependencies, license, and description.
<code>ls</code>	Lists the files and directories in the current directory.
<code>colcon build --packages-select <package_name></code>	Builds only the <code><package_name></code> package.
<code>colcon build --packages-select <package_name> --symlink-install</code>	Builds the <code><package_name></code> package with symbolic link installation.
<code>cd src/<package_name></code>	Navigates to the <code><package_name></code> directory.
<code>code package.xml</code>	Opens the <code>package.xml</code> file in Visual Studio Code.
<code>code CMakeLists.txt</code>	Opens the <code>CMakeLists.txt</code> file in Visual Studio Code.
<code>code setup.py</code>	Opens the <code>setup.py</code> file in Visual Studio Code.
<code>cd <ws_name>/src/<package_name>/src</code>	Navigates to the <code>src</code> subdirectory of a C++ package.
<code>cd <ws_name>/src/<package_name>/<package_name></code>	Navigates to the <code><package_name></code> subdirectory of a Python package.
<code>touch <filename>.cpp</code>	Creates a new C++ source file named <code><filename>.cpp</code> .
<code>touch <filename>.py</code>	Creates a new Python source file named <code><filename>.py</code> .
<code>cd ../../..</code>	Navigates three directories up.

Table A.IV: Package creation and editing in C++ and Python.

A.5 Dev & Debug Essentials: CLI/GUI Commands

Command	Description
<code>ros2 run <package_name> <node_name></code>	Runs the specified node from the specified package.
<code>ros2 run <package_name> <node_name> <args></code>	Runs the specified node from the specified package with specified arguments.
<code>ros2 service call <service_name> <service_type> <args></code>	Calls the service <service_name> from the specified package with specified arguments.
<code>ros2 node list</code>	Lists all active nodes in the ROS 2 system.
<code>ros2 node info <node_name></code>	Displays detailed information about the specified node.
<code>ros2 topic list</code>	Lists all active topics in the ROS 2 system.
<code>ros2 topic info <topic_name></code>	Displays detailed information about the specified topic.
<code>ros2 topic echo <topic_name></code>	Echoes the messages of a specified topic.
<code>ros2 topic pub <topic_name> <msg_type> <args></code>	Publishes a message to the specified topic.
<code>ros2 topic hz <topic_name></code>	Displays the message frequency of a specified topic.
<code>ros2 run rqt_graph rqt_graph</code>	Runs the <code>rqt_graph</code> tool to visualize the ROS 2 nodes and topics.
<code>ros2 run rqt_service_caller rqt_service_caller</code>	Runs the <code>rqt_service_caller</code> tool to call services in a GUI.

Table A.V: Essential command-line and graphical interface commands for running nodes, interacting with topics and services, and debugging ROS 2 systems.

APPENDIX



Using the `geometry_msgs` Package in C++ and Python

Author: Dr. Eduardo de Jesús Dávila Meza.

 [EduardoDavila-AI-PhD](#)

This appendix provides an overview of how to utilize the `geometry_msgs` package in both C++ and Python. This package defines standardized message types for common geometric primitives such as points, vectors, quaternions, and poses, facilitating spatial data representation and interoperability throughout the ROS 2 ecosystem.

B.1 Importing the Package

B.1.1 In C++

To use the `geometry_msgs` in C++, include the necessary header files in your source code:

```
#include <geometry_msgs/msg/point.hpp>
#include <geometry_msgs/msg/quaternion.hpp>
#include <geometry_msgs/msg/pose.hpp>
```

Each message type corresponds to its own header file within the `geometry_msgs/msg` directory.

B.1.2 In Python

In Python, import the required message classes as follows:

```
from geometry_msgs.msg import Point, Quaternion, Pose
```

B.1.3 Including `geometry_msgs` in Your Package Dependencies

To use the `geometry_msgs` message types in your ROS 2 package, you must explicitly declare it as a dependency in both `CMakeLists.txt` (for C++) and `package.xml` (for both C++ and Python packages):

In `CMakeLists.txt`:

```
find_package(geometry_msgs REQUIRED)

ament_target_dependencies(<node_name> geometry_msgs)
```

In `package.xml`:

```
<depend>geometry_msgs</depend>
```

B.2 Using Message Definitions

The following examples demonstrate how to create, assign, and access values for the most commonly used message types from `geometry_msgs`.

B.2.1 Point

Represents a 3D position using Cartesian coordinates `x`, `y`, and `z`.

C++

```
#include <geometry_msgs/msg/point.hpp>

geometry_msgs::msg::Point point;
point.x = 1.0;
point.y = 2.0;
point.z = 3.0;

// Accessing values
double x_value = point.x;
double y_value = point.y;
double z_value = point.z;
```

Python

```
from geometry_msgs.msg import Point

point = Point()
point.x = 1.0
point.y = 2.0
point.z = 3.0

# Accessing values
x_value = point.x
y_value = point.y
z_value = point.z
```



B.2.2 Quaternion

Represents an orientation in free space using quaternion components **x**, **y**, **z**, and **w**. This representation avoids the gimbal lock issues found in Euler angles and supports smooth interpolations.

- **x**, **y**, and **z**: Vector part of the quaternion, defining the axis of rotation.
- **w**: Scalar part, representing the amount of rotation around the axis.
- A unit quaternion (with magnitude 1) is typically required for valid orientation.

C++

```
#include <geometry_msgs/msg/quaternion.hpp>

geometry_msgs::msg::Quaternion quaternion;
quaternion.x = 0.0;
quaternion.y = 0.0;
quaternion.z = 0.0;
quaternion.w = 1.0;

// Accessing values
double w_value = quaternion.w;
```

Python

```
from geometry_msgs.msg import Quaternion

quaternion = Quaternion()
quaternion.x = 0.0
quaternion.y = 0.0
quaternion.z = 0.0
quaternion.w = 1.0

# Accessing values
w_value = quaternion.w
```

B.2.3 Pose

Combines a **position** (as a Point) and **orientation** (as a Quaternion) to define a complete 6-DOF pose in 3D space.

C++

```
#include <geometry_msgs/msg/pose.hpp>

geometry_msgs::msg::Pose pose;
pose.position.x = 1.0;
pose.position.y = 2.0;
pose.position.z = 3.0;
pose.orientation.x = 0.0;
pose.orientation.y = 0.0;
pose.orientation.z = 0.0;
pose.orientation.w = 1.0;

// Accessing values
double pos_x = pose.position.x;
double orient_w = pose.orientation.w;
```



Python

```
from geometry_msgs.msg import Pose

pose = Pose()
pose.position.x = 1.0
pose.position.y = 2.0
pose.position.z = 3.0
pose.orientation.x = 0.0
pose.orientation.y = 0.0
pose.orientation.z = 0.0
pose.orientation.w = 1.0

# Accessing values
pos_x = pose.position.x
orient_w = pose.orientation.w
```

B.3 Additional Message Types

The `geometry_msgs` package includes other message types such as `Twist`, `Accel`, and `Wrench`, each with specific fields for representing different geometric concepts. The usage pattern for these messages is similar: import the message type, create an instance, and assign or access the relevant fields accordingly.

For more detailed information on each message type, refer to the ROS 2 official reference: [geometry_msgs: Humble](#).

