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

List of Acronyms and Special Terms

Acronyms

S

SLAM simultaneous localization and mapping. — Pages: 173, 174, 181, 182, 186.

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 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.

1.4.1 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
```

1.4.2 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
```

1.5 Configuring the Development Environment

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

1.5.1 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.

1.5.2 Installing Recommended VS Code Extensions

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

- **C++** by *Microsoft* - Offers C++ IntelliSense, debugging, and code browsing.
- **CMake** by *twxs* - Simplifies working with CMake projects.
- **CMake Tools** by *Microsoft* - Provides CMake project integration.
- **Error Lens** by *Alexander* - Displays inline error messages in the code editor.
- **Indent-Rainbow** by *oderwat* - Highlights indentation levels with different colors.
- **Python** by *Microsoft* - Provides rich support for Python, including features such as IntelliSense, linting, and debugging.
- **ROS** by *Microsoft* - Adds support for Robot Operating System (ROS) development.
- **URDF** by *smilerobotics* - Adds syntax highlighting and support for URDF files.
- **URDF Visualizer** by *morningfrog* - Allows to preview URDF files within the editor.
- **vscode-icons** by *VSCode Icons Team* - Adds file icons for better visual identification.
- **XML** by *Red Hat* - Enhances XML editing capabilities.
- **XML Tools** by *Josh Johnson* - Offers additional functionalities for XML files.

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*.

1.5.3 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`.

Installing Terminator may have set it as the default, so when you press `Ctrl+Alt+T` it opens Terminator instead of GNOME Terminal (or your original terminal). To achieve your goal, you can change the default and then assign Terminator to a different shortcut, as described in the next.

Reset the Default Terminal Emulator

Ubuntu uses the alternative system for the "x-terminal-emulator." You can reconfigure it so that pressing `Ctrl+Alt+T` launches your original terminal (for example, GNOME Terminal):

1. **Open a terminal** (if Terminator is the only one available, you can temporarily launch it).
2. Run the command:
   ```
   $ sudo update-alternatives --config x-terminal-emulator
   ```
3. You wiell see a list of installed terminal emulators. Type the number corresponding to your original terminal (e.g., `/usr/bin/gnome-terminal.wrapper`) and press **Enter**.

This sets the default terminal emulator that the system shortcut uses.

Create a Custom Shortcut for Terminator

Next, if you want a separate shortcut (like `Ctrl+Alt+E`) that launches Terminator:

1. **Open Settings** in your Ubuntu desktop.
2. Navigate to **Keyboard** (or **Keyboard Shortcuts**).
3. Scroll down to or click on **Custom Shortcuts**.
4. Click the **Add Shortcut** button.
5. Set a name (e.g., "Terminator") and for the command enter:
   ```
   terminator
   ```
6. Assign the new shortcut to `Ctrl+Alt+E` (click the shortcut key field and press the keys).
7. Save your new shortcut.

Now:

- `Ctrl+Alt+T` will launch your default terminal (GNOME Terminal),
- `Ctrl+Alt+E` will launch Terminator.

This way, you keep your usual terminal for everyday tasks and have Terminator available on a separate key combination for when you need its advanced features.

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_sX.png`,

where `sX` indicates the session number, for example:

`EduardoDavila_evidence_s1.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**, run 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.

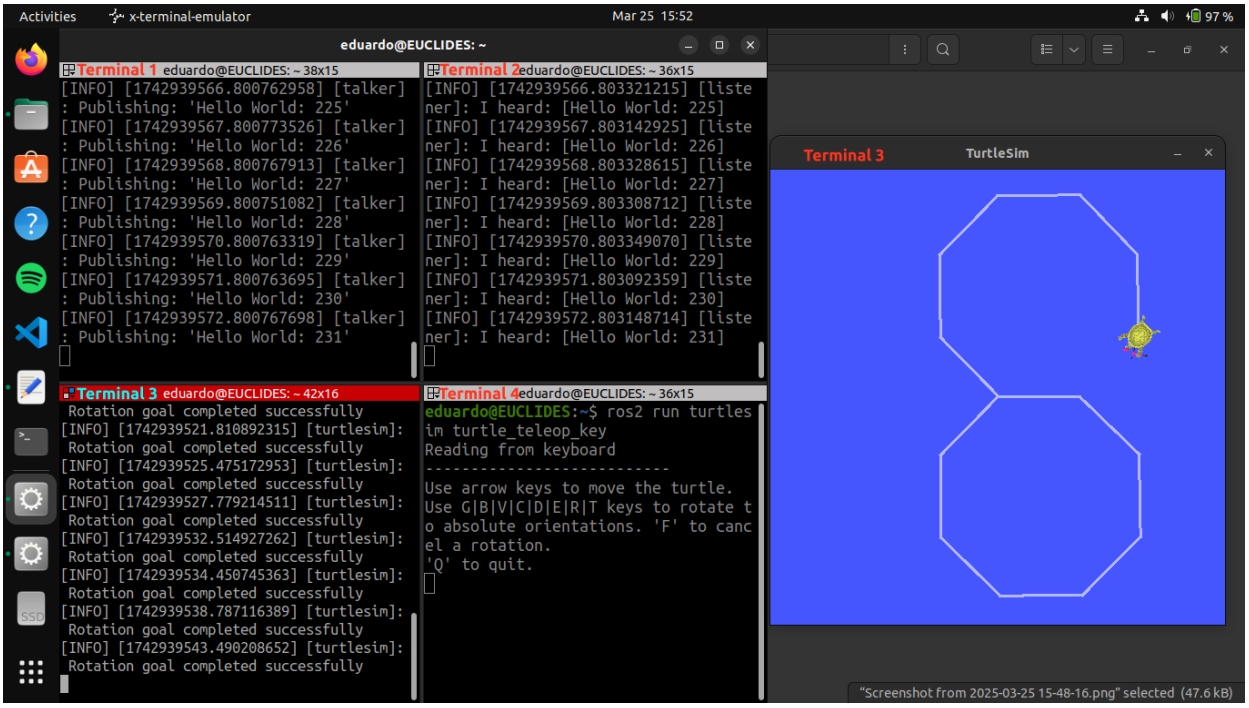


Figure 1.2: Example screenshot showing terminal outputs from both the Talker-Listener example and the Turtlesim node example in action.

4. Name the file according to the format: `FirstnameLastname_evidence_sX.png` (replace `X` 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 2

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 clients for initiating long-running tasks with feedback.
- An action servers 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., `ros2bagrecord`) 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 `ros2pkgcreate` command to create a new C++ package. Replace `s2_cpp_pubsub` with your desired package name and provide an appropriate description:

```
$ ros2 pkg create s2_cpp_pubsub --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:

- s2_cpp_pubsub: Specifies the name of the new package.
- --build-typeament_cmake: Indicates that the package uses C++ and will be built with ament_cmake.
- --dependenciesrclcppstd_msgs: Lists the package dependencies, in this case, rclcpp (the ROS 2 C++ API) and std_msgs (standard message definitions).
- --licenseApache-2.0: Sets the license type for the package.
- --description"Yourpackagedescriptionhere": 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 s2_cpp_pubsub
```

Alternatively, to build all packages in the workspace:

```
$ colcon build
```

2.3.4 Examining the Package Contents

Navigate to the newly created package directory, s2_cpp_pubsub, and list its contents:

```
$ cd src/s2_cpp_pubsub
$ 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 `ros2pkgcreate` command to create a new Python package. Replace `s2_py_pubsub` with your desired package name and provide an appropriate description:

```
$ ros2 pkg create s2_py_pubsub --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:

- `s2_py_pubsub`: Specifies the name of the new package.
- `--build-typeament_python`: Indicates that the package uses Python and will be built with `ament_python`.
- `--dependenciesrclpystd_msgs`: Lists the package dependencies, in this case, `rclpy` (the ROS 2 Python API) and `std_msgs` (standard message definitions).
- `--licenseApache-2.0`: Sets the license type for the package.
- `--description"Yourpackagedescriptionhere"`: 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 s2_py_pubsub
```

Alternatively, to build all packages in the workspace:

```
$ colcon build
```

2.4.4 Examining the Package Contents

Navigate to the newly created package directory, `s2_py_pubsub`, and list its contents:

```
$ cd src/s2_py_pubsub
$ ls
```

You should see the following key files and directories:

- `s2_py_pubsub/`: 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/s2_cpp_pubsub/src
```

Replace `s2_cpp_pubsub` with your actual package name.

Creating the C++ Node File

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

```
$ touch publisher.cpp
```

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

```
$ ls
```

You should now see the `publisher.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` file for editing.

Implementing the Node Code

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

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

```
    private:
        void timer_callback() {
            std_msgs::msg::String 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);
    std::shared_ptr<rclcpp::Node> node = std::make_shared<CppPublisher>();
    rclcpp::spin(node);
    node.reset();
    rclcpp::shutdown();
    return 0;
}
```

This code defines a node that publishes a "Hello, World" message along with a counter to the topic `cpp_str_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 `usingnamespacestd::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_node"), count_(0) {
            RCLCPP_INFO(this->get_logger(), "C++ Publisher node has been started");
            publisher_ = this->create_publisher<std_msgs::msg::String>("cpp_str_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_node"` and sets the counter `count_` to 0.
- `RCLCPP_INFO(this->get_logger(),"C++Publishernodehasbeenstarted");` logs an informational message.
- A publisher is created to send `std_msgs::msg::String` messages on the `"cpp_str_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() {
            std_msgs::msg::String 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 `cpp_str_topic`.
- `rclcpp::TimerBase::SharedPtrtimer_;` is a shared pointer to a timer object that triggers the callback at regular intervals.
- `rclcpp::Publisher<std_msgs::msg::String>::SharedPtrpublisher_;` is a shared pointer to a publisher that sends messages of type `std_msgs::msg::String`.

- `size_tcount_;` is 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);
    std::shared_ptr<rclcpp::Node> node = std::make_shared<CppPublisher>();
    rclcpp::spin(node);
    node.reset();
    rclcpp::shutdown();
    return 0;
}
```

- `rclcpp::init(argc,argv)` initializes ROS 2 communication by parsing command-line arguments and setting up necessary middleware.
- `node=std::make_shared<CppPublisher>()` creates an instance of the `CppPublisher` node.
- `rclcpp::spin(node)` keeps the node active, processing callbacks (such as `timer_callbac k()`) until a shutdown signal is received.
- `node.reset()` releases the shared pointer to the node so that its resources can be deallocated and the destructor can be called before ROS 2 shuts down.
- `rclcpp::shutdown()` gracefully shuts down ROS 2, releasing all allocated resources when the node stops.
- `return0;` indicates 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(publisher_exe src/publisher.cpp)
ament_target_dependencies(publisher_exe rclcpp std_msgs)

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

Breaking down each line:

- `find_package(ament_cmakeREQUIRED)` locates the ament build system.

- `find_package(rclcppREQUIRED)` and `find_package(std_msgsREQUIRED)` ensure that the required dependencies are found.
- `add_executable(publisher_exesrc/publisher.cpp)` creates an executable from your source file.
- `ament_target_dependencies(publisher_exerclcppstd_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 s2_cpp_pubsub
```

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 s2_cpp_pubsub --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/s2_cpp_pubsub
```

or:

```
$ cd install/s2_cpp_pubsub/lib/s2_cpp_pubsub
```

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

```
$ ./publisher_exe
```

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 `ros2run` command:

```
$ ros2 run s2_cpp_pubsub publisher_exe
```

At this point, if everything is set up correctly, your node should be actively publishing messages to the `cpp_str_topic` 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`: The filename of your C++ source file.
2. `cpp_publisher_node`: The name of the node as defined in the constructor of your C++ class.
3. `publisher_exe`: 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/s2_cpp_pubsub/src
```

Replace `s2_cpp_pubsub` with your actual package name.

Creating the C++ Node File

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

```
$ touch subscriber.cpp
```

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

```
$ ls
```

You should now see the `subscriber.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` file for editing.

Implementing the Node Code

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

```cpp
#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_node") {
            RCLCPP_INFO(this->get_logger(), "C++ Subscriber node has been started");
            subscription_ = this->create_subscription<std_msgs::msg::String>(
            "py_str_topic", 10, std::bind(&CppSubscriber::topic_callback, this, _1));
        }

    private:
        void topic_callback(const std_msgs::msg::String & msg) {
            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);
    std::shared_ptr<rclcpp::Node> node = std::make_shared<CppSubscriber>();
    rclcpp::spin(node);
    node.reset();
    rclcpp::shutdown();
    return 0;
}
```

This code defines a node that subscribes to the topic `py_str_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.
- `usingstd::placeholders::_1;` simplifies binding the callback function.

2. Defining the Node Class

```
class CppSubscriber : public rclcpp::Node
{
    public:
        CppSubscriber()
        : Node("cpp_subscriber_node") {
            RCLCPP_INFO(this->get_logger(), "C++ Subscriber node has been started");
            subscription_ = this->create_subscription<std_msgs::msg::String>(
            "py_str_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_node"` and creates a subscription to the `"py_str_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 & msg) {
            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);
    std::shared_ptr<rclcpp::Node> node = std::make_shared<CppSubscriber>();
    rclcpp::spin(node);
    node.reset();
    rclcpp::shutdown();
    return 0;
}
```

- `rclcpp::init(argc,argv)` initializes ROS 2 communication by parsing command-line arguments and setting up necessary middleware.
- `node=std::make_shared<CppSubscriber>()` creates an instance of the `CppSubscriber` node.
- `rclcpp::spin(node)` keeps the node active, processing callbacks (such as `topic_callback()`) until a shutdown signal is received.
- `node.reset()` releases the shared pointer to the node so that its resources can be deallocated and the destructor can be called before ROS 2 shuts down.
- `rclcpp::shutdown()` gracefully shuts down ROS 2, releasing all allocated resources when the node stops.
- `return0;` indicates 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., `publisher_exe`):

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

add_executable(publisher_exe src/publisher.cpp)
ament_target_dependencies(publisher_exe rclcpp std_msgs)

add_executable(subscriber_exe src/subscriber.cpp)
ament_target_dependencies(subscriber_exe rclcpp std_msgs)

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

Breaking down each line:

- `find_package(ament_cmakeREQUIRED)` locates the ament build system.
- `find_package(rclcppREQUIRED)` and `find_package(std_msgsREQUIRED)` ensure that the required dependencies are found.
- `add_executable(subscriber_exesrc/subscriber.cpp)` creates an executable from your source file.
- `ament_target_dependencies(subscriber_exerclcppstd_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 s2_cpp_pubsub
```

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 s2_cpp_pubsub --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/s2_cpp_pubsub
```

or:

```
$ cd install/s2_cpp_pubsub/lib/s2_cpp_pubsub
```

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

```
$ ./subscriber_exe
```

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 `ros2run` command:

```
$ ros2 run s2_cpp_pubsub subscriber_exe
```

At this point, if everything is set up correctly and your Python publisher node is running, your subscriber node should be actively receiving and logging messages published from the `py_str_topic` 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`: The filename of your C++ source file.
2. `cpp_subscriber_node`: The name of the node as defined in the constructor of your C++ class.
3. `subscriber_exe`: 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/s2_py_pubsub/s2_py_pubsub
```

Replace `s2_py_pubsub` with your actual package name.

Creating the Python Node File

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

```
$ touch publisher.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` files.

Making the Python File Executable

Ensure the new Python file is executable:

```
$ chmod +x publisher.py
```

Examine the folder contents again to confirm that `publisher.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` file for editing.

Implementing the Node Code

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

```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_node')
        self.get_logger().info("Python Publisher node has been started")
        self.publisher_ = self.create_publisher(String, 'py_str_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.get_logger().info('Publishing: "%s"' % msg.data)
        self.publisher_.publish(msg)
        self.i += 1
```

```python
# To ensure that resources are properly cleaned up and that
# all intended statements are executed upon shutdown,
# you should handle the KeyboardInterrupt exception.
def main(args=None):
    rclpy.init(args=args)
    node = PyPublisher()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        node.destroy_node()
    finally:
        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 `py_str_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

```python
#!/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_node')
        self.get_logger().info("Python Publisher node has been started")
        self.publisher_ = self.create_publisher(String, 'py_str_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__()` initializes the node with the name `py_publisher_node`.
  - `self.get_logger().info("PythonPublishernodehasbeenstarted")` logs an informational message when the node starts.
  - `self.create_publisher(String,'py_str_topic',10)` creates a publisher that will publish messages of type `String` to the topic `py_str_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 `py_str_topic`.
- An informational log displays the published message.
- The counter `self.i` is incremented for the next callback.

4. The Main Function

```python
# To ensure that resources are properly cleaned up and that
# all intended statements are executed upon shutdown,
# you should handle the KeyboardInterrupt exception.
def main(args=None):
    rclpy.init(args=args)
    node = PyPublisher()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        node.destroy_node()
    finally:
        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(node)` keeps the node active, processing callbacks until a shutdown signal is received.
- The `try-except-finally` block ensures graceful shutdown:
  - `try`: Runs the spinning loop to keep the node active.
  - `exceptKeyboardInterrupt`: Catches the interrupt signal (e.g., `Ctrl+C`) and explicitly destroys the node to release resources.
  - `finally`: Ensures that `rclpy.shutdown()` is called to clean up resources, regardless of how the spin loop is exited.
- `if__name__=='__main__'`: Ensures that the `main()` function is called only when the script is executed directly.

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/s2_py_pubsub/s2_py_pubsub
```

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

```
$ ./publisher.py
```

to run it as an executable, or

```
$ python3 publisher.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': [
        'publisher_exe = s2_py_pubsub.publisher:main',
    ],
},
```

Breaking down the components of this specification:

- `publisher_exe`: Specifies the command-line executable name that you will use to run the node.
- `s2_py_pubsub`: Indicates the name of your package.
- `publisher`: 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 s2_py_pubsub
```

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 s2_py_pubsub --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 `ros2run` command:

```
$ ros2 run s2_py_pubsub publisher_exe
```

At this point, if everything is set up correctly, your node should be actively publishing messages to the `py_str_topic` 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`: The filename of your Python script.
2. `py_publisher_node`: The name of the node as defined in the `super().__init__()` call within your script.
3. `publisher_exe`: 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/s2_py_pubsub/s2_py_pubsub
```

Replace `s2_py_pubsub` with your actual package name.

Creating the Python Node File

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

```
$ touch subscriber.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` files.

Making the Python File Executable

Ensure the new Python file is executable:

```
$ chmod +x subscriber.py
```

Check the folder contents again to confirm that `subscriber.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` file for editing.

Implementing the Node Code

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

```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_node')
        self.get_logger().info("Python Subscriber node has been started")
        self.subscription = self.create_subscription(
            String,
            'cpp_str_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)

# To ensure that resources are properly cleaned up and that
# all intended statements are executed upon shutdown,
# you should handle the KeyboardInterrupt exception.
def main(args=None):
    rclpy.init(args=args)
    node = PySubscriber()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        node.destroy_node()
    finally:
        rclpy.shutdown()

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

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

2.8.3 Breaking Down the Subscriber Node Code

1. Shebang and Imports

```python
#!/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_node')
        self.get_logger().info("Python Subscriber node has been started")
        self.subscription = self.create_subscription(
            String,
            'cpp_str_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_node`.
- `self.get_logger().info("PythonSubscribernodehasbeenstarted")` logs an informational message when the node starts.
- `self.create_subscription` creates a subscription to the topic `cpp_str_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

```
# To ensure that resources are properly cleaned up and that
# all intended statements are executed upon shutdown,
# you should handle the KeyboardInterrupt exception.
def main(args=None):
    rclpy.init(args=args)
    node = PySubscriber()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        node.destroy_node()
    finally:
        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 `PySubscriber` node is created.
- `rclpy.spin(node)` keeps the node active, processing callbacks until a shutdown signal is received.
- The `try-except-finally` block ensures graceful shutdown:
  - `try`: Runs the spinning loop to keep the node active.
  - `exceptKeyboardInterrupt`: Catches the interrupt signal (e.g., `Ctrl+C`) and explicitly destroys the node to release resources.
  - `finally`: Ensures that `rclpy.shutdown()` is called to clean up resources, regardless of how the spin loop is exited.
- `if__name__=='__main__'`: Ensures that the `main()` function is called only when the script is executed directly.

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/s2_py_pubsub/s2_py_pubsub
```

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

```
$ ./subscriber.py
```

to run it as an executable, or

```
$ python3 subscriber.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., `publisher_exe`):

```
entry_points={
    'console_scripts': [
        'publisher_exe = s2_py_pubsub.publisher:main',
        'subscriber_exe = s2_py_pubsub.subscriber:main',
    ],
},
```

Breaking down the components of this specification:

- `subscriber_exe`: Specifies the command-line executable name that you will use to run the subscriber node.

- `s2_py_pubsub`: Indicates the name of your package.
- `subscriber`: 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 s2_py_pubsub
```

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 s2_py_pubsub --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 `ros2run` command:

```
$ ros2 run s2_py_pubsub subscriber_exe
```

At this point, if everything is set up correctly and your C++ publisher node is running, your subscriber node should be actively receiving and logging messages published from the `cpp_str_topic` 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`: The filename of your Python script.
2. `py_subscriber_node`: The name of the node as defined in the `super().__init__()` call within your script.
3. `subscriber_exe`: 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_sX.png`,

where `sX` indicates the session number. For example, in this case, the correct filename would be:

`EduardoDavila_evidence_s2.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 s2_py_pubsub
$ source install/setup.bash
$ ros2 run s2_py_pubsub publisher_exe
```

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

```
$ source install/setup.bash
$ ros2 run s2_py_pubsub subscriber_exe
```

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 s2_cpp_pubsub
$ source install/setup.bash
$ ros2 run s2_cpp_pubsub publisher_exe
```

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

```
$ source install/setup.bash
$ ros2 run s2_cpp_pubsub subscriber_exe
```

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
```

or simply:

```
$ rqt_graph
```

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

2.9.2 Submission Instructions

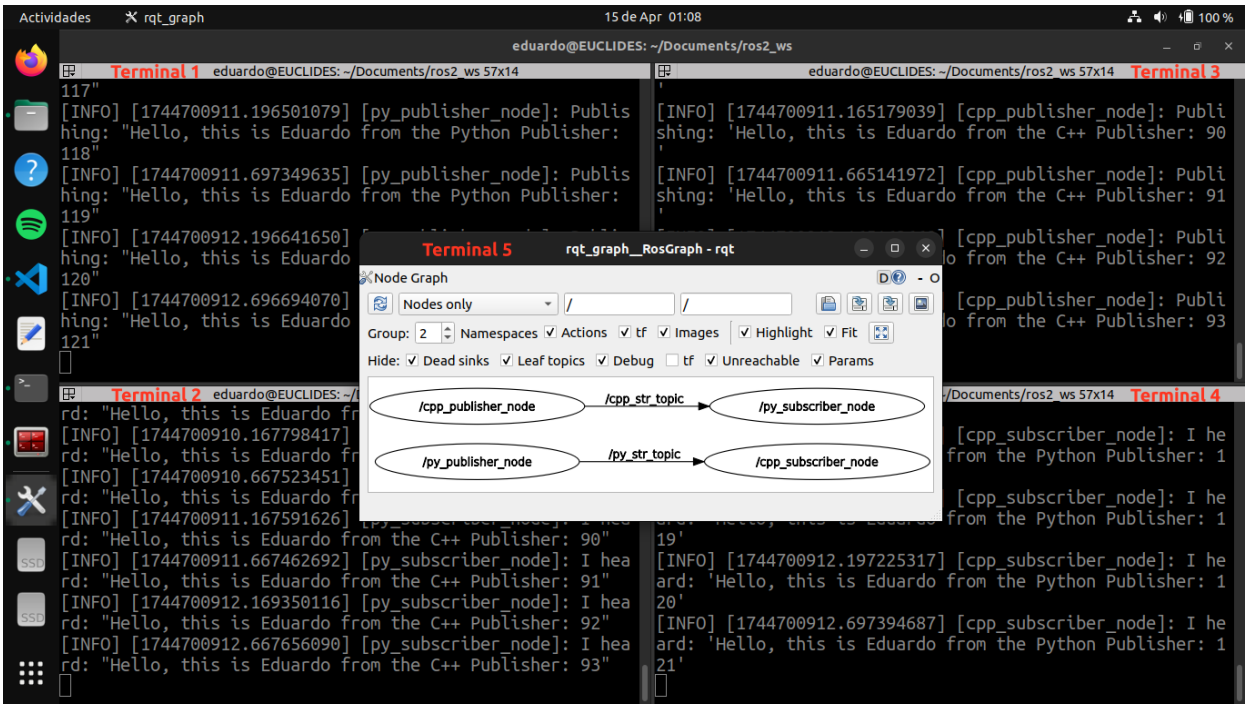


Figure 2.1: Example screenshot showing terminal outputs of the publisher and subscriber nodes, in C++ and Python, in action.

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_sX.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++:

```cpp
#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);
    node.reset();
    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

```cpp
#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++Servernodehasbeenstarted")` 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'isreadytoreceiverequests")` 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);
    node.reset();
    rclcpp::shutdown();
    return 0;
}
```

- `rclcpp::init(argc,argv)` initializes ROS 2 communication by parsing command-line arguments and setting up necessary middleware.
- `node=std::make_shared<CppService>()` creates an instance of the `CppService` node.
- `rclcpp::spin(node)` keeps the node active, processing callbacks (such as those from the timer) until a shutdown signal is received.
- `node.reset()` releases the shared pointer to the node so that its resources can be deallocated and the destructor can be called before ROS 2 shuts down.
- `rclcpp::shutdown()` gracefully shuts down ROS 2, releasing all allocated resources when the node stops.
- `return0;` indicates successful program termination.

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(server_exe src/server.cpp)
ament_target_dependencies(server_exe rclcpp example_interfaces)

install(TARGETS
  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 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.
- `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()`).

```cpp
#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"
                // 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...");
            }
            // Request is called
            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.get()->a, request.get()->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]));
    node.reset();
    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...");
            }
            // Request is called
            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++Clientnodehasbeenstarted")` 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_;
            // 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.get()->a, request.get()->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]));
    node.reset();
    rclcpp::shutdown();
    return 0;
}
```

- `rclcpp::init(argc,argv)` initializes ROS 2 communication by parsing command-line arguments and setting up 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.
- `node.reset()` releases the shared pointer to the node so that its resources can be deallocated and the destructor can be called before ROS 2 shuts down.
- `rclcpp::shutdown()` gracefully shuts down ROS 2, releasing all allocated resources when the node stops.
- `return0;` indicates successful program termination.

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., `server_exe`):

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

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

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

install(TARGETS
  server_exe
  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 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.
- `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:

```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)
    node = PyService()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        node.destroy_node()
    finally:
        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

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

- The shebang line (`#!/usr/bin/envpython3`) 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("PythonServernodehasbeenstarted")` 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'isreadytoreceiverequests")` 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)
    node = PyService()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        node.destroy_node()
    finally:
        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 `PyService` node is created.
- `rclpy.spin(node)` keeps the node active, processing callbacks until a shutdown signal is received.
- The `try-except-finally` block ensures graceful shutdown:
  - `try`: Runs the spinning loop to keep the node active.
  - `exceptKeyboardInterrupt`: Catches the interrupt signal (e.g., `Ctrl+C`) and explicitly destroys the node to release resources.
  - `finally`: Ensures that `rclpy.shutdown()` is called to clean up resources, regardless of how the spin loop is exited.
- `if__name__==’__main__’`: Ensures that the `main()` function is called only when the script is executed directly.

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’: [
        ’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 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.
- `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()`).

```python
#!/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: client_exe a b")
        sys.exit(1)
    node = PyClientAsync()
    future = node.send_request(int(sys.argv[1]), int(sys.argv[2]))
    # Spin until the future is complete
    rclpy.spin_until_future_complete(node, future)
    result = future.result()
    if result is not None:
        node.get_logger().info(
            'Received response: %d + %d = %d' %
            (int(sys.argv[1]), int(sys.argv[2]), result.sum)
        )
    else:
        node.get_logger().error('Failed to call service add_two_ints')
    node.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/envpython3`) 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("PythonClientnodehasbeenstarted")` 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: client_exe a b")
        sys.exit(1)
    node = PyClientAsync()
    future = node.send_request(int(sys.argv[1]), int(sys.argv[2]))
    # Spin until the future is complete
    rclpy.spin_until_future_complete(node, future)
    result = future.result()
    if result is not None:
        node.get_logger().info(
            'Received response: %d + %d = %d' %
            (int(sys.argv[1]), int(sys.argv[2]), result.sum)
        )
    else:
        node.get_logger().error('Failed to call service add_two_ints')
    node.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.
- `node.destroy_node()` explicitly destroys the node to release resources.
- After execution, `rclpy.shutdown()` gracefully shuts down ROS 2, releasing all allocated resources.
- `if__name__=='__main__'`: Ensures that the `main()` function is called only when the script is executed directly.

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., `server_exe`):

```
entry_points={
    'console_scripts': [
        'server_exe = s3_py_srvcli.server:main',
        '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 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.
- `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_sX.png`

where sX corresponds to the session number. For example, a correct filename would be:

EduardoDavila_evidence_s3.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 server_exe
```

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

```
$ source install/setup.bash
$ ros2 run s3_py_srvcli 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 processes service requests from both the py_client 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 server_exe
```

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

```
$ source install/setup.bash
$ ros2 run s3_cpp_srvcli 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 processes service requests from both the cpp_client node and the command-line client in Terminal 6.

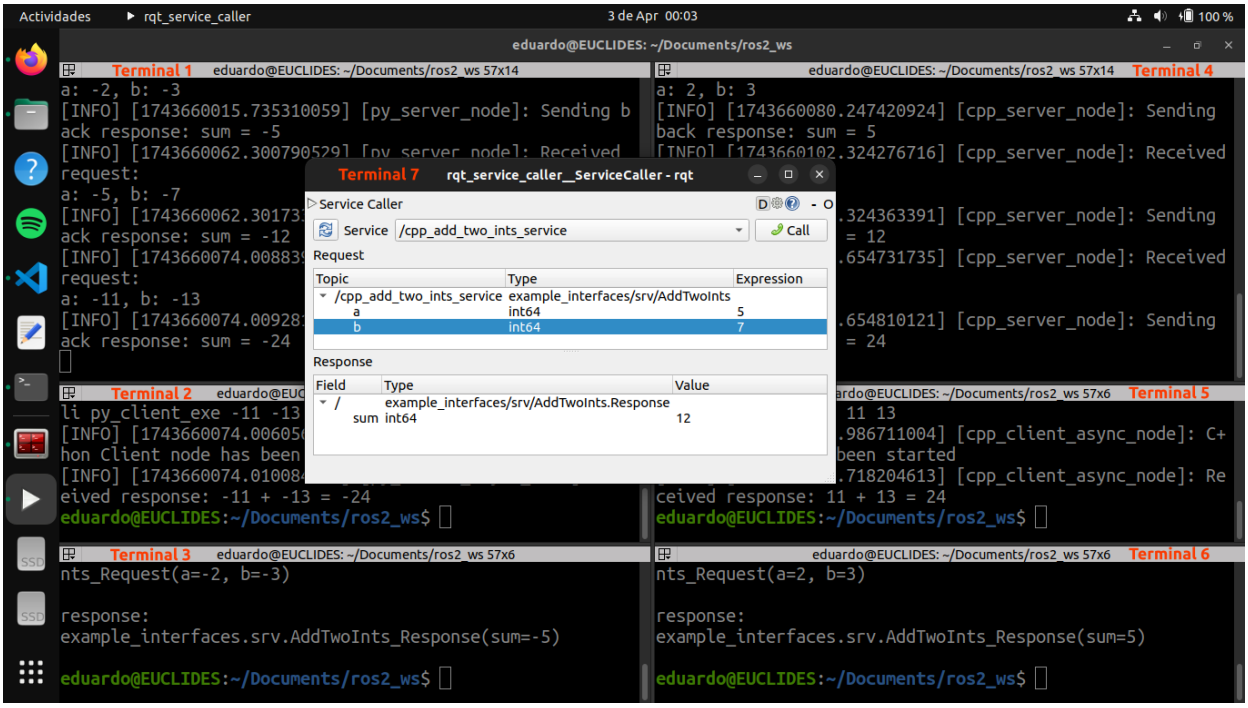


Figure 3.1: Example screenshot showing terminal outputs from the servers and clients in action.

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

You should observe that the `py_server` and `cpp_server` 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_sX.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 a service with 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 `s2_cpp_pubsub`, `s2_py_pubsub`, `s3_cpp_srvcli`, and `s3_py_srvcli` 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` and `Sphere.msg` in 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 will be changing the standard string msg to a numerical one, the output will be slightly different.

Publisher in C++

```cpp
#include <chrono>
#include <memory>
#include "rclcpp/rclcpp.hpp"
#include "s4_custom_interface/msg/hardware_status.hpp"
#include "s4_custom_interface/msg/sphere.hpp"

using namespace std::chrono_literals;

class CppPublisher : public rclcpp::Node {
    public:
        CppPublisher()
        : Node("cpp_publisher_node") {
            RCLCPP_INFO(this->get_logger(), "C++ Publisher node has been started");
            publisher_ =
            ↪ this->create_publisher<s4_custom_interface::msg::HardwareStatus>("cpp_hs_topic",
            ↪ 10);
            publisher2_ =
            ↪ this->create_publisher<s4_custom_interface::msg::Sphere>("cpp_sphere_topic",
            ↪ 10);
            timer_ = this->create_wall_timer(500ms,
            ↪ std::bind(&CppPublisher::timer_callback, this));
        }

    private:
        void timer_callback() {
            // HardwareStatus message
            auto message = s4_custom_interface::msg::HardwareStatus();
            message.temperature = 45;
            message.are_motors_up = true;
            message.debug_message = "Eduardo's C++ system runs smoothly";
            message.the_time = this->get_clock()->now();
```

```
            RCLCPP_INFO(this->get_logger(), "Publishing HardwareStatus: Temperature: %ld,
            ↪  Motors Up: %s, Debug Message: %s, Time: %d.%09d",
                        message.temperature,
                        message.are_motors_up ? "True" : "False",
                        message.debug_message.c_str(),
                        message.the_time.sec,
                        message.the_time.nanosec);

            publisher_->publish(message);

            // Sphere message
            auto message2 = s4_custom_interface::msg::Sphere();
            message2.center.x = 1.0;
            message2.center.y = 2.0;
            message2.center.z = 3.0;
            message2.radius = 4.0;

            RCLCPP_INFO(this->get_logger(), "Publishing Sphere: x: %f, y: %f, z: %f,
            ↪  Radius: %f",
                        message2.center.x,
                        message2.center.y,
                        message2.center.z,
                        message2.radius);

            publisher2_->publish(message2);
        }

        rclcpp::TimerBase::SharedPtr timer_;
        rclcpp::Publisher<s4_custom_interface::msg::HardwareStatus>::SharedPtr
        ↪  publisher_;
        rclcpp::Publisher<s4_custom_interface::msg::Sphere>::SharedPtr publisher2_;
};

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

Publisher in Python

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from s4_custom_interface.msg import HardwareStatus
from s4_custom_interface.msg import Sphere
```

```
class PyPublisher(Node):
    def __init__(self):
        super().__init__('py_publisher_node')
        self.get_logger().info("Python Publisher node has been started")
        self.publisher_ = self.create_publisher(HardwareStatus, 'py_hs_topic', 10)
        self.publisher2_ = self.create_publisher(Sphere, 'py_sphere_topic', 10)
        timer_period = 0.5  # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)

    def timer_callback(self):
        # HardwareStatus message
        msg = HardwareStatus()
        msg.temperature = 45
        msg.are_motors_up = True
        msg.debug_message = "Eduardo's Python system running smoothly"
        msg.the_time = self.get_clock().now().to_msg()
        self.get_logger().info("Publishing HardwareStatus: Temperature: %d, Motors Up:
        ↪  %s, Debug Message: %s, Time: %d.%09d" % (
                    msg.temperature,
                    msg.are_motors_up,
                    msg.debug_message,
                    msg.the_time.sec,
                    msg.the_time.nanosec))
        self.publisher_.publish(msg)

        # Sphere message
        msg2 = Sphere()
        msg2.center.x = 1.0
        msg2.center.y = 2.0
        msg2.center.z = 3.0
        msg2.radius = 4.0
        self.get_logger().info("Publishing Sphere: x: %f, y: %f, z: %f, Radius: %f" % (
                    msg2.center.x,
                    msg2.center.y,
                    msg2.center.z,
                    msg2.radius))
        self.publisher2_.publish(msg2)

def main(args=None):
    rclpy.init(args=args)
    node = PyPublisher()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        node.destroy_node()
        rclpy.shutdown()

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

Subscriber in C++

```cpp
#include <memory>
#include "rclcpp/rclcpp.hpp"
#include "s4_custom_interface/msg/hardware_status.hpp"
#include "s4_custom_interface/msg/sphere.hpp"

using std::placeholders::_1;
class CppSubscriber : public rclcpp::Node{
    public:
        CppSubscriber()
        : Node("cpp_subscriber_node") {
            RCLCPP_INFO(this->get_logger(), "C++ Subscriber node has been started");
            subscription_ =
            ↪ this->create_subscription<s4_custom_interface::msg::HardwareStatus>("py_hs_topic",
            ↪ 10, std::bind(&CppSubscriber::topic_callback, this, _1));
            subscription2_ =
            ↪ this->create_subscription<s4_custom_interface::msg::Sphere>("cpp_sphere_topic",
            ↪ 10, std::bind(&CppSubscriber::sphere_callback, this, _1));
        }
    private:
        void topic_callback(const s4_custom_interface::msg::HardwareStatus::SharedPtr
        ↪ msg) {
            RCLCPP_INFO(this->get_logger(), "Received HardwareStatus: Temperature: %ld,
            ↪ Motors Up: %s, Debug Message: %s, Time: %d.%09d",
                    msg->temperature,
                    msg->are_motors_up ? "True" : "False",
                    msg->debug_message.c_str(),
                    msg->the_time.sec,
                    msg->the_time.nanosec);
        }
        void sphere_callback(const s4_custom_interface::msg::Sphere::SharedPtr msg) {
        RCLCPP_INFO(this->get_logger(), "Received Sphere: x: %f, y: %f, z: %f, Radius:
        ↪ %f",
                    msg->center.x,
                    msg->center.y,
                    msg->center.z,
                    msg->radius);
        }
        rclcpp::Subscription<s4_custom_interface::msg::HardwareStatus>::SharedPtr
        ↪ subscription_;
        rclcpp::Subscription<s4_custom_interface::msg::Sphere>::SharedPtr
        ↪ subscription2_;
};

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

Subscriber in Python

```python
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from s4_custom_interface.msg import HardwareStatus
from s4_custom_interface.msg import Sphere

class PySubscriber(Node):
    def __init__(self):
        super().__init__('py_subscriber_node')
        self.get_logger().info("Python Subscriber node has been started")
        self.subscription = self.create_subscription(
            HardwareStatus, 'cpp_hs_topic', self.listener_callback, 10)
        self.subscription2 = self.create_subscription(
            Sphere, 'py_sphere_topic', self.sphere_callback, 10)
        self.subscription  # prevent unused variable warning
        self.subscription2 # prevent unused variable warning

    def listener_callback(self, msg):
        self.get_logger().info("Received HardwareStatus: Temperature: %d, Motors Up: %s,
        ↪  Debug Message: %s, Time: %d.%09d" % (
                    msg.temperature,
                    msg.are_motors_up,
                    msg.debug_message,
                    msg.the_time.sec,
                    msg.the_time.nanosec))

    def sphere_callback(self, msg):
        self.get_logger().info("Received Sphere: x: %f, y: %f, z: %f, Radius: %f" % (
                    msg.center.x,
                    msg.center.y,
                    msg.center.z,
                    msg.radius))

def main(args=None):
    rclpy.init(args=args)
    node = PySubscriber()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        node.destroy_node()
    finally:
        rclpy.shutdown()

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

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

Service in C++

```cpp
#include "rclcpp/rclcpp.hpp"
#include "s4_custom_interface/srv/add_three_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<s4_custom_interface::srv::AddThreeInts>(
                "cpp_add_three_ints_service",
                std::bind(&CppService::AddThreeInts_callback, this,
                 ↪ std::placeholders::_1, std::placeholders::_2)
            );
            RCLCPP_INFO(this->get_logger(), "Service 'cpp_add_three_ints_service' is
             ↪ ready to receive requests");
        }

    private:
        void AddThreeInts_callback(
            const std::shared_ptr<s4_custom_interface::srv::AddThreeInts::Request>
             ↪ request,
            std::shared_ptr<s4_custom_interface::srv::AddThreeInts::Response> response)
             ↪ {
                response->sum = request->a + request->b + request->c;
                RCLCPP_INFO(this->get_logger(), "Received request:\na: %ld, b: %ld, c:
                 ↪ %ld", request->a, request->b, request->c);
                RCLCPP_INFO(this->get_logger(), "Sending back response: sum = %ld",
                 ↪ response->sum);
            }

        rclcpp::Service<s4_custom_interface::srv::AddThreeInts>::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);
    node.reset();
    rclcpp::shutdown();
    return 0;
}
```

Service in Python

```python
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from s4_custom_interface.srv import AddThreeInts
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(AddThreeInts, 'py_add_three_ints_service',
        ↪  self.AddThreeInts_callback)
        self.get_logger().info("Service 'py_add_three_ints_service' is ready to receive
        ↪  requests")
    def AddThreeInts_callback(self, request, response):
        response.sum = request.a + request.b + request.c
        self.get_logger().info('Received request:\na: %d, b: %d, b: %d' % (request.a,
        ↪  request.b, request.c))
        self.get_logger().info('Sending back response: sum = %d' % (response.sum))
        return response
def main(args=None):
    rclpy.init(args=args)
    node = PyService()
    try:
        rclpy.spin(node)
    except:
        node.destroy_node()
    finally:
        rclpy.shutdown()
if __name__ == '__main__':
    main()
```

Client in C++

```cpp
#include "rclcpp/rclcpp.hpp"
#include "s4_custom_interface/srv/add_three_ints.hpp"
using namespace std::chrono_literals;
class CppClientAsync : public rclcpp::Node{
    public:
        CppClientAsync(int64_t a, int64_t b, int64_t c)
            : Node("cpp_client_async_node"), a_(a), b_(b), c_(c) {
                RCLCPP_INFO(this->get_logger(), "C++ Client node has been started");
                client_ = this->create_client<s4_custom_interface::srv::AddThreeInts>(
                    "cpp_add_three_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...");
            }
            // Request is called
            send_request();
        }

    private:
        void send_request() {
            std::shared_ptr<s4_custom_interface::srv::AddThreeInts::Request> request =
            ↪  std::make_shared<s4_custom_interface::srv::AddThreeInts::Request>();
            request->a = a_;
            request->b = b_;
            request->c = c_;
            rclcpp::Client<s4_custom_interface::srv::AddThreeInts>::FutureAndRequestId
            ↪  future_and_request_id =
            client_->async_send_request(request);

            ↪  std::shared_future<s4_custom_interface::srv::AddThreeInts::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 =
                ↪  %ld", request.get()->a, request.get()->b, request.get()->c,
                ↪  result.get()->sum);
            }
            else {
                RCLCPP_ERROR(this->get_logger(), "Failed to call service
                ↪  add_three_ints");
            }
        }

        rclcpp::Client<s4_custom_interface::srv::AddThreeInts>::SharedPtr client_;
        int64_t a_;
        int64_t b_;
        int64_t c_;
};

int main(int argc, char **argv) {
    rclcpp::init(argc, argv);
    if (argc != 4) {
        RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Usage: client_exe a b c");
        return 1;
    }
    std::shared_ptr<rclcpp::Node> node = std::make_shared<CppClientAsync>(atoll(argv[1]),
    ↪  atoll(argv[2]), atoll(argv[3]));
    node.reset();
    rclcpp::shutdown();
    return 0;
}
```

Client in Python

```python
#!/usr/bin/env python3
import sys
import rclpy
from rclpy.node import Node
from s4_custom_interface.srv import AddThreeInts

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(AddThreeInts, 'py_add_three_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 = AddThreeInts.Request()

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

def main(args=None):
    rclpy.init(args=args)
    if len(sys.argv) != 4:
        print("Usage: client_exe a b c")
        sys.exit(1)
    node = PyClientAsync()
    future = node.send_request(int(sys.argv[1]), int(sys.argv[2]), int(sys.argv[3]))
    # Spin until the future is complete
    rclpy.spin_until_future_complete(node, future)
    result = future.result()
    if result is not None:
        node.get_logger().info(
            'Received response: %d + %d + %d = %d' %
            (int(sys.argv[1]), int(sys.argv[2]), int(sys.argv[3]), result.sum)
        )
    else:
        node.get_logger().error('Failed to call service add_three_ints')
    node.destroy_node()
    rclpy.shutdown()

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

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(publisher_exe src/publisher.cpp)
ament_target_dependencies(publisher_exe rclcpp s4_custom_interface)

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

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

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

install(TARGETS
    publisher_exe
    subscriber_exe
    server_exe
    client_exe
    DESTINATION lib/${PROJECT_NAME}
)
```

4.7.4 setup.py

Add the following lines (Python package only):

```
entry_points={
    'console_scripts': [
        "publisher_exe = s4_py_apps.publisher:main",
        "subscriber_exe = s4_py_apps.subscriber:main",
        "server_exe = s4_py_apps.server:main",
        "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 Python publisher node:

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

- In **terminal 2**, run the Python subscriber node:

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

- In **terminal 3**, run the Python server node:

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

- In **terminal 4**, run the Python client node:

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

- In **terminal 5**, run the C++ publisher node:

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

- In **terminal 6**, run the C++ subscriber node:

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

- In **terminal 7**, run the C++ server node:

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

- In **terminal 8**, run the C++ client node:

```
$ source install/setup.bash
$ ros2 run s4_cpp_apps 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: ROS 2 Custom `.msg` and `.srv` Files

In this assignment, you will run the ROS 2 publisher/subscriber and service/client nodes implemented in both C++ and Python, as described in Section 4.7. 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_sX_Y.png

where sX corresponds to the session number, and Y is the evidence number. For example, correct filenames would be:

EduardoDavila_evidence_s4_1.png,
EduardoDavila_evidence_s4_2.png.

4.8.1 Executing ROS 2 Publishers, Subscribers, Services and Clients

After editing and saving your custom .msg and .srv files, as well as your C++ and Python nodes, and updating the CMakeLists.txt, setup.py, and package.xml files, build the packages:

```
$ colcon build --packages-select s4_custom_interface
$ 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 your ROS 2 workspace in each terminal and run the nodes as follows:

- In **terminal 1**, run the Python publisher node:

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

- In **terminal 2**, run the Python subscriber node:

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

- In **terminal 3**, run the Python server node:

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

- In **terminal 4**, run the Python client node:

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

- In **terminal 5**, run the C++ publisher node:

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

- In **terminal 6**, run the C++ subscriber node:

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

- In **terminal 7**, run the C++ server node:

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

- In **terminal 8**, run the C++ client node:

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

You should observe that:

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

4.8.2 Visualizing the ROS Graph with `rqt_graph`

To verify that the 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 will visualize the ROS 2 graph, showing the service nodes, as well as the topics connecting your publisher and subscriber 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 `rqt_graph`), illustrating the connections among the publisher and subscriber nodes, and the service nodes, 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_sX_Y.png` (replace `X` with the with the session number and `Y` with the evidence number).
6. Submit the files as specified by the course guidelines on Canvas.

Note: Your machine's username and hostname must be visible in the screenshots to verify the authenticity of your submission, as shown in Figure 4.1. Any submission that appears copied, unclear, or altered will be considered invalid and may receive a score of 0.

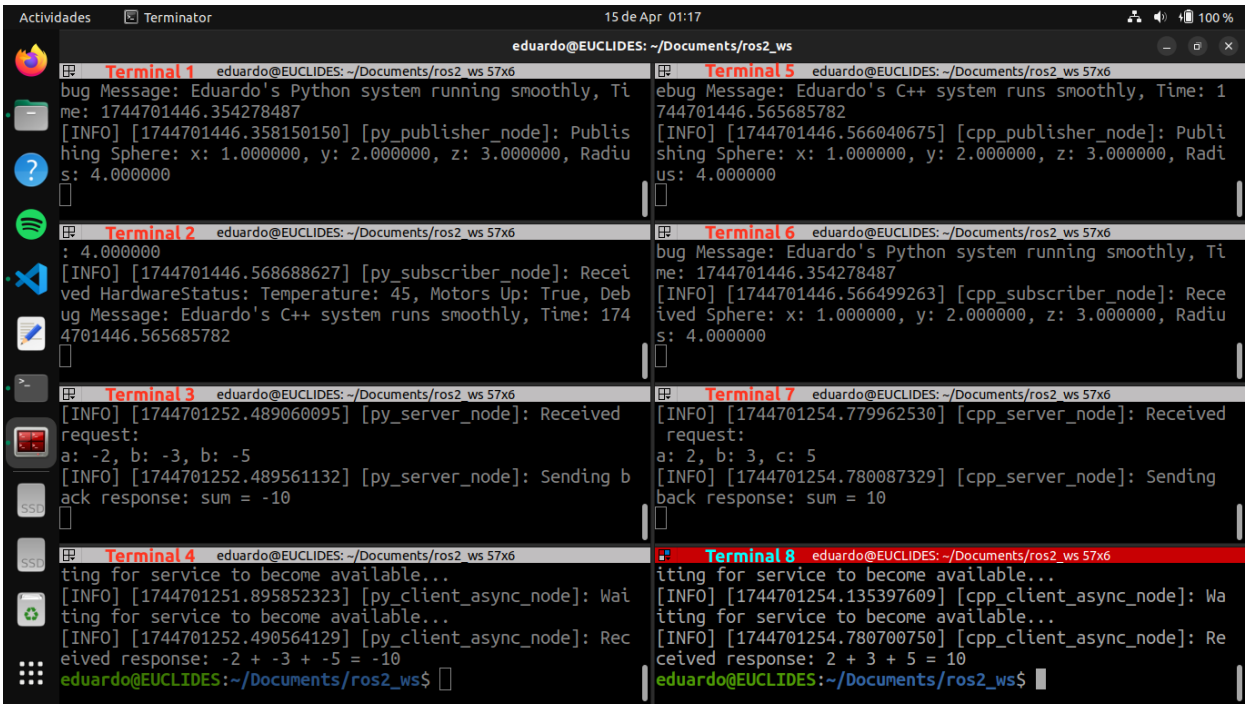


Figure 4.1: Example screenshot showing terminal outputs of the publishers, subscribers, services, and clients, in C++ and Python, in action.

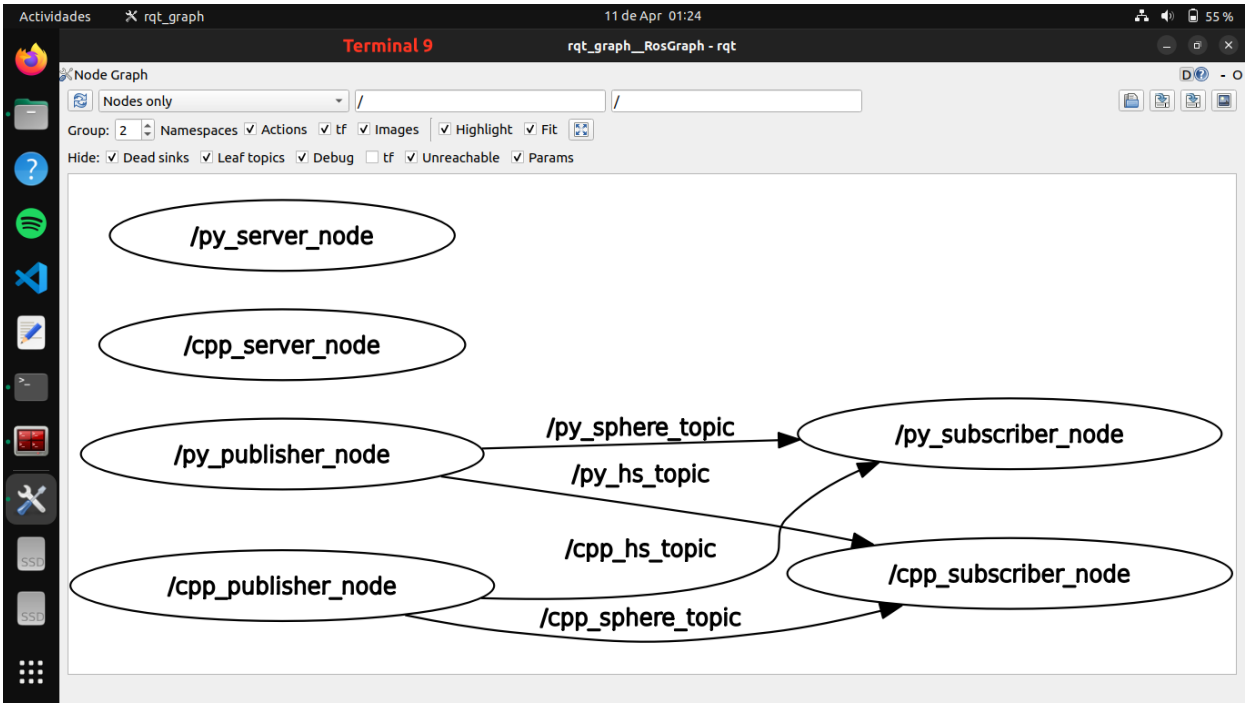


Figure 4.2: Example screenshot showing the ROS 2 graph of the publisher, subscriber, and service nodes.

CHAPTER 5

ROS 2 Parameters and YAML and Launch Files (Image Acquisition)

Author: Dr. Eduardo de Jesús Dávila Meza.
EduardoDavila-AI-PhD

5.1 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.

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 image publisher node:

```
$ ros2 pkg create s5_cpp_camera --build-type ament_cmake --dependencies rclcpp
↪  sensor_msgs image_transport cv_bridge --license Apache-2.0 --description "Your
↪  package description here"
```

- C++ package for the image subscriber node:

```
$ ros2 pkg create s5_cpp_camera_usr --build-type ament_cmake --dependencies rclcpp
↪  sensor_msgs image_transport cv_bridge --license Apache-2.0 --description -"Your
↪  package description here"
```

- Python package for the image publisher node:

```
$ ros2 pkg create s5_py_camera --build-type ament_python --dependencies rclpy
↪  sensor_msgs image_transport cv_bridge --license Apache-2.0 --description "Your
↪  package description here"
```

- Python package for the image subscriber node:

```
$ ros2 pkg create s5_py_camera_usr --build-type ament_python --dependencies rclpy
↪  sensor_msgs image_transport cv_bridge --license Apache-2.0 --description -"Your
↪  package description here"
```

You may replace `s5_cpp_camera`, `s5_cpp_camera_usr`, `s5_py_camera`, and `s5_py_camera_usr` with package names of your preference.

The parameter (`.yaml`) and launch (`.py`) files are required to be placed in directories called `config` and `launch`, respectively. Create the directories in `ros2_ws/src/s5_cpp_camera` for the C++ package, and in `ros2_ws/src/s5_py_camera` for the Python package:

```
$ mkdir config launch
```

5.2 Creating the parameter (.yaml) and launch (.py) Files

5.2.1 Parameter definition

In the `s5_cpp_camera/config` and `s5_py_camera/config` directories you just created, make a new file called `camera_params.yaml` with a few lines of code declaring its data structure. For the C++ package:

```yaml
py_camera_node:
  ros__parameters:
    camera:
      deviceID: 0
      width: 640
      height: 480
```

And for the Python package:

```yaml
cpp_camera_node:
  ros__parameters:
    camera:
      deviceID: 0
      width: 640
      height: 480
```

These are `.yaml` files that assign different parameters for the image publisher nodes.

5.2.2 Launch file definition

Back in the `s5_cpp_camera/launch` and `s5_py_camera/launch` directories you just created, make a new file called `camera_launch.py` with the following structure. For the C++ package:

```
import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    # Path to the YAML file
    params_file = os.path.join(
        get_package_share_directory('s5_cpp_camera'),
        'config',
        'camera_params.yaml'
    )

    return LaunchDescription([
        Node(
            package='s5_cpp_camera',
            name='cpp_camera_node',
            executable='camera_exe',
            parameters=[params_file],
            output = 'screen'
        )
    ])
```

And for the Python package:

```
import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    # Path to the YAML file
    params_file = os.path.join(
        get_package_share_directory('s5_py_camera'),
        'config',
        'camera_params.yaml'
    )

    return LaunchDescription([
        Node(
            package='s5_py_camera',
            name='py_camera_node',
            executable='camera_exe',
            parameters=[params_file],
            output = 'screen'
        )
    ])
```

These are your launch files that run the image publisher nodes and setting the `deviceID`, `width`, and `height` parameters.

5.3 CMakeLists.txt

To make accessible the parameter (`.yaml`) and launch (`.py`) files you defined in the C++ package, add the following lines to `CMakeLists.txt`:

```
# Added manually
install(DIRECTORY
  config
  launch
  DESTINATION share/${PROJECT_NAME}
)
```

5.4 setup.py

To make accessible the parameter (`.yaml`) and launch (`.py`) files you defined in the Python package, add the data files while preserving any existing file setup, with the following lines to `setup.py`:

```
data_files=[
    ('share/ament_index/resource_index/packages',
        ['resource/' + package_name]),
    ('share/' + package_name, ['package.xml']),
    ('share/' + package_name + '/config', ['config/camera_params.yaml']),
    ('share/' + package_name + '/launch', ['launch/camera_launch.py'])
],
```

5.5 package.xml

Make sure the required dependencies are defined within the `package.xml` files, for all the C++ packages and the Python packages:

```
<depend>sensor_msgs</depend>
<depend>image_transport</depend>
<depend>cv_bridge</depend>
```

5.6 Creating a ROS 2 Image Publisher Node with C++

5.6.1 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/s5_cpp_camera/src
```

Replace `s5_cpp_camera` with your actual package name.

Creating the C++ Node File

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

```
$ touch camera.cpp
```

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

```
$ ls
```

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

5.6.2 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 `camera.cpp` file for editing.

Implementing the Node Code

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

Image Publisher in C++

```
#include <rclcpp/rclcpp.hpp>
#include <sensor_msgs/msg/image.hpp>
#include <image_transport/image_transport.hpp>
#include <cv_bridge/cv_bridge.h>
#include <opencv2/opencv.hpp>
#include <chrono>
```

```cpp
class CameraNode : public rclcpp::Node {
public:
    CameraNode()
        : Node("cpp_camera_node"), frame_count_(0), fps_(0.0) {
        RCLCPP_INFO(this->get_logger(), "C++ Camera node has been started");

        // Declare parameters with default values
        this->declare_parameter<int>("camera.deviceID", 0);
        this->declare_parameter<int>("camera.width", 960);
        this->declare_parameter<int>("camera.height", 540);

        // Get parameters
        this->get_parameter("camera.deviceID", deviceID_);
        this->get_parameter("camera.width", width_);
        this->get_parameter("camera.height", height_);

        // Open the camera
        cap_.open(deviceID_);
        if (!cap_.isOpened()) {
            RCLCPP_ERROR(this->get_logger(), "Failed to open camera");
            rclcpp::shutdown();
        }

        // Set camera properties
        cap_.set(cv::CAP_PROP_FRAME_WIDTH, width_);
        cap_.set(cv::CAP_PROP_FRAME_HEIGHT, height_);
        std::cout << "Set Frame Width: " << width_ << std::endl;
        std::cout << "Set Frame Height: " << height_ << std::endl;

        cv::Mat frame;
        cap_ >> frame;

        // Current cam roperties
        std::cout << "Current camera properties:" << std::endl;
        std::cout << "Device ID: " << deviceID_ << std::endl;
        std::cout << "Frame Width: " << frame.cols << std::endl;
        std::cout << "Frame Height: " << frame.rows << std::endl;

        // Initialize start time
        start_time_ = std::chrono::steady_clock::now();
    }
    ~CameraNode() {
        if (cap_.isOpened()) {
            cap_.release();
            RCLCPP_INFO(this->get_logger(), "Camera resource released.");
        }
    }
```

```
    void init() {
        // Initialize ImageTransport and create a publisher
        image_transport::ImageTransport it(shared_from_this());
        pub_ = it.advertise("camera/image_raw", 1);

        // Start the timer for capturing and publishing images
        timer_ = this->create_wall_timer(
            std::chrono::milliseconds(16),
            std::bind(&CameraNode::CaptureAndPublish, this));
    }

private:
    void CaptureAndPublish() {
        cv::Mat frame;
        cap_ >> frame;
        if (!frame.empty()) {
            // Increment frame count
            frame_count_++;

            // Calculate FPS every second
            std::chrono::time_point<std::chrono::steady_clock> now =
            ↪  std::chrono::steady_clock::now();
            std::chrono::duration<double> elapsed = now - start_time_;
            if (elapsed.count() >= 1.0) {
                fps_ = frame_count_ / elapsed.count();
                frame_count_ = 0;
                start_time_ = now;
            }

            // Convert FPS to string
            std::ostringstream fps_text;
            fps_text << "FPS: " << std::fixed << std::setprecision(2) << fps_;

            // Put FPS text on the frame
            cv::putText(frame, fps_text.str(), cv::Point(10,
            ↪  30),cv::FONT_HERSHEY_SIMPLEX, 1.0, cv::Scalar(0, 255, 0), 2);

            // Convert OpenCV image to ROS message
            std_msgs::msg::Header header;
            header.stamp = this->now();
            sensor_msgs::msg::Image::SharedPtr msg = cv_bridge::CvImage(header, "bgr8",
            ↪  frame).toImageMsg();

            // Publish the image
            pub_.publish(msg);
        }
    }

    int deviceID_;
    int width_;
    int height_;
    cv::VideoCapture cap_;
    image_transport::Publisher pub_;
    rclcpp::TimerBase::SharedPtr timer_;
```

```
    // FPS calculation variables
    std::chrono::steady_clock::time_point start_time_;
    int frame_count_;
    double fps_;
};

int main(int argc, char** argv) {
    rclcpp::init(argc, argv);
    std::shared_ptr<CameraNode> node = std::make_shared<CameraNode>();
    node->init();  // Call the initialization function after construction
    rclcpp::spin(node);
    node.reset();
    rclcpp::shutdown();
    return 0;
}
```

5.6.3 Integrating the C++ Image Publisher Node into the Package

Modify `CMakeLists.txt` of your C++ package (`s5_cpp_camera`) to add an executable for the image publisher node:

```
# Added manually
add_executable(camera_exe src/camera.cpp)
ament_target_dependencies(camera_exe rclcpp sensor_msgs image_transport cv_bridge)
target_link_libraries(camera_exe
    ${OpenCV_LIBRARIES}
)

# Added manually
install(TARGETS
    camera_exe
    DESTINATION lib/${PROJECT_NAME}
)

# Added manually
install(DIRECTORY
    config
    launch
    DESTINATION share/${PROJECT_NAME}
)
```

5.6.4 Summary of C++ Image Publisher Node Key Identifiers

- `camera.cpp`: Name of the source file for the C++ image publisher node.
- `cpp_camera_node`: Name of the C++ image publisher node as defined in the constructor of the C++ class.
- `camera_exe`: Name of the executable defined in `CMakeLists.txt`.
- `camera/image_raw`: Name of the topic provided by the C++ image publisher node.

5.7 Creating a ROS 2 Image Subscriber Node with C++

5.7.1 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/s5_cpp_camera_usr/src
```

Replace `s5_cpp_camera_usr` with your actual package name.

Creating the C++ Node File

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

```
$ touch camera_usr.cpp
```

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

```
$ ls
```

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

5.7.2 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 `camera_usr.cpp` file for editing.

Implementing the Node Code

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

Image Subscriber in C++

```cpp
#include <rclcpp/rclcpp.hpp>
#include <sensor_msgs/msg/image.hpp>
#include <image_transport/image_transport.hpp>
#include <cv_bridge/cv_bridge.h>
#include <opencv2/opencv.hpp>

class ImageSubscriber : public rclcpp::Node {
public:
    ImageSubscriber()
        : Node("cpp_camera_user_node") {
        RCLCPP_INFO(this->get_logger(), "C++ Camera User node has been started");
    }
    ~ImageSubscriber() {
        cv::destroyAllWindows();
        RCLCPP_INFO(this->get_logger(), "All OpenCV windows destroyed.");
    }

    void init() {
        // Initialize ImageTransport and create a subscriber using image_transport
        image_transport::ImageTransport it(shared_from_this());
        subscription_ = it.subscribe("camera/image_raw", 1,
        ↪  std::bind(&ImageSubscriber::image_callback, this, std::placeholders::_1));
    }

private:
    void image_callback(const sensor_msgs::msg::Image::ConstSharedPtr & msg) {
        try {
            // Convert ROS Image message to OpenCV image
            cv::Mat cv_image = cv_bridge::toCvShare(msg, "bgr8")->image;
            // Display the image
            cv::imshow("C++ Camera User", cv_image);
            cv::waitKey(1);
            // ImageSubscriber::main_process(cv_image);
        }
        catch (cv_bridge::Exception & e) {
            RCLCPP_ERROR(this->get_logger(), "cv_bridge exception: %s", e.what());
        }
    }

    void main_process(const cv::Mat image) {
        // Resize the image to half its original size
        cv::Mat resized_image;
        cv::resize(image, resized_image, cv::Size(), 0.5, 0.5);

        // Convert the resized image to grayscale
        cv::Mat gray_image;
        cv::cvtColor(resized_image, gray_image, cv::COLOR_BGR2GRAY);

        // Apply standard histogram equalization
        cv::Mat equalized_image;
        cv::equalizeHist(gray_image, equalized_image);
```

```
        // Apply CLAHE
        cv::Mat clahe_image;
        cv::Ptr<cv::CLAHE> clahe = cv::createCLAHE(0.5, cv::Size(8, 8));
        clahe->apply(gray_image, clahe_image);

        // Concatenate images horizontally
        cv::Mat concatenated_image;
        std::vector<cv::Mat> images = {gray_image, equalized_image, clahe_image};
        cv::hconcat(images, concatenated_image);

        // Display the concatenated images
        cv::imshow("Grayscale - HE - CLAHE (CPP)", concatenated_image);
        cv::waitKey(1);
    }

    image_transport::Subscriber subscription_;
};

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

5.7.3 Integrating the C++ Image Subscriber Node into the Package

Modify `CMakeLists.txt` of your C++ package (`s5_cpp_camera_usr`) to add an executable for the image subscriber node:

```
# Added manually
add_executable(camera_usr_exe src/camera_usr.cpp)
ament_target_dependencies(camera_usr_exe rclcpp sensor_msgs image_transport cv_bridge)
target_link_libraries(camera_usr_exe
    ${OpenCV_LIBRARIES}
)

# Added manually
install(TARGETS
    camera_usr_exe
    DESTINATION lib/${PROJECT_NAME}
)
```

5.7.4 Summary of C++ Image Subscriber Node Key Identifiers

- `camera_usr.cpp`: Name of the source file for the C++ image subscriber node.
- `cpp_camera_user_node`: Name of the C++ image publisher node as defined in the constructor of the C++ class.

- `camera_usr_exe`: Name of the executable defined in `CMakeLists.txt`.
- `camera/image_raw`: Name of the topic that the C++ image subscriber node is subscribed.

5.8 Creating a ROS 2 Image Publisher Node with Python

5.8.1 Setting Up the Python Node

Navigating to the Package Directory

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

```
$ cd ros2_ws/src/s5_py_camera/s5_py_camera
```

Replace `s5_py_camera` with your actual package name.

Creating the Python Node File

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

```
$ touch camera.py
```

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

```
$ ls
```

You should now see the `camera.py` file.

5.8.2 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 `camera.py` file for editing.

Implementing the Node Code

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

Image Publisher in Python

```python
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Image
from cv_bridge import CvBridge
import cv2
from std_msgs.msg import Header
import time

class CameraNode(Node):
    def __init__(self):
        super().__init__('py_camera_node')
        self.get_logger().info("Python Camera node has been started")

        # Declare parameters with default values
        self.declare_parameter('camera.deviceID', 0)
        self.declare_parameter('camera.width', 960)
        self.declare_parameter('camera.height', 540)

        # Get parameters
        self.deviceID =
        ↪  self.get_parameter('camera.deviceID').get_parameter_value().integer_value
        self.width =
        ↪  self.get_parameter('camera.width').get_parameter_value().integer_value
        self.height =
        ↪  self.get_parameter('camera.height').get_parameter_value().integer_value

        # Open the camera
        self.cap = cv2.VideoCapture(self.deviceID)
        if not self.cap.isOpened():
            self.get_logger().error('Failed to open camera')
            rclpy.shutdown()

        # Set camera properties
        self.cap.set(cv2.CAP_PROP_FRAME_WIDTH, self.width)
        self.cap.set(cv2.CAP_PROP_FRAME_HEIGHT, self.height)
        print(f"Set Frame Width: {self.width}")
        print(f"Set Frame Height: {self.height}")

        ret, frame = self.cap.read()

        # Current cam roperties
        print(f"Current camera properties:")
        print(f"Device ID: {self.deviceID}")
        print(f"Frame Width: {frame.shape[1]}")
        print(f"Frame Height: {frame.shape[0]}")

        # Initialize start time
        self.start_time = time.time()
        self.fps = 0.0
```

```
        # Initialize CvBridge and create a publisher
        self.bridge = CvBridge()
        self.publisher = self.create_publisher(Image, 'camera/image_raw', 1)

        # Start the timer for capturing and publishing images
        self.timer = self.create_timer(0.016, self.CaptureAndPublish)
        self.frame_count = 0

    def CaptureAndPublish(self):
        ret, frame = self.cap.read()
        if ret:
            # Increment frame count
            self.frame_count += 1

            # Calculate FPS every second
            elapsed_time = time.time() - self.start_time
            if elapsed_time >= 1.0:
                self.fps = self.frame_count / elapsed_time
                self.frame_count = 0
                self.start_time = time.time()

            # Convert FPS to string
            fps_text = f"FPS: {self.fps:.2f}"

            # Put FPS text on the frame
            cv2.putText(frame, fps_text, (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 1.0, (0,
            ↪  255, 0), 2)

            # Convert OpenCV image to ROS message
            header = Header()
            header.stamp = self.get_clock().now().to_msg()
            image_message = self.bridge.cv2_to_imgmsg(frame, encoding='bgr8')
            image_message.header = header

            # Publish the image
            self.publisher.publish(image_message)

def main(args=None):
    rclpy.init(args=args)
    node = CameraNode()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        node.cap.release()
        node.get_logger().info("Camera resource released.")
    finally:
        node.destroy_node()
        rclpy.shutdown()

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

5.8.3 Integrating the Python Image Publisher Node into the Package

Modify `setup.py` of your Python package (`s5_py_camera`) to add an executable for the image publisher node:

```
entry_points={
    'console_scripts': [
        'camera_exe = s5_py_camera.camera:main',
    ],
},
```

5.8.4 Summary of Python Image Publisher Node Key Identifiers

- `camera.py`: Name of the source file for the Python image publisher node.
- `py_camera_node`: Name of the Python image publisher node as defined in the constructor of the Python class.
- `camera_exe`: Name of the executable defined in `setup.py`.
- `camera/image_raw`: Name of the topic provided by the Python image publisher node.

5.9 Creating a ROS 2 Image Subscriber Node with Python

5.9.1 Setting Up the Python Node

Navigating to the Package Directory

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

```
$ cd ros2_ws/src/s5_py_camera_usr/s5_py_camera_usr
```

Replace `s5_py_camera_usr` with your actual package name.

Creating the Python Node File

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

```
$ touch camera_usr.py
```

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

```
$ ls
```

You should now see the `camera_usr.py` file.

5.9.2 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 `camera_usr.py` file for editing.

Implementing the Node Code

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

Image Subscriber in Python

```python
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Image
from cv_bridge import CvBridge, CvBridgeError
import cv2
import numpy as np

class ImageSubscriber(Node):
    def __init__(self):
        super().__init__('py_camera_user_node')
        self.get_logger().info("Python Camera User node has been started")

        # Initialize CvBridge and create a subscriber
        self.bridge = CvBridge()
        self.subscription = self.create_subscription(Image, 'camera/image_raw',
        ↪  self.image_callback, 1)

    def image_callback(self, msg):
        try:
            # Convert ROS Image message to OpenCV image
            cv_image = self.bridge.imgmsg_to_cv2(msg, 'bgr8')

            # Display the image
            cv2.imshow('Python Camera User', cv_image)
            cv2.waitKey(1)
            # self.main_process(cv_image)
        except CvBridgeError as e:
            self.get_logger().error(f'CvBridge Error: {e}')
    def main_process(self, image):
        # Get the height and width
        height, width = image.shape[:2]

        # Resize the image
        resized_image = cv2.resize(image, (int(0.5*width), int(0.5*height)))

        # Convert the image to grayscale
        gray_image = cv2.cvtColor(resized_image, cv2.COLOR_BGR2GRAY)
```

```
        # Apply histogram equalization
        equalized_image = cv2.equalizeHist(gray_image)

        # create a CLAHE object (Arguments are optional).
        clahe = cv2.createCLAHE(clipLimit=0.5, tileGridSize=(8,8))
        clahe_image = clahe.apply(gray_image)

        # Display the images, stacking them side-by-side
        stacked_img = np.hstack((gray_image, equalized_image, clahe_image))
        cv2.imshow("Grayscale - HE - CLAHE (PY)", stacked_img)
        cv2.waitKey(1)

def main(args=None):
    rclpy.init(args=args)
    node = ImageSubscriber()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        cv2.destroyAllWindows()
        node.get_logger().info("All OpenCV windows destroyed.")
    finally:
        node.destroy_node()
        rclpy.shutdown()

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

5.9.3 Integrating the Python Image Subscriber Node into the Package

Modify `setup.py` of your Python package (`s5_py_camera_usr`) to add an executable for the image subscriber node:

```
entry_points={
    'console_scripts': [
        'camera_user_exe = s5_py_camera_usr.camera_usr:main',
    ],
},
```

5.9.4 Summary of Python Image Subscriber Node Key Identifiers

- `camera_usr.py`: Name of the source file for the Python image subscriber node.
- `py_camera_user_node`: Name of the Python image publisher node as defined in the constructor of the Python class.
- `camera_usr_exe`: Name of the executable defined in `setup.py`.
- `camera/image_raw`: Name of the topic that the Python image subscriber node is subscribed.

5.10 Building the Packages and Running the Nodes

After making the above edits and saving all the changes, in the root of your workspace (e.g., `/ros2_ws`), build the packages:

```
$ colcon build --packages-select s5_cpp_camera
$ colcon build --packages-select s5_cpp_camera_usr
$ colcon build --packages-select s5_py_camera
$ colcon build --packages-select s5_py_camera_usr
```

Then, in three terminals, source `ros2_ws` in each, and run:

- In **terminal 1**, run either the C++ image publisher node or the Python image publisher node:

```
$ source install/setup.bash
$ ros2 run s5_cpp_camera camera_exe
```

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

```
$ source install/setup.bash
$ ros2 run s5_cpp_camera_usr camera_usr_exe
```

- In **terminal 3**, run the Python image subscriber node:

```
$ source install/setup.bash
$ ros2 run s5_py_camera_usr camera_usr_exe
```

Additionally, you can open two additional terminals to run the `rqt_graph` and `rqt_image_view` tools to visualize the current ROS 2 graph and visualize the enabled image topic, respectively.

5.11 Practice Assignment: ROS 2 Parameters and YAML and Launch files (Image Acquisition)

In this assignment, you will run the ROS 2 image publisher and subscriber nodes implemented in both C++ and Python, as described in Sections 5.6, 5.7, 5.8, and 5.9. 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_sX_Y.png`

where `sX` corresponds to the session number, and `Y` is the evidence number. For example, correct filenames would be:

`EduardoDavila_evidence_s5_1.png`,
`EduardoDavila_evidence_s5_2.png`.

5.11.1 Executing ROS 2 Image Publishers and Subscribers

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 s5_cpp_camera
$ colcon build --packages-select s5_cpp_camera_usr
$ colcon build --packages-select s5_py_camera
$ colcon build --packages-select s5_py_camera_usr
```

Then, in three separate terminal sessions (or using the `Terminator` emulator), source your ROS 2 workspace in each terminal and run the nodes as follows:

- In **terminal 1**, run either the C++ image publisher node or the Python image publisher node:

```
$ source install/setup.bash
$ ros2 run s5_cpp_camera camera_exe
```

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

```
$ source install/setup.bash
$ ros2 run s5_cpp_camera_usr camera_usr_exe
```

- In **terminal 3**, run the Python image subscriber node:

```
$ source install/setup.bash
$ ros2 run s5_py_camera_usr camera_usr_exe
```

You should observe that:

1. The `cpp_camera` `py_camera` node publishes the image message.
2. The `cpp_camera_usr` and `py_camera_usr` nodes receive the image message.

5.11.2 Visualizing the image topic with `rqt_image_view`

To verify that the publisher node is correctly publishing the image topic, use the `rqt_image_view` tool. Then, in **Terminal 4**, execute the following command (without the need to enter the workspace directory):

```
$ ros2 run rqt_image_view rqt_image_view
```

This will visualize the image acquisition provided by either the C++ publisher node or the Python publisher node.

5.11.3 Visualizing the ROS Graph with `rqt_graph`

To verify that the 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
```

or simply:

```
$ rqt_graph
```

This will visualize the ROS 2 graph, showing the topic connecting your publisher and subscriber nodes, as described above.

5.11.4 Submission Instructions

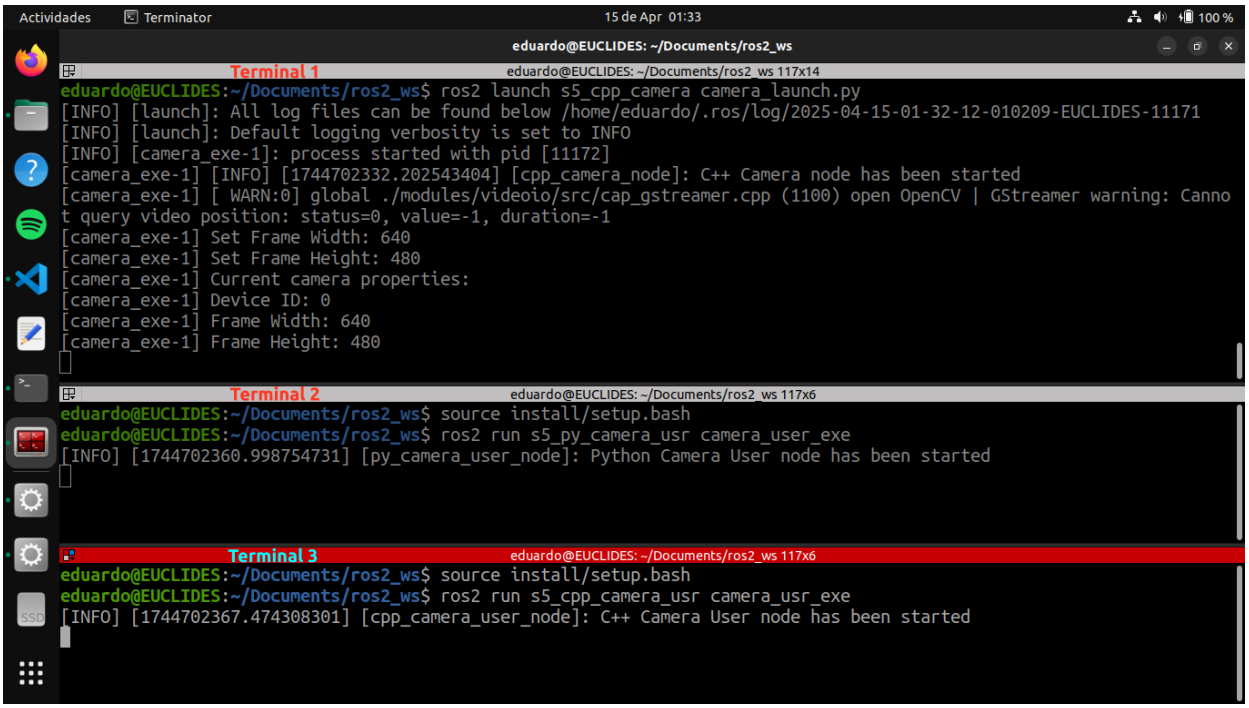


Figure 5.1: Example screenshot showing terminal outputs of the publisher and subscribers, in C++ and Python, in action.

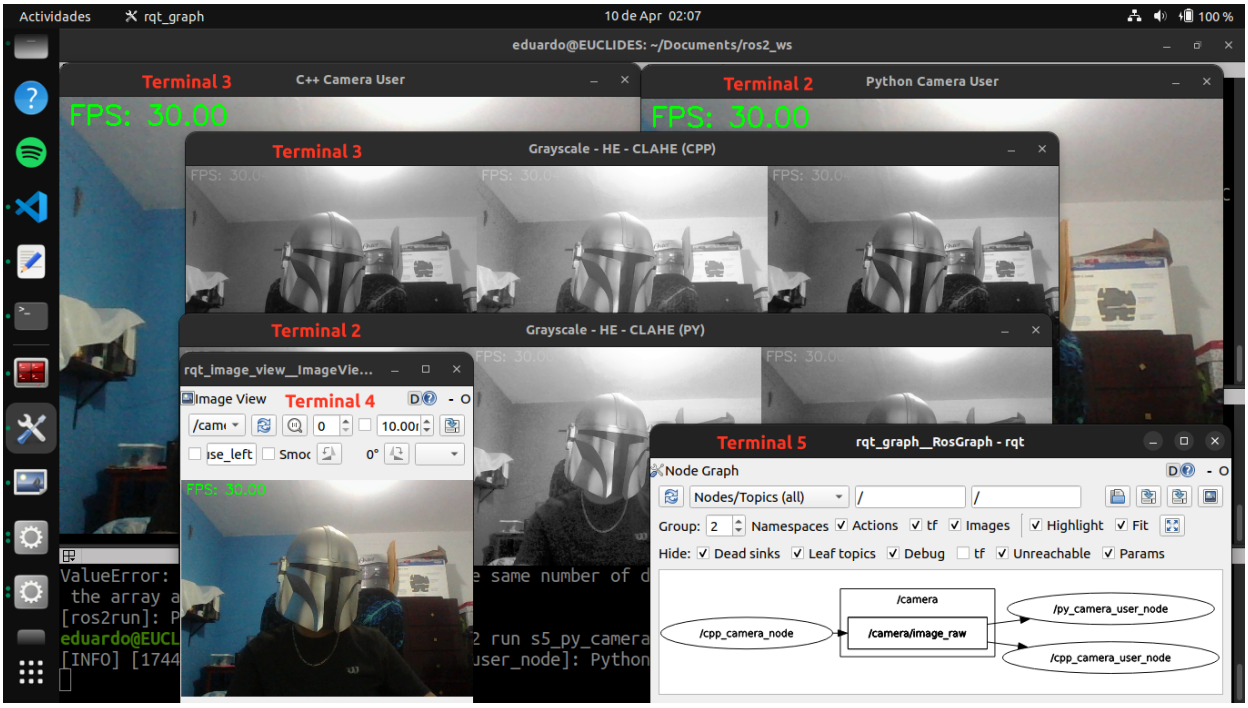


Figure 5.2: Example screenshot showing the image windows and the ROS 2 graph of the publisher and subscriber nodes.

1. Run the ROS 2 image publisher and subscriber nodes as described above.
2. Capture a **screenshot showing the terminal outputs** from both the C++ and Python nodes (publisher and subscribers), as described in Section 5.11.1. See Figure 5.1 for reference.
3. Capture a **screenshot showing the image windows and the ROS 2 graph** (from `OpenCV`, `rqt_image_view` and `rqt_graph`), illustrating the image publication and subscription, and the connections among the publisher and subscriber nodes, as described in Sections 5.11.1, 5.11.2, and 5.11.3. See Figure 5.2 for reference.
4. Save the screenshots as PNG files.
5. Name the files following this format: `FirstnameLastname_evidence_sX_Y.png` (replace `X` with the session number and `Y` with the evidence number).
6. Submit the files as specified by the course guidelines on Canvas.

Note: Your machine's username and hostname must be visible in the screenshots to verify the authenticity of your submission, as shown in Figure 5.1. Any submission that appears copied, unclear, or altered will be considered invalid and may receive a score of 0.

CHAPTER 6

URDF Modeling and Launch Files (Robot States)

Author: Dr. Eduardo de Jesús Dávila Meza.
EduardoDavila-AI-PhD

6.1 Introduction

URDF (Unified Robot Description Format) is an XML-based format used to describe the physical configuration, geometry, and properties of robots in ROS 2.

All the URDF models and associated source files referenced in this chapter are available in our ROS 2 course GitHub repository.

6.2 Prerequisites

6.2.1 Workspace and Package Creation

Before getting started, make sure 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:

- A C++ package to contain the URDF models and state publisher nodes:

```
$ ros2 pkg create s6_cpp_urdf --build-type ament_cmake --dependencies rclcpp
↪  geometry_msgs sensor_msgs tf2_ros tf2_geometry_msgs --license Apache-2.0
↪  --description "Your package description here"
```

- A Python package to contain the URDF models and state publisher nodes:

```
$ ros2 pkg create s6_py_urdf --build-type ament_python --dependencies rclpy
↪  geometry_msgs sensor_msgs tf2_ros tf_transformations --license Apache-2.0
↪  --description "Your package description here"
```

Feel free to replace `s6_cpp_urdf` and `s6_py_urdf` with names of your preference.

The launch (`.py`), mesh (`.STL` and `.dae`), RViz (`.rviz`), and URDF (`.urdf`) files should be organized into directories named `launch`, `meshes`, `rviz`, and `urdf`, respectively. Create these directories in both the C++ and Python packages:

```
$ mkdir launch meshes rviz urdf
```

6.2.2 Installing Required Packages

Before proceeding, ensure that the following library and packages are installed. Update your package list (apt repository) and install them with:

```
$ sudo apt update
$ sudo apt install ros-humble-joint-state-publisher
$ sudo apt install ros-humble-urdf-tutorial
$ sudo apt install ros-humble-tf-transformations
$ sudo pip3 install transforms3d
$ sudo apt install ros-humble-tf2-tools # or ros-humble-tf2*
```

Also, from the root of your workspace, use `rosdep` to check for any other missing dependencies before building:

```
$ rosdep install -i --from-path src --rosdistro humble -y
```

6.2.3 Installing Recommended VS Code Extensions

For an improved development experience, we recommend installing the following VS Code extensions, already listed in Section 1.5.2:

- **URDF** by *smilerobotics* - Adds syntax highlighting and support for URDF files.
- **URDF Visualizer** by *morningfrog* - Allows to preview URDF files within the editor.

6.3 Exploring and Visualizing URDF (`.urdf`) Files

In this section, we will explore and analyze various visual URDF models. Some models are simple and consist of just a few lines, whereas others are more complex, exceeding 200 lines of `XML`.

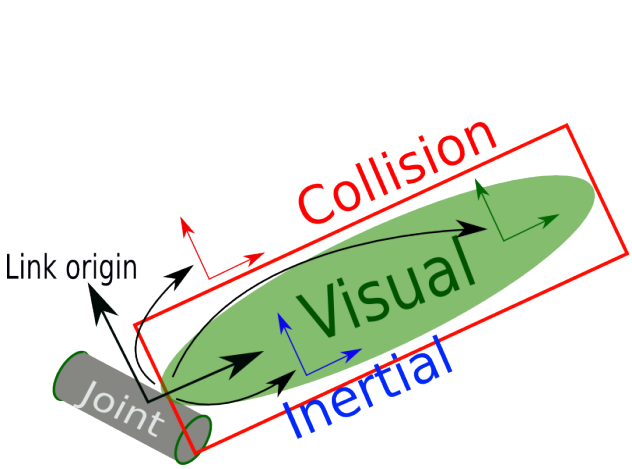
To help you understand and experiment with these models, you may choose one of the following options:

- **Copy and paste** the contents of the provided URDF files directly into new files within the `meshes` directory of your own C++ and Python packages.
- **Download or clone** the entire ROS 2 course repository and explore the models directly.

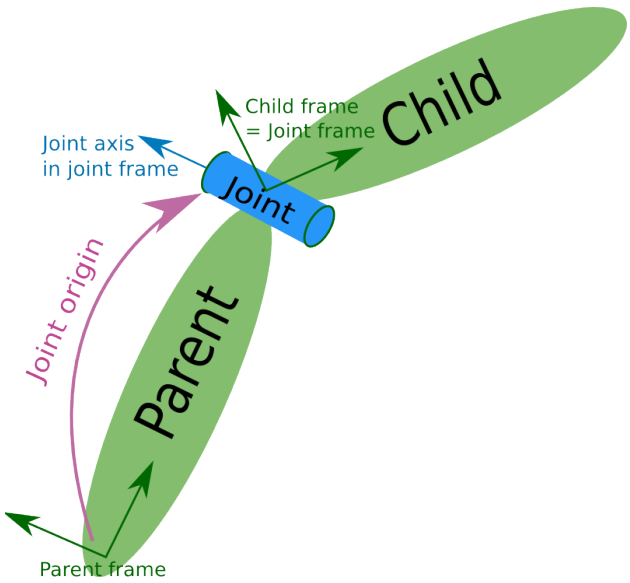
Each URDF model will be briefly described, and where appropriate, a preview of its structure will be provided. For larger files, only the most relevant sections will be highlighted, with explanations of components such as `link`, `joint`, `visual`, `inertial`, and `collision` elements.

At this stage, our main objective is to understand the geometric structure of the robot and how its parts are connected. In some models, we will incorporate articulation, as well as visual and physical properties.

We recommend becoming familiar with the `XML` specifications that define URDF modeling. For reference, consult the official documentation on the `<link>` element, which describes the kinematic and dynamic properties of a link, and the `<joint>` element, which defines the corresponding kinematic and dynamic properties of a joint, as illustrated in Figure 6.1.



(a) `<link>` element.



(b) `<joint>` element.

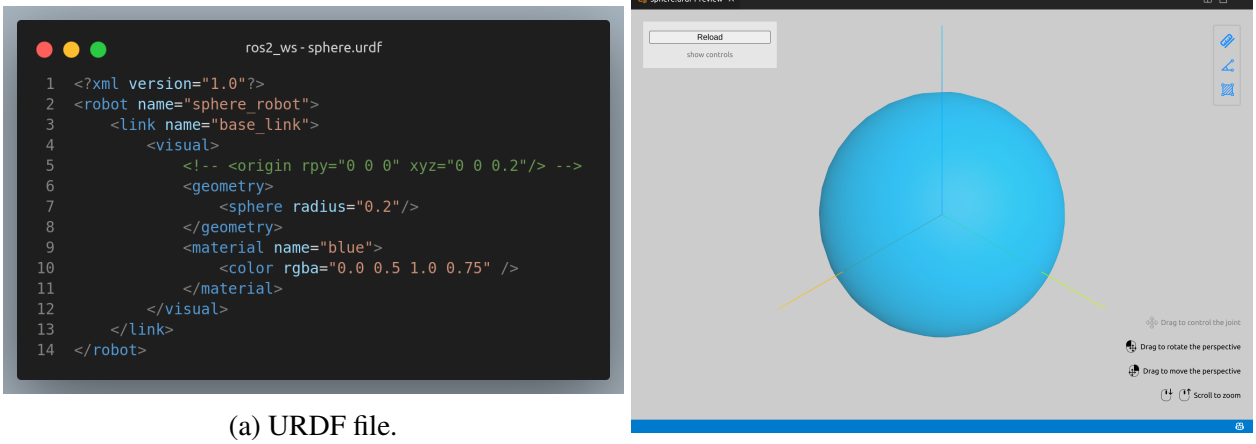
Figure 6.1: `XML` especifications for URDF modeling.

6.3.1 One Shape

We will begin by exploring some basic shapes using simple URDF models. These serve as straightforward examples to introduce key URDF concepts.

`sphere.urdf`

Source: src/s6_cpp_urdf/urdf/sphere.urdf or src/s6_py_urdf/urdf/sphere.urdf



(a) URDF file.

(b) URDF visualization.

Figure 6.2: URDF modeling of a sphere using the URDF development and visualization extensions.

In plain terms, the XML code defines a robot named `sphere_robot` with a single link called `base_link`. The link includes one visual element: a sphere with a radius of 0.2 m, as defined and displayed in Figure 6.2. Whereas the structure may seem verbose for such a simple model, it lays the foundation for more complex and articulated URDF models.

Key observations:

- The *fixed frame* refers to the coordinate frame used as the reference for the simulation environment. In this case, it is defined by the only link, `base_link`.
- By default, the origin of the sphere's geometry is at its center. As a result, half of the sphere is correctly placed below the grid plane in the visualization.

`cylinder.urdf`

Source: src/s6_cpp_urdf/urdf/cylinder.urdf or src/s6_py_urdf/urdf/cylinder.urdf

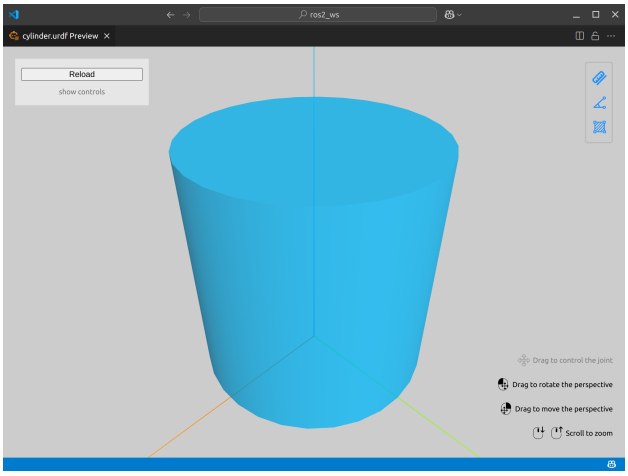
ros2_ws - cylinder.urdf

```xml
<?xml version="1.0"?>
<robot name="cylinder_robot">
    <link name="cylinder_base_footprint"/>

    <link name="cylinder_base_link">
        <visual>
            <origin rpy="0 0 0" xyz="0 0 0.0"/>
            <geometry>
                <cylinder radius="0.1" length="0.2"/>
            </geometry>
            <material name="blue">
                <color rgba="0.0 0.5 1.0 0.75" />
            </material>
        </visual>
    </link>

    <joint name="base_joint" type="fixed">
        <origin xyz="0.0 0.0 0.1" rpy="0.0 0.0 0.0"/>
        <parent link="cylinder_base_footprint"/>
        <child link="cylinder_base_link"/>
    </joint>
</robot>
```

(a) URDF file.



(b) URDF visualization.

Figure 6.3: URDF modeling of a cylinder using the URDF development and visualization extensions.

The XML code defines a robot named `cylinder_robot`, composed of a parent link called `cylinder_base_footprint` and a child link called `cylinder_base_link`. These are connected via a fixed joint named `base_joint`, which positions the child link's coordinate frame 0.1 m above that of the parent link. The robot contains a single visual element, a cylinder with a radius of 0.1 m and a height of 0.2 m, as defined and displayed in Figure 6.3.

Key observations:

- The *fixed frame* is the reference coordinate frame for the simulation environment. In this case, it is defined by the `cylinder_base_footprint` link.
- By default, the origin of the cylinder's geometry is at its base. As a result, the cylinder is correctly placed on the grid plane in the visualization.

wheel.urdf

Source: src/s6_cpp_urdf/urdf/wheel.urdf or src/s6_py_urdf/urdf/wheel.urdf

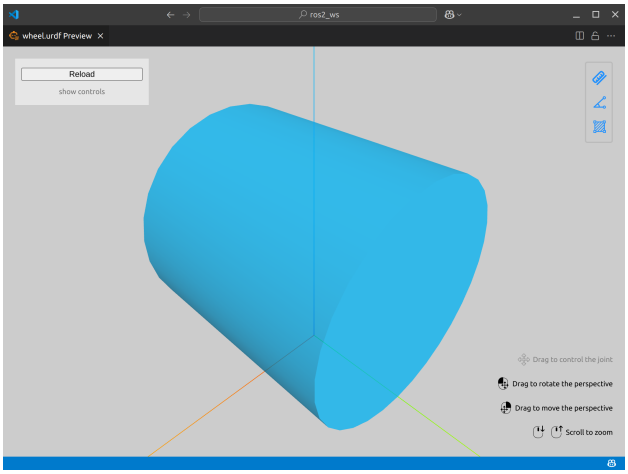
ros2_ws - wheel.urdf

```xml
<?xml version="1.0"?>
<robot name="wheel_robot">
    <link name="wheel_base_footprint"/>

    <link name="wheel_base_link">
        <visual>
            <origin rpy="0 0 0" xyz="0 0 0.0"/>
            <geometry>
                <cylinder radius="0.1" length="0.2"/>
            </geometry>
            <material name="blue">
                <color rgba="0.0 0.5 1.0 0.75" />
            </material>
        </visual>
    </link>

    <joint name="base_joint" type="continuous">
        <origin xyz="0.0 0.0 0.1" rpy="1.57079 0.0 0.0"/>
        <parent link="wheel_base_footprint"/>
        <child link="wheel_base_link"/>
        <axis xyz="0.0 0.0 1.0"/>
    </joint>
</robot>
```

(a) URDF file.



(b) URDF visualization.

Figure 6.4: URDF modeling of a wheel using the URDF development and visualization extensions.

The XML code defines a robot named wheel_robot, composed of a parent link called wheel_base_footprint and a child link called wheel_base_link. These are connected via a continuous joint named base_joint, which positions the child link's coordinate frame 0.1 m above that of the parent link. The child link is also rotated by $\frac{\pi}{2}$ radians around the x-axis (roll), aligning the cylinder's axis of symmetry with the x-axis. The joint allows continuous rotation about the z-axis of the child link, simulating the spinning behavior of a wheel. The visual element is a cylinder (representing the wheel) with a radius of 0.1 m and a height of 0.2 m, as defined and displayed in Figure 6.4.

Key observations:

- The *fixed frame* is the reference coordinate frame for the simulation environment. In this case, it is defined by the wheel_base_footprint link.
- By default, the wheel is visualized as a rotated cylinder, aligned horizontally due to the roll transformation. Its origin is positioned at the base of its geometry, so it appears correctly placed on the grid.

6.3.2 Multiple Shapes

Now let's explore how to add multiple shapes and links to form a more complex robot model.

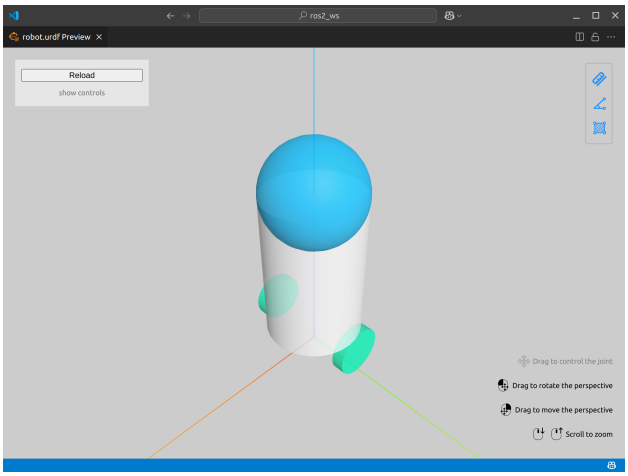
`robot.urdf`

Source: src/s6_cpp_urdf/urdf/robot.urdf or src/s6_py_urdf/urdf/robot.urdf

ros2_ws - robot.urdf

```
57      <joint name="base_joint" type="fixed">
58          <origin xyz="0.0 0.0 0.1" rpy="0.0 0.0 0.0"/>
59          <parent link="base_footprint"/>
60          <child link="base_link"/>
61      </joint>
62
63      <joint name="base_sphere" type="fixed">
64          <origin xyz="0.0 0.0 0.6" rpy="0 0 0"/>
65          <parent link="base_link"/>
66          <child link="sphere_link"/>
67      </joint>
68
69      <joint name="base_left_wheel_joint" type="continuous">
70          <origin xyz="0.0 0.225 0.0" rpy="0.0 0.0 0.0"/>
71          <parent link="base_link"/>
72          <child link="left_wheel"/>
73          <axis xyz="0.0 1.0 0.0"/>
74      </joint>
75
76      <joint name="base_right_wheel_joint" type="continuous">
77          <origin xyz="0.0 -0.225 0.0" rpy="0.0 0.0 0.0"/>
78          <parent link="base_link"/>
79          <child link="right_wheel"/>
80          <axis xyz="0.0 1.0 0.0"/>
81      </joint>
82  </robot>
```

(a) URDF file.



(b) URDF visualization.

Figure 6.5: URDF modeling of a robot using the URDF development and visualization extensions.

The `XML` code defines a robot named `my_robot`, composed of multiple links connected by both fixed and continuous joints. The main fixed joint, `base_joint`, positions the `base_link` 0.1 m above the `base_footprint` link. Additional joints connect the robot's body and its wheels, allowing continuous rotation of the wheels about their respective y-axis, thus simulating spinning behavior. The robot includes several visual elements: a blue sphere, a white cylinder, and two additional green cylinders representing the wheels, as defined and displayed in Figure 6.5.

Key observations:

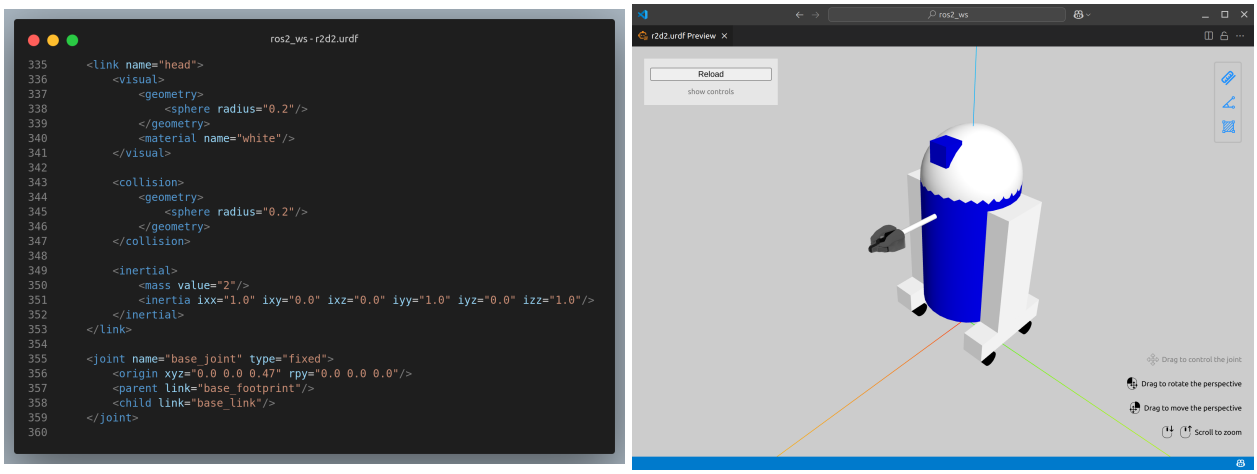
- The *fixed frame* is the reference coordinate frame for the simulation environment. In this case, it is defined by the `base_footprint` link.
- By default, the origin of the robot's geometry is placed at its base, ensuring that the entire robot model is correctly aligned with the grid plane in the visualization.

6.3.3 Physical Properties

Now, we will look at how to add some basic physical properties to your URDF model and how to specify its collision and inertial properties.

`r2d2.urdf`

Source: src/s6_cpp_urdf/urdf/r2d2.urdf or src/s6_py_urdf/urdf/r2d2.urdf



(a) URDF file.

(b) URDF visualization.

Figure 6.6: URDF modeling of a R2D2 using the URDF development and visualization extensions.

The `XML` code defines a robot named `r2d2_physics`, composed of multiple links connected by fixed, continuous, prismatic, and revolute joints. The main fixed joint, `base_joint`, positions the `base_link` 0.47 m above the `base_footprint` link. Additional joints connect the robot's body, wheels, and gripper, allowing continuous rotation of the wheels about their respective y-axes to simulate spinning behavior, and prismatic motion of the gripper along its respective x-axis. The R2D2 model includes several visual elements: a white sphere, blue and white boxes, and black cylinders representing the wheels, as defined and displayed in Figure 6.6.

Key observations:

- The *fixed frame* is the reference coordinate frame for the simulation environment. In this case, it is defined by the `base_footprint` link.
- By default, the origin of the R2D2's geometry is placed at its base, ensuring that the entire robot model is correctly aligned with the grid plane in the visualization.

`r2d2_2.urdf`

Source: src/s6_cpp_urdf/urdf/r2d2_2.urdf or src/s6_py_urdf/urdf/r2d2_2.urdf

ros2_ws - r2d2_2.urdf

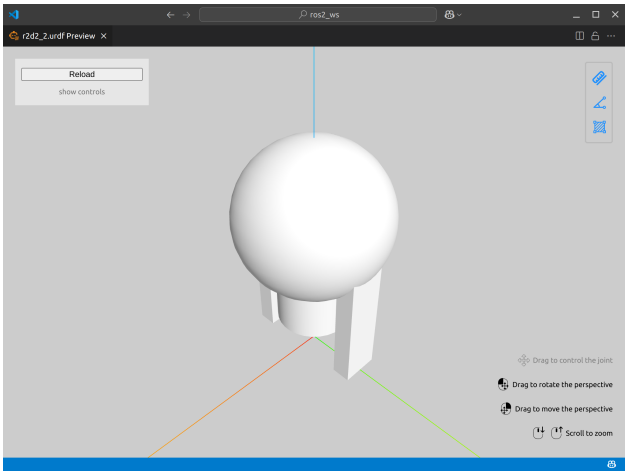
```xml
<?xml version="1.0"?>
<robot name="r2d2_2">
  <link name="base_footprint"/>

  <link name="axis">
    <visual>
      <origin xyz="0 0 0" rpy="1.57 0 0"/>
      <geometry>
        <cylinder radius="0.01" length=".5"/>
      </geometry>
      <material name="gray">
        <color rgba=".2 .2 .2 1"/>
      </material>
    </visual>

    <collision>
      <origin xyz="0 0 0" rpy="1.57 0 0"/>
      <geometry>
        <cylinder radius="0.01" length=".5"/>
      </geometry>
      <contact_coefficients mu="0" kp="1000.0" kd="1.0"/>
    </collision>
  </link>

  <joint name="base_joint" type="fixed">
    <origin xyz="0.0 0.0 0.7" rpy="0.0 0.0 0.0"/>
    <parent link="base_footprint"/>
    <child link="axis"/>
  </joint>
```

(a) URDF file.



(b) URDF visualization.

Figure 6.7: URDF modeling of a R2D2 using the URDF development and visualization extensions.

The `XML` code defines a robot named `r2d2_2`, composed of multiple links connected by fixed, continuous, prismatic, and revolute joints. The main fixed joint, `base_joint`, positions the `base_link` 0.7 m above the `base_footprint` link. Additional joints connect the robot's body and periscope, allowing prismatic motion of the periscope along its respective z-axis. The R2D2 model includes several visual elements: a sphere, boxes, and cylinders, as defined and displayed in Figure 6.7.

Key observations:

- The *fixed frame* is the reference coordinate frame for the simulation environment. In this case, it is defined by the `base_footprint` link.
- By default, the origin of the R2D2's geometry is placed at its base, ensuring that the entire robot model is correctly aligned with the grid plane in the visualization.

6.3.4 Meshes

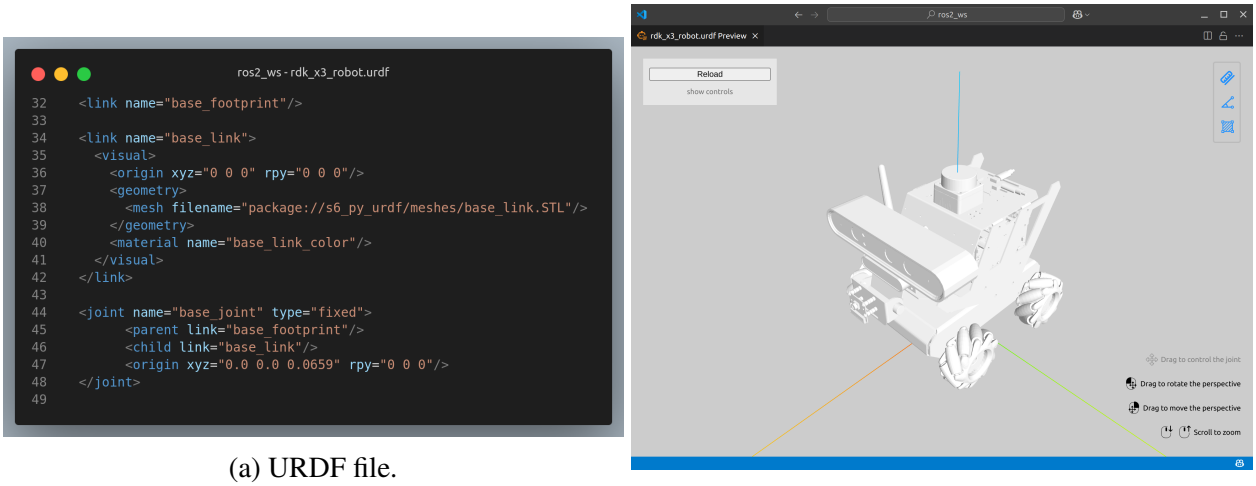
Meshes are separate files whose paths must be explicitly specified using the `package://NAME_OF_PACKAGE/path` notation. The meshes referenced in the previous `r2d2_2.urdf` file are located within the `urdf_tutorial` package, in a folder named `meshes`. Refer to the Appendix C to load the mesh files to correctly visualize the URDF models.

Meshes can be imported in various formats. `STL` is quite common, but the engine also supports `DAE` files, which can include embedded color data, eliminating the need to specify colors or materials separately. Often, these are provided as separate files. Meshes can be resized using relative scaling parameters or a bounding box size. Additionally, it is possible to reference meshes from entirely different packages.

Next, we will explore the URDF model for our challenge project robot: the `RDKX3Robot`. Refer to the Appendix C to load the mesh files to correctly visualize the URDF model.

`rdk_x3_robot.urdf`

Source: src/s6_cpp_urdf/urdf/rdk_x3_robot.urdf or src/s6_py_urdf/urdf/rdk_x3_robot.urdf



(a) URDF file.

(b) URDF visualization.

Figure 6.8: URDF modeling of the RDK X3 Robot using URDF development and visualization extensions.

The `XML` code defines a robot named `rdk_x3_robot`, composed of multiple links connected by fixed, continuous, and revolute joints. The main fixed joint, `base_joint`, positions the `base_link` 0.0659 m above the `base_footprint` link. Additional joints connect the robot's body and wheels, allowing continuous rotation of the wheels about their respective y-axes to simulate spinning behavior. The RDK X3 model includes several visual elements, all of them corresponding to `STL` files, as defined and displayed in Figure 6.8.

Key observations:

- The *fixed frame* is the reference coordinate frame for the simulation environment. In this case, it is defined by the `base_footprint` link.

- By default, the origin of the R2D2's geometry is placed at its base, ensuring that the entire robot model is correctly aligned with the grid plane in the visualization.

6.4 CMakeLists.txt

To make accessible the launch (.py), mesh (.STL and .dae), RViz (.rviz), and URDF (.urdf) files we have defined in the C++ package, add the following lines to CMakeLists.txt:

```
# Added manually
install(DIRECTORY
  launch
  meshes
  rviz
  urdf
  DESTINATION share/${PROJECT_NAME}
)
```

6.5 setup.py

To make accessible the launch (.py), mesh (.STL and .dae), RViz (.rviz), and URDF (.urdf) files we have defined in the Python package, add the data files while preserving any existing file setup, with the following lines to setup.py:

```
data_files=[
    ('share/ament_index/resource_index/packages',
        ['resource/' + package_name]),
    ('share/' + package_name, ['package.xml']),
    (os.path.join('share', package_name, 'meshes'), glob('meshes/*')),
    (os.path.join('share', package_name, 'urdf'), glob('urdf/*')),
    (os.path.join('share', package_name, 'rviz'), glob('rviz/*')),
    (os.path.join('share', package_name, 'launch'), glob('launch/*'))
],
```

6.6 package.xml

Make sure the required dependencies are defined within the package.xml files. For the C++ package:

```
    <depend>rclcpp</depend>
    <depend>geometry_msgs</depend>
    <depend>sensor_msgs</depend>
    <depend>tf2_ros</depend>
    <depend>tf2_geometry_msgs</depend>
```

And for the Python package:

```
    <depend>rclpy</depend>
    <depend>geometry_msgs</depend>
    <depend>sensor_msgs</depend>
    <depend>tf2_ros</depend>
    <depend>tf_transformations</depend>
```

6.7 Visualizing URDF (.urdf) Files with RViz

In addition to the URDF visualizer extension introduced in the Section 6.3, we can also use the ROS 2 environment to observe URDF models through RViz (ROS Visualization). To visualize a URDF model, launch the `display.launch.py` file from the `urdf_tutorial` package using the following command:

```
ros2launchurdf_tutorialdisplay.launch.pymodel:=<file_path>/<urdf_file>.urdf
```

This command performs the following actions:

- Loads the specified model and sets it as a parameter for the `robot_state_publisher` node.
- Launches nodes that publish `sensor_msgs/msg/JointState` and coordinate transforms.
- Starts RViz with a predefined configuration file.

After executing `display.launch.py`, RViz will display a grid, a fixed (main) coordinate frame, by default set to `base_link`, and the loaded URDF model.
Key observations:

- The *fixed frame* is the reference coordinate frame for the simulation environment. In most cases, this should be updated to `base_footprint`.
- When using this fixed frame, the origin of most URDF models aligns with their base, ensuring the robot model is correctly positioned on the grid plane in RViz.

6.7.1 Visualizing URDF Models with the `urdf_tutorial` Package

After saving your changes, navigate to the root of your workspace (e.g., `/ros2_ws`), build the relevant packages, and source the workspace:

```
$ colcon build --packages-select s6_cpp_urdf
$ colcon build --packages-select s6_py_urdf
$ source install/setup.bash
```

Then, launch `display.launch.py` from the `urdf_tutorial` package to load the URDF models from either the C++ or Python package.
Note: Ensure that your workspace is properly sourced, as `display.launch.py` requires access to the mesh (`.dae` and `.STL`) files used by the `r2d2.urdf` and `rdk_x3_robot.urdf` models, respectively.

sphere.urdf

To visualize the sphere.urdf model from either the C++ or Python package, use the following command:

```
$ ros2 launch urdf_tutorial display.launch.py model:=<file_path>/sphere.urdf
```

Replace <file_path> with the absolute path to the sphere.urdf file. Figure 6.9 shows the simulation of this model in RViz.

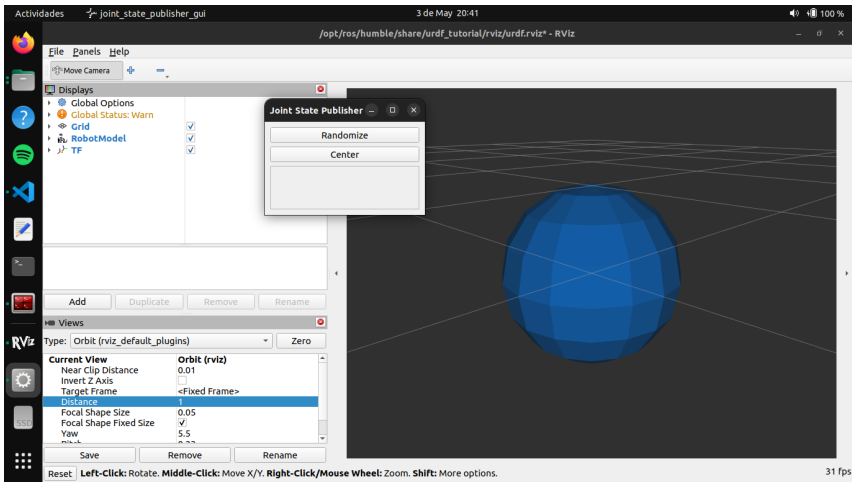


Figure 6.9: Visualization of the sphere using RViz.

cylinder.urdf

To visualize the cylinder.urdf model from either the C++ or Python package, use the following command:

```
$ ros2 launch urdf_tutorial display.launch.py model:=<file_path>/cylinder.urdf
```

Replace <file_path> with the absolute path to the cylinder.urdf file. Figure 6.10 shows the simulation of this model in RViz.

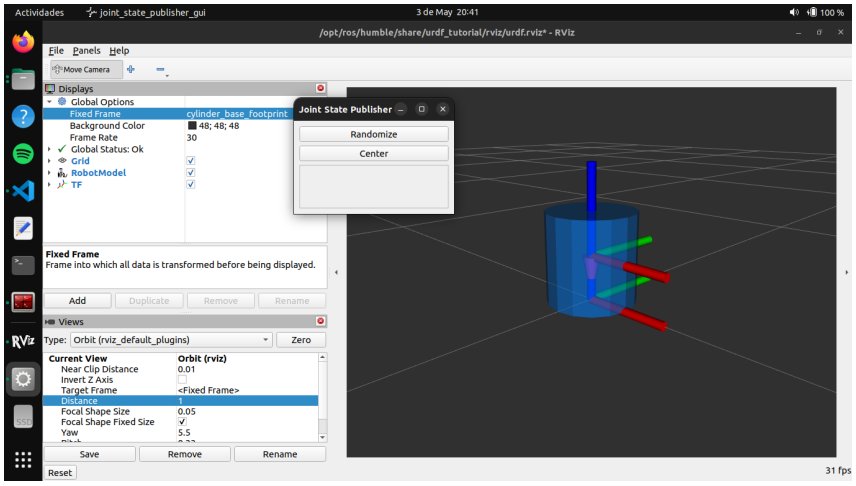


Figure 6.10: Visualization of the cylinder using RViz.

wheel.urdf

To visualize the wheel.urdf model from either the C++ or Python package, use the following command:

```
$ ros2 launch urdf_tutorial display.launch.py model:=<file_path>/wheel.urdf
```

Replace <file_path> with the absolute path to the wheel.urdf file. Figure 6.11 shows the simulation of this model in RViz.

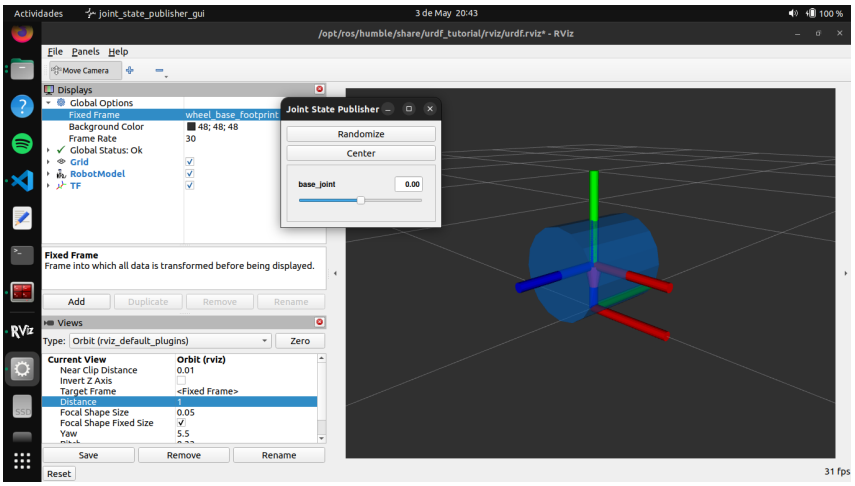


Figure 6.11: Visualization of the wheel using RViz.

robot.urdf

To visualize the robot.urdf model from either the C++ or Python package, use the following command:

```
$ ros2 launch urdf_tutorial display.launch.py model:=<file_path>/robot.urdf
```

Replace <file_path> with the absolute path to the robot.urdf file. Figure 6.12 shows the simulation of this model in RViz.

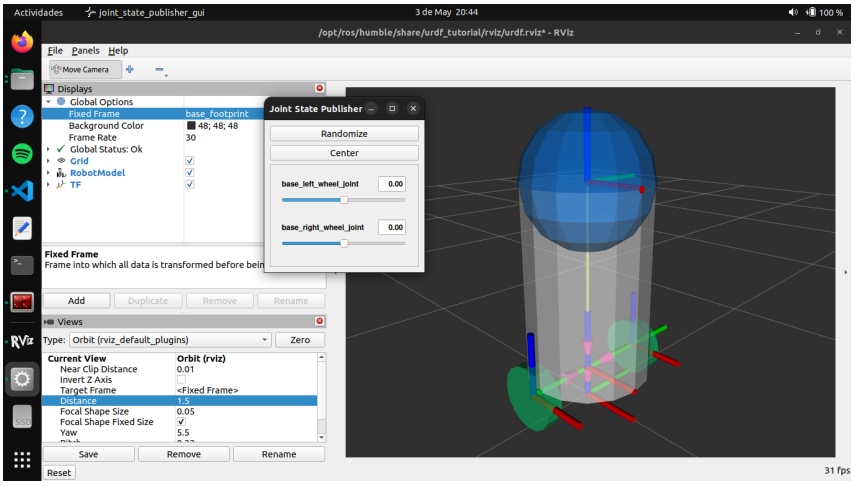


Figure 6.12: Visualization of the robot using RViz.

r2d2.urdf

To visualize the r2d2.urdf model from either the C++ or Python package, use the following command:

```
$ ros2 launch urdf_tutorial display.launch.py model:=<file_path>/r2d2.urdf
```

Replace <file_path> with the absolute path to the r2d2.urdf file. Figure 6.13 shows the simulation of this model in RViz.

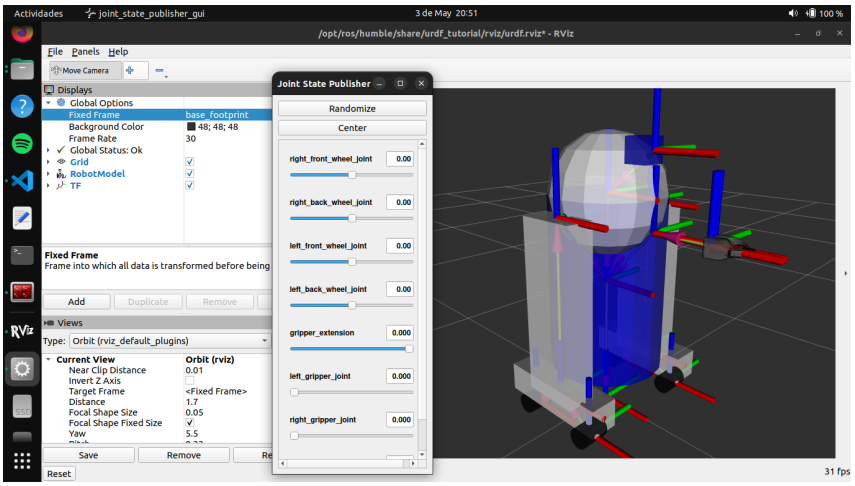


Figure 6.13: Visualization of the R2D2 using RViz.

r2d2_2.urdf

To visualize the r2d2_2.urdf model from either the C++ or Python package, use the following command:

```
$ ros2 launch urdf_tutorial display.launch.py model:=<file_path>/r2d2\_2.urdf
```

Replace <file_path> with the absolute path to the r2d2_2.urdf file. Figure 6.14 shows the simulation of this model in RViz.

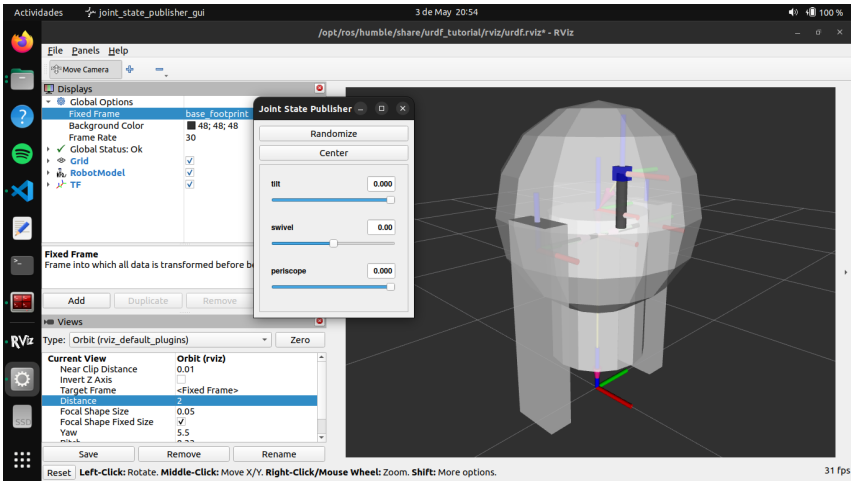


Figure 6.14: Visualization of the R2D2_2 using RViz.

rdk_x3_robot.urdf

To visualize the rdk_x3_robot.urdf model from either the C++ or Python package, use the following command:

```
$ ros2 launch urdf_tutorial display.launch.py model:=<file_path>/rdk\_x3\_robot.urdf
```

Replace <file_path> with the absolute path to the rdk_x3_robot.urdf file. Figure 6.15 shows the simulation of this model in RViz.

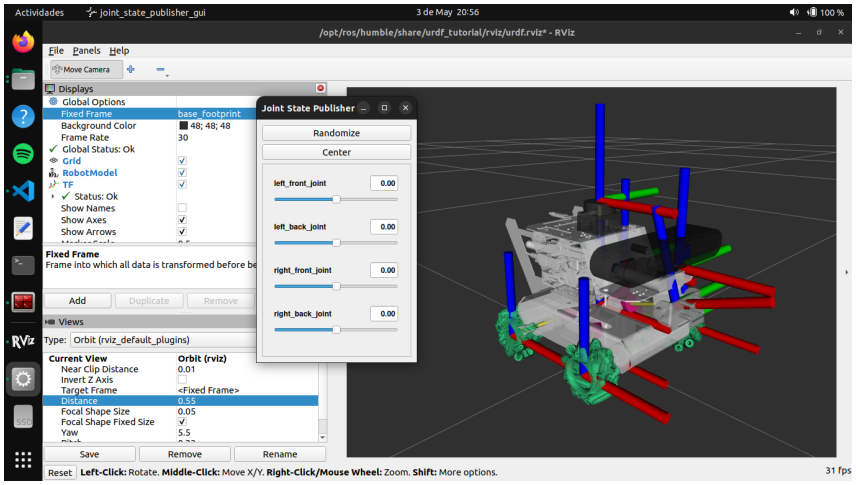


Figure 6.15: Visualization of the RDK X3 Robot using RViz.

6.8 Using URDF with robot_state_publisher in C++

In this section, we cover how to setting up the robot_state_publisher node in C++ by working with the rdk_x3_robot.cpp file.

The corresponding C++ nodes for the cylinder, wheel, and R2D2 models are also available in our ROS 2 course GitHub repository.

6.8.1 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/s6_cpp_urdf/src
```

Replace s6_cpp_urdf with your actual package name.

Creating the C++ Node File

Create a new C++ file for your state publisher node:

```
$ touch rdk_x3_robot.cpp
```

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

```
$ ls
```

You should now see the `rdk_x3_robot.cpp` file.

6.8.2 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 `rdk_x3_robot.cpp` file for editing.

Implementing the Node Code

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

State Publisher in C++

```
#include <rclcpp/rclcpp.hpp>
#include <geometry_msgs/msg/quaternion.hpp>
#include <sensor_msgs/msg/joint_state.hpp>
#include <tf2_ros/transform_broadcaster.h>
#include <tf2_geometry_msgs/tf2_geometry_msgs.hpp>
#include <cmath>
#include <thread>
#include <chrono>

using namespace std::chrono;

class StatePublisher : public rclcpp::Node{
    public:
        StatePublisher(rclcpp::NodeOptions options=rclcpp::NodeOptions()):
            Node("state_publisher_node",options){
                RCLCPP_INFO(this->get_logger(),"C++ state publisher node has been
                ↪  started");

                // Create a publisher to tell robot_state_publisher the JointState
                ↪  information.
                // robot_state_publisher will deal with this transformation
                joint_pub_ =
                ↪  this->create_publisher<sensor_msgs::msg::JointState>("joint_states",10);
```

```
                // Create a broadcaster to publish the transform between coordinate
                ↪  frames that
                // will determine the position of coordinate system 'base_footprint' in
                ↪  coordinate system 'odom'
                broadcaster = std::make_shared<tf2_ros::TransformBroadcaster>(this);

                // Timer for publishing at ~30Hz

                ↪  timer_=this->create_wall_timer(33ms,std::bind(&StatePublisher::publish,this));
            }

        void publish();

    private:
        rclcpp::Publisher<sensor_msgs::msg::JointState>::SharedPtr joint_pub_;
        std::shared_ptr<tf2_ros::TransformBroadcaster> broadcaster;
        rclcpp::TimerBase::SharedPtr timer_;

    // State variables
    // degree means one degree
    const double degree = M_PI/180.0;
    double wheel_angle = 0.0;
    double wheel_step = degree;
    double angle = 0.0;
};

void StatePublisher::publish(){
    // Create JointState message
    sensor_msgs::msg::JointState joint_state;
    // Add time stamp
    joint_state.header.stamp=this->get_clock()->now();
    // Specify joints' name which are defined in the URDF file and their content
    joint_state.name={"left_front_joint","left_back_joint",
                      "right_front_joint","right_back_joint"};
    joint_state.position={wheel_angle,wheel_angle,-wheel_angle,-wheel_angle};

    // Create TransformStamped message
    geometry_msgs::msg::TransformStamped t;
    // Add time stamp
    t.header.stamp=this->get_clock()->now();
    // Specify the father and child frames
    // odom is the base coordinate system of tf2
    t.header.frame_id="odom";
    // base_footprint is defined in URDF file and it is the base coordinate of model
    t.child_frame_id="base_footprint";

    // Add translation change
    t.transform.translation.x = cos(angle) * 1.0;
    t.transform.translation.y = sin(angle) * 1.0;
    t.transform.translation.z = 0.0;
```

```
    // Euler angle into Quaternion and add rotation change
    tf2::Quaternion q;
    q.setRPY(0,0,angle + M_PI / 2);
    t.transform.rotation.x = q.x();
    t.transform.rotation.y = q.y();
    t.transform.rotation.z = q.z();
    t.transform.rotation.w = q.w();

    // Update state for next cycle
    wheel_angle += wheel_step * 4;
    if (wheel_angle<-M_PI || wheel_angle>M_PI){
        wheel_angle *= -1;
    }

    angle += degree; // Change the angle at a slow pace

    // Publish messages
    joint_pub_->publish(joint_state);
    broadcaster->sendTransform(t);

    RCLCPP_INFO(this->get_logger(),"Publishing joint state and transform");
}

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

6.8.3 Integrating the C++ State Publisher Node into the Package

Modify `CMakeLists.txt` of your C++ package (`s6_cpp_urdf`) to add an executable for the image publisher node(s):

```
# Added manually
add_executable(cylinder_exe src/cylinder.cpp)
ament_target_dependencies(cylinder_exe rclcpp geometry_msgs sensor_msgs tf2_ros
↪ tf2_geometry_msgs)

add_executable(wheel_exe src/wheel.cpp)
ament_target_dependencies(wheel_exe rclcpp geometry_msgs sensor_msgs tf2_ros
↪ tf2_geometry_msgs)

add_executable(r2d2_exe src/r2d2.cpp)
ament_target_dependencies(r2d2_exe rclcpp geometry_msgs sensor_msgs tf2_ros
↪ tf2_geometry_msgs)

add_executable(rdk_x3_robot_exe src/rdk_x3_robot.cpp)
ament_target_dependencies(rdk_x3_robot_exe rclcpp geometry_msgs sensor_msgs tf2_ros
↪ tf2_geometry_msgs)
```

```
# Added manually
install(TARGETS
  cylinder_exe
  wheel_exe
  r2d2_exe
  rdk_x3_robot_exe
  DESTINATION lib/${PROJECT_NAME}
)

# Added manually
install(DIRECTORY
  launch
  meshes
  rviz
  urdf
  DESTINATION share/${PROJECT_NAME}/
)
```

6.8.4 Summary of C++ State Publisher Node Key Identifiers

- `rdk_x3_robot.cpp`: Name of the source file for the C++ state publisher node.
- `cpp_state_publisher_node`: Name of the C++ state publisher node as defined in the constructor of the C++ class.
- `rdk_x3_robot_exe`: Name of the executable defined in `CMakeLists.txt`.
- `joint_states`: Name of the topic provided by the C++ state publisher node.

6.9 Using URDF with `robot_state_publisher` in Python

In this section, we cover how to setting up the `robot_state_publisher` node in Python by working with the `rdk_x3_robot.py` file.

The corresponding Python nodes for the cylinder, wheel, and R2D2 models are also available in our ROS 2 course GitHub repository.

6.9.1 Setting Up the Python Node

Navigating to the Package Directory

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

```
$ cd ros2_ws/src/s6_py_urdf/s6_py_urdf
```

Replace `s6_py_urdf` with your actual package name.

Creating the Python Node File

Create a new Python file for your state publisher node:

```
$ touch rdk_x3_robot.py
```

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

```
$ ls
```

You should now see the `rdk_x3_robot.py` file.

6.9.2 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 `rdk_x3_robot.py` file for editing.

Implementing the Node Code

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

State Publisher in Python

```python
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import TransformStamped
from sensor_msgs.msg import JointState
from tf2_ros import TransformBroadcaster
import tf_transformations
import math

class StatePublisher(Node):
    def __init__(self):
        super().__init__('py_state_publisher_node')
        self.get_logger().info("Python state publisher node has been started")

        # Create a publisher to tell robot_state_publisher the JointState information.
        # robot_state_publisher will deal with this transformation
        self.joint_pub = self.create_publisher(JointState, 'joint_states', 10)

        # Create a broadcaster to publish the transform between coordinate frames that
        # will determine the position of coordinate system 'base_footprint' in
        ↪  coordinate system 'odom'
        self.broadcaster = TransformBroadcaster(self)

        # Timer for publishing at ~30Hz
        self.timer = self.create_timer(0.033, self.publish)
```

```
        # State variables
        # degree means one degree
        self.degree = math.pi / 180.0
        self.wheel_angle = 0.0
        self.wheel_step = self.degree
        self.angle = 0.0

    def publish(self):
        # Create JointState message
        joint_state = JointState()
        # Add time stamp
        joint_state.header.stamp = self.get_clock().now().to_msg()
        # Specify joints' name which are defined in the URDF file and their content
        joint_state.name =
        ↪  ['left_front_joint','left_back_joint','right_front_joint','right_back_joint']
        joint_state.position =
        ↪  [self.wheel_angle,self.wheel_angle,-self.wheel_angle,-self.wheel_angle]

        # Create TransformStamped message
        t = TransformStamped()
        # Add time stamp
        t.header.stamp = self.get_clock().now().to_msg()
        # Specify the father and child frames
        # odom is the base coordinate system of tf2
        t.header.frame_id = 'odom'
        # base_footprint is defined in URDF file and it is the base coordinate of model
        t.child_frame_id = 'base_footprint'

        # Add translation change
        t.transform.translation.x = math.cos(self.angle) * 1.0
        t.transform.translation.y = math.sin(self.angle) * 1.0
        t.transform.translation.z = 0.0

        # Euler angle into Quaternion and add rotation change
        q = tf_transformations.quaternion_from_euler(0, 0, self.angle + math.pi / 2)
        t.transform.rotation.x = q[0]
        t.transform.rotation.y = q[1]
        t.transform.rotation.z = q[2]
        t.transform.rotation.w = q[3]

        # Update state for next cycle
        self.wheel_angle += self.wheel_step * 4
        if self.wheel_angle < -math.pi or self.wheel_angle > math.pi:
            self.wheel_angle *= -1

        self.angle += self.degree # Change the angle at a slow pace

        # Publish messages
        self.joint_pub.publish(joint_state)
        self.broadcaster.sendTransform(t)

        self.get_logger().info("Publishing joint state and transform")
```

```
def main(args=None):
    rclpy.init(args=args)
    node = StatePublisher()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

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

6.9.3 Integrating the Python State Publisher Node into the Package

Modify `setup.py` of your Python package (`s6_py_urdf`) to add an executable for the image publisher node(s):

```
entry_points={
    'console_scripts': [
        "cylinder_exe = s6_py_urdf.cylinder:main",
        "wheel_exe = s6_py_urdf.wheel:main",
        "r2d2_exe = s6_py_urdf.r2d2:main",
        "rdk_x3_robot_exe = s6_py_urdf.rdk_x3_robot:main"
    ],
},
```

6.9.4 Summary of Python State Publisher Node Key Identifiers

- `rdk_x3_robot.py`: Name of the source file for the Python state publisher node.
- `py_state_publisher_node`: Name of the Python state publisher node as defined in the constructor of the Python class.
- `rdk_x3_robot_exe`: Name of the executable defined in `setup.py`.
- `joint_states`: Name of the topic provided by the Python state publisher node.

6.10 Launch File Definition

Within the `s6_cpp_urdf/launch` and `s6_py_urdf/launch` directories, create (or open) a file named `rdk_x3_robot_launch.py` with the following structure. For the C++ package:

```
import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    # Path to the URDF file
    urdf_file = os.path.join(
        get_package_share_directory('s6_cpp_urdf'),
        'urdf',
        'rdk_x3_robot.urdf'
    )

    # open the whole urdf_file_name file and read it content to robot_desc
    with open(urdf_file, 'r') as infp:
        robot_desc = infp.read()

    return LaunchDescription([
        Node(
            package='robot_state_publisher',
            executable='robot_state_publisher',
            name='robot_state_publisher',
            output='screen',
            parameters=[{'robot_description': robot_desc}],
            arguments=[urdf_file]
        ),
        Node(
            package='s6_cpp_urdf',
            executable='rdk_x3_robot_exe',
            name='rdk_x3_robot',
            output='screen'
        ),
    ])
```

And for the Python package:

```
import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    # Path to the URDF file
    urdf_file = os.path.join(
        get_package_share_directory('s6_py_urdf'),
        'urdf',
        'rdk_x3_robot.urdf'
    )
```

```
    # open the whole urdf_file_name file and read it content to robot_desc
    with open(urdf_file, 'r') as infp:
        robot_desc = infp.read()

    return LaunchDescription([
        Node(
            package='robot_state_publisher',
            executable='robot_state_publisher',
            name='robot_state_publisher',
            output='screen',
            parameters=[{'robot_description': robot_desc}],
            arguments=[urdf_file]
        ),
        Node(
            package='s6_py_urdf',
            executable='rdk_x3_robot_exe',
            name='rdk_x3_robot',
            output='screen'
        ),
    ])
```

These launch files start two nodes:

- The `robot_state_publisher` node, which publishes the URDF model and its transformations.
- The `rdk_x3_robot` node, responsible for publishing the joint state values and the overall robot geometry.

6.11 Building the Packages and Running the Nodes

After making the above edits and saving all the changes, go to the root of your workspace (e.g., `/ros2_ws`) and build the packages:

```
$ colcon build --packages-select s6_cpp_urdf
$ colcon build --packages-select s6_py_urdf
```

Then, in two terminals, source the workspace environment in each, and proceed as follows:

- In **Terminal 1**, launch either the C++ or Python robot state publisher nodes:

  ```
  $ source install/setup.bash
  $ ros2 launch s6_py_urdf rdk_x3_robot_launch.py
  ```

- In **Terminal 2**, start RViz with the configuration file stored in the respective package:

  ```
  $ source install/setup.bash
  $ rviz2 -d src/s6_py_urdf/rviz/rdk_x3_robot.rviz
  ```

Additionally, open two terminals for diagnostics and introspection tools:

- **Terminal 3:** Use the `view_frames` tool from the `tf2_tools` package to visualize the current coordinate frame hierarchy (tree):

  ```
  $ ros2 run tf2_tools view_frames
  ```

- **Terminal 4:** Launch the `rqt_graph` tool to view the current ROS 2 node and topic graph:

```
$ rqt_graph
```

6.12 Practice Assignment: URDF Modeling and Launch Files (Robot States)

In this assignment, you will run the ROS 2 robot state publisher node implemented in both C++ and Python, as described in Sections 6.8 and 6.9. Capture evidence of successful execution using two screenshots as PNG files and one PDF file, and submit them in the designated assignment area on Canvas (within the ROS 2 module). The filenames must follow this format:

`FirstnameLastname_evidence_sX_Y.png`

where `sX` corresponds to the session number, and `Y` is the evidence number. For example, correct filenames would be:

`EduardoDavila_evidence_s6_1.png`,
`EduardoDavila_evidence_s6_2.png`,
`EduardoDavila_evidence_s6_3.pdf`.

6.12.1 Executing ROS 2 Robot State Publishers

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 s6_cpp_urdf
$ colcon build --packages-select s6_py_urdf
```

Then, in two terminals, source the workspace environment in each, and proceed as follows:

- In **Terminal 1**, launch either the C++ or Python robot state publisher nodes:

```
$ source install/setup.bash
$ ros2 launch s6_py_urdf rdk_x3_robot_launch.py
```

- In **Terminal 2**, start RViz with the configuration file stored in the respective package:

```
$ source install/setup.bash
$ rviz2 -d src/s6_py_urdf/rviz/rdk_x3_robot.rviz
```

You should observe that:

1. The `robot_state_publisher` node publishes the URDF model and its coordinate transforms for visualization in RViz.
2. The `rdk_x3_robot` node publishes the joint state values, enabling dynamic updates of the model in RViz.

6.12.2 Visualizing the frames with `view_frames`

To observe the current coordinate frame hierarchy, use the `image_frames` tool from the `tf2_tools` package. Then, in **Terminal 3**, execute the following command (without the need to enter the workspace directory):

```
$ ros2 run tf2_tools view_frames
```

This tool will generate a PDF file in the current directory of the terminal. The file provides a graphical representation of the current coordinate frame tree published by the `robot_state_publisher` node from the robot URDF model.

6.12.3 Visualizing the ROS Graph with `rqt_graph`

To verify that the robot state nodes are correctly connected, use the `rqt_graph` tool. Then, in **Terminal 4**, execute the following command (without the need to enter the workspace directory):

```
$ ros2 run rqt_graph rqt_graph
```

or simply:

```
$ rqt_graph
```

This will visualize the ROS 2 graph, showing the `joint_states` topic connecting the robot state publisher nodes, as described above.

6.12.4 Submission Instructions

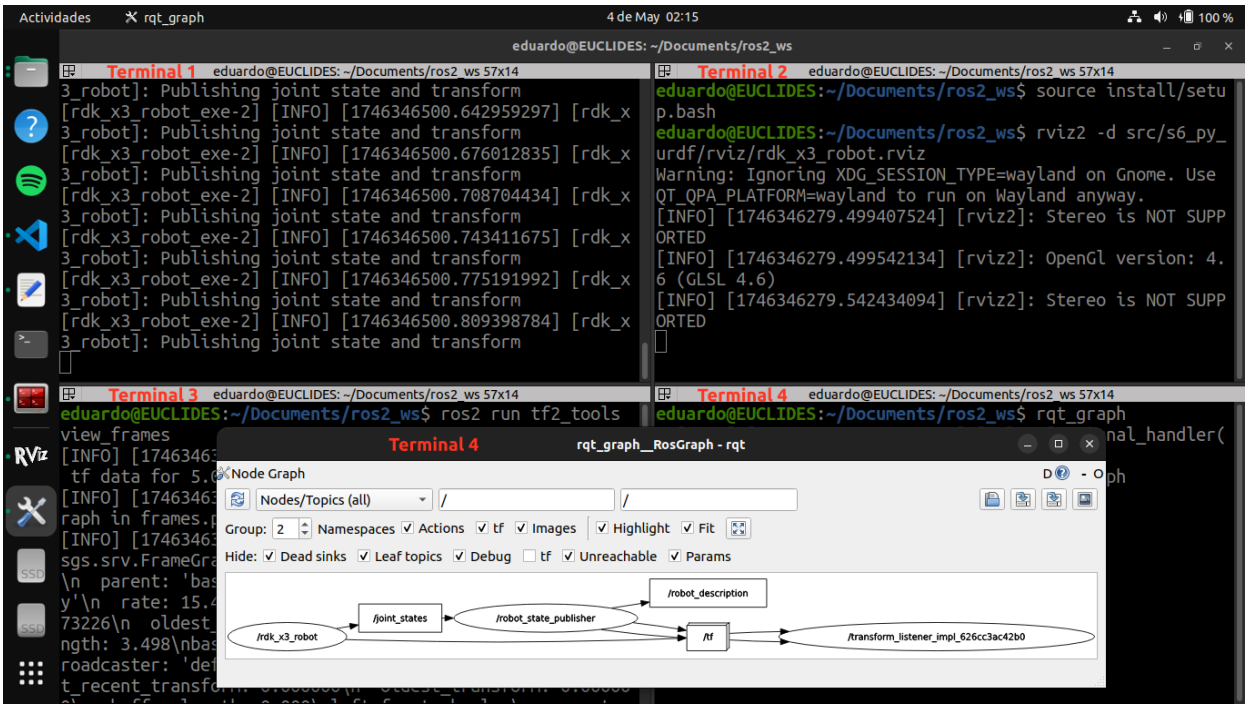


Figure 6.16: Example screenshot showing terminal outputs and the ROS 2 graph of the robot state publishers in action.

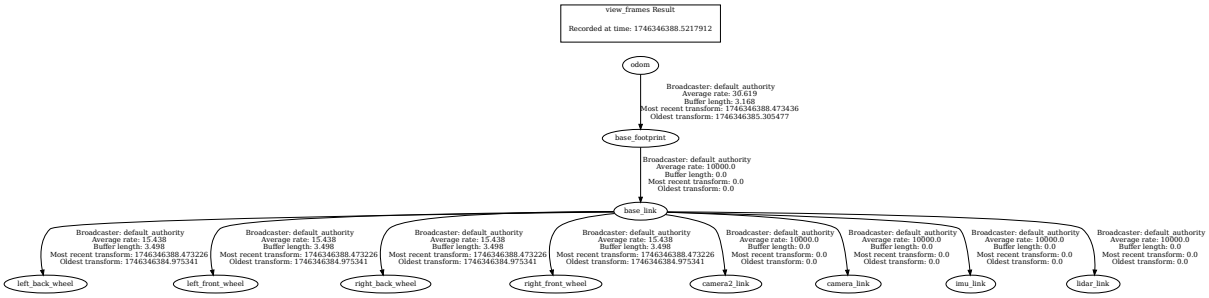


Figure 6.17: PDF showing the graphical representation of the current coordinate frame tree.

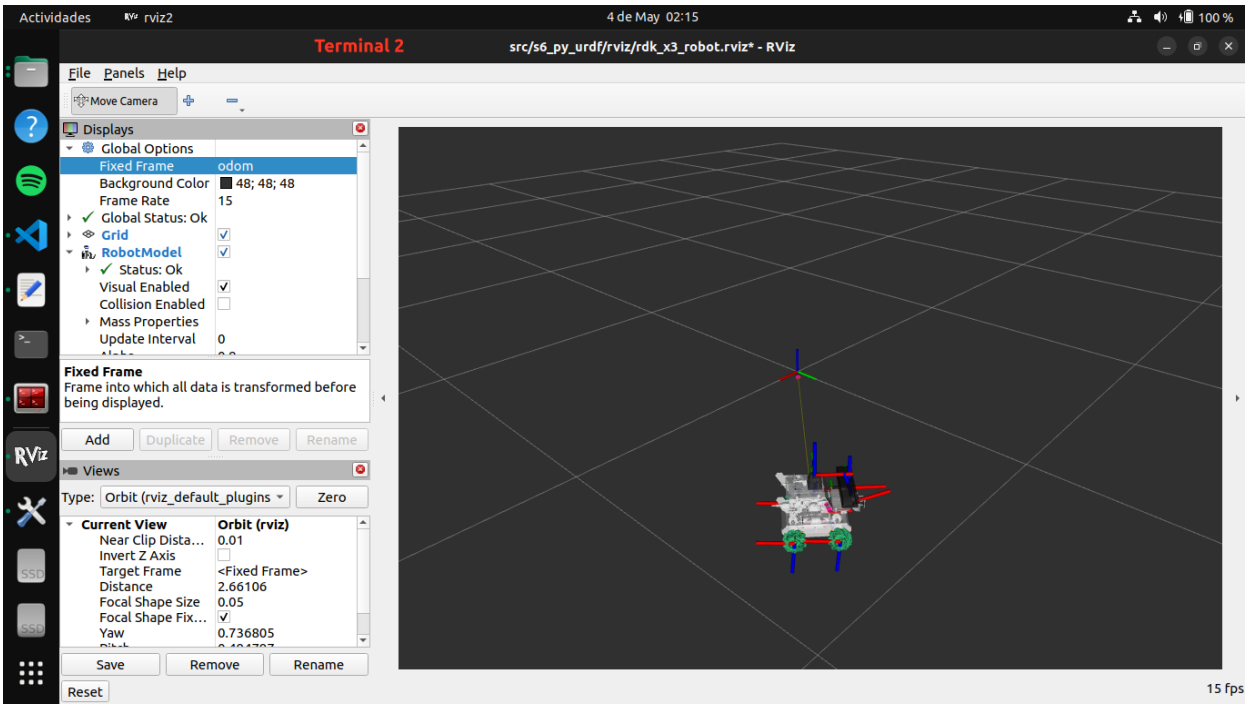


Figure 6.18: Example screenshot showing the simulation of the RDK X3 Robot in RViz.

1. Run the ROS 2 robot state publisher nodes as described above.
2. As first evidence, capture a **screenshot showing the terminal outputs and the ROS 2 graph** from either the C++ or Python nodes, as described in Sections 6.12.1 and 6.12.3. See Figure 6.16 for reference.
3. As second evidence, locate the PDF file generated by the `view_frames` tool from the `tf2_tools` package, as described in Section 6.12.2. See Figure 6.17 for reference.
4. As third evidence, capture a **screenshot showing the simulation of the RDK X3 Robot in RViz**. See Figure 6.18 for reference.
5. Save the screenshots as PNG files.
6. Name the files following this format: `FirstnameLastname_evidence_sX_Y.png` (replace `X` with the session number, and `Y` with the evidence number).
7. Submit the files as specified by the course guidelines on Canvas.

Note: Your machine's username and hostname must be visible in the screenshots to verify the authenticity of your submission, as shown in Figure 6.16. Any submission that appears copied, unclear, or altered will be considered invalid and may receive a score of 0.

CHAPTER 7

tf2 Library (Robot Network)

Author: Dr. Eduardo de Jesús Dávila Meza.
EduardoDavila-AI-PhD

Abstract - All the source files referenced in this chapter are available in our ROS 2 course GitHub repository, located in the `ros2_ws/src` folder, corresponding to the ROS 2 packages with the `s7` prefix.

7.1 Introduction

`tf2` is the transform library, which lets the user keep **track of multiple coordinate frames over time**. `tf2` maintains the relationship between coordinate frames in a tree structure, as shown in Figure 7.1, buffered in time and lets the user transform points, vectors, etc., between any two coordinate frames at any desired point in time.

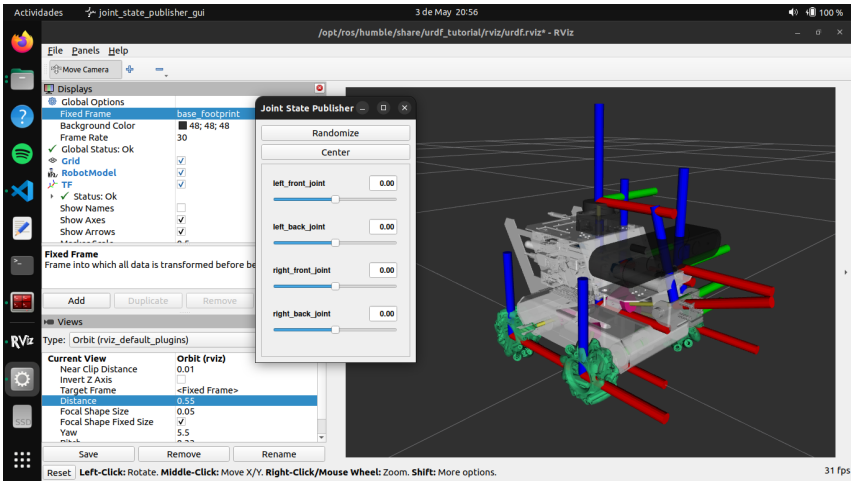


Figure 7.1: Visualization of joint states and coordinate frames of the RDK X3 Robot in RViz using `display.launch.py` from the `urdf_tutorial` package.

7.1.1 Properties of `tf2`

A robotic system typically has many 3D coordinate frames that change over time, such as a world frame, base frame, gripper frame, head frame, etc. tf2 keeps track of all these frames over time, and allows you to ask questions like:

- Where was the head frame relative to the world frame 5 seconds ago?
- What is the pose of the object in my gripper relative to my base?
- What is the current pose of the base frame in the map frame?

7.2 Prerequisites

7.2.1 Workspace and Package Creation

Before getting started, make sure 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 custom `.msg` and `.srv` files:

```
$ ros2 pkg create s7_robot_network_interface --build-type ament_cmake --license
↪  Apache-2.0 --description "Task Manager Interface: .msg and .srv files"
```

- Python package to contain the task manager node (server):

```
$ ros2 pkg create s7_py_task_manager --build-type ament_cmake --dependencies rclpy
↪  s7_robot_network_interface geometry_msgs --license Apache-2.0 --description
↪  "Task Manager - Server"
```

- Python package to contain the robot state machine nodes (clients):

```
$ ros2 pkg create s7_py_client_robot --build-type ament_python --dependencies rclpy
↪  s7_robot_network_interface geometry_msgs sensor_msgs tf2_ros tf_transformations
↪ --license Apache-2.0 --description "Robot State Machine - Clients"
```

- Python package to contain the nodes for monitoring the robot network tasks:

```
$ ros2 pkg create s7_py_robot_task_monitoring --build-type ament_python
↪ --dependencies rclpy s7_robot_network_interface geometry_msgs sensor_msgs
↪  tf2_ros tf_transformations --license Apache-2.0 --description "Robot Network
↪  Task Monitoring"
```

Feel free to replace s7_robot_network_interface, s7_py_task_manager, s7_py_client_robot, and s7_py_robot_task_monitoring with names of your preference.

7.2.2 Installing Required Packages

Before proceeding, ensure that the following library and packages are installed. Update your package list (apt repository) and install them with:

```
$ sudo apt update
$ sudo apt install ros-humble-joint-state-publisher
$ sudo apt install ros-humble-tf-transformations
$ sudo pip3 install transforms3d
$ sudo apt install ros-humble-tf2-tools # or ros-humble-tf2*
```

Also, from the root of your workspace, use rosdep to check for any other missing dependencies before building:

```
$ rosdep install -i --from-path src --rosdistro humble -y
```

7.3 Package Setup: s7_robot_network_interface

The .msg and .srv files are required to be placed in directories called msg and srv respectively. Create the directories in ros2_ws/src/s7_robot_network_interface:

```
$ mkdir msg srv
```

7.3.1 Creating the .msg and .srv files

msg definition

In the s7_robot_network_interface/msg directory you just created, make a new file called RobotStatus.msg with a few lines of code declaring its data structure:

```
builtin_interfaces/Time timestamp
int64 robot_id
int64 task_stage
geometry_msgs/Pose2D pose
bool align_yaw
```

This is a custom message that transfers different message types, and uses a message from other message packages (builtin_interfaces/Time and geometry_msgs/Pose2D).

srv definition

Back in the s7_robot_network_interface/srv directory you just created, make a new file called GetTwoPoses.srv with the following request and response structure:

```
int64 robot_id
---
geometry_msgs/Pose2D pickup_pose
geometry_msgs/Pose2D delivery_pose
```

This is your custom service that requests a service with one integer named robot_id, and responds with two 2D poses called pickup_pose and delivery_pose.

7.3.2 Configuring 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/RobotStatus.msg"
  "srv/GetTwoPoses.srv"
  DEPENDENCIES builtin_interfaces geometry_msgs
)
```

7.3.3 Configuring 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, s7_robot_network_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>
```

7.3.4 Build the `s7_robot_network_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 s7_robot_network_interface
```

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

7.3.5 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 s7_robot_network_interface/msg/RobotStatus
```

should return:

```
builtin_interfaces/Time timestamp
    int32 sec
    uint32 nanosec
int64 robot_id
int64 task_stage
geometry_msgs/Pose2D pose
    float64 x
    float64 y
    float64 theta
bool align_yaw
```

And

```
$ ros2 interface show s7_robot_network_interface/srv/GetTwoPoses
```

should return:

```
int64 robot_id
---
geometry_msgs/Pose2D pickup_pose
    float64 x
    float64 y
    float64 theta
geometry_msgs/Pose2D delivery_pose
    float64 x
    float64 y
    float64 theta
```

7.4 Package Setup: s7_py_task_manager

7.4.1 Creating a ROS 2 Pose Server Node with Python

Setting Up the Python Node

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

```
$ cd ros2_ws/src/s7_py_task_manager/s7_py_task_manager
```

Replace s7_py_task_manager with your actual package name.

Creating the Python Node File:
Create a new Python file for your pose server node, for example, task_manager.py:

```
$ touch task_manager.py
```

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

```
$ ls
```

You should now see the task_manager.py file.

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 task_manager.py file for editing.

Implementing the Node Code:
Below is an example implementation of a ROS 2 pose server node in Python:

Pose Server in Python:

```python
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from s7_robot_network_interface.srv import GetTwoPoses
from geometry_msgs.msg import Pose2D
import random

class TaskManagerService(Node):
    def __init__(self):
        super().__init__('py_task_manager_node')
        self.get_logger().info("Task Manager node has been started")

        # Create service
        self.srv = self.create_service(GetTwoPoses, 'get_two_poses',
        ↪  self.GetTwoPoses_callback)
        self.get_logger().info("Service 'get_two_poses' is ready to receive requests")

        # Create publishers for each robot ID (1-4)
        self.pickup_publishers = {}
        self.delivery_publishers = {}
        for robot_id in range(1, 5):
            ns = f'/robot{robot_id}'
            self.pickup_publishers[robot_id] = self.create_publisher(Pose2D,
            ↪  f'{ns}/pickup_pose', 10)
            self.delivery_publishers[robot_id] = self.create_publisher(Pose2D,
            ↪  f'{ns}/delivery_pose', 10)


    def GetTwoPoses_callback(self, request, response):
        lim = 2.0

        # Generate random pickup pose
        pickup_pose = Pose2D()
        pickup_pose.x = round(random.uniform(-lim, lim), 2)
        pickup_pose.y = round(random.uniform(-lim, lim), 2)
        pickup_pose.theta = round(random.uniform(-3.14, 3.14), 2)

        # Generate random delivery pose
        delivery_pose = Pose2D()
        delivery_pose.x = round(random.uniform(-lim, lim), 2)
        delivery_pose.y = round(random.uniform(-lim, lim), 2)
        delivery_pose.theta = round(random.uniform(-3.14, 3.14), 2)

        # Fill response
        response.pickup_pose = pickup_pose
        response.delivery_pose = delivery_pose

        self.get_logger().info(f'Received request from ID: {request.robot_id}')
        self.get_logger().info(f'Providing pickup pose: {pickup_pose}')
        self.get_logger().info(f'Providing delivery pose: {delivery_pose}')
```

```
        # Publish to the robot-specific topics
        robot_id = request.robot_id
        if robot_id in self.pickup_publishers:
            self.pickup_publishers[robot_id].publish(pickup_pose)
            self.delivery_publishers[robot_id].publish(delivery_pose)
        else:
            self.get_logger().warn(f"No publishers set up for robot ID {robot_id}")

        return response

def main(args=None):
    rclpy.init(args=args)
    node = TaskManagerService()
    try:
        rclpy.spin(node)
    except Exception as e:
        node.get_logger().error(f'Exception caught: {e}')
    finally:
        node.destroy_node()
        rclpy.shutdown()

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

7.4.2 Integrating the Python Pose Sever Node into the Package

Modify `setup.py` of your Python package (`s7_py_task_manager`) to add an executable for the pose server node:

```
entry_points={
    'console_scripts': [
        "task_manager_exe = s7_py_task_manager.task_manager:main"
    ],
},
```

7.4.3 Summary of Python Pose Server Node Key Identifiers

- `task_manager.py`: Name of the source file for the Python pose server node.
- `py_task_manager_node`: Name of the Python pose server node as defined in the constructor of the Python class.
- `task_manager_exe`: Name of the executable defined in `setup.py`.
- `get_two_poses`: Name of the service provided by the Python server node.

7.5 Package Setup: s7_py_client_robot

In this Section, we present the setup of the s7_py_client_robot package for one robot client named, labeled, and namespaced as robot1, including the corresponding node in Python, and configuration in the client_robots_launch.py and setup.py files.

On the other hand, in our ROS 2 course GitHub repository, you can find the corresponding source files and configuration for other three robot clients named, labeled, and namespaced as robot2, robot3, and robot4.

7.5.1 Creating a ROS 2 Robot Client Node with Python

Setting Up the Python Node

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

```
$ cd ros2_ws/src/s7_py_client_robot/s7_py_client_robot
```

Replace s7_py_client_robot with your actual package name.

Creating the Python Node File:
Create a new Python file for your robot client node, for example, client_robot1.py:

```
$ touch client_robot1.py
```

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

```
$ ls
```

You should now see the client_robot1.py file.

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 client_robot1.py file for editing.

Implementing the Node Code:
Below is an example implementation of a ROS 2 robot client node in Python:

Robot Client in Python:

```python
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from s7_robot_network_interface.srv import GetTwoPoses
from s7_robot_network_interface.msg import RobotStatus
from sensor_msgs.msg import JointState
from geometry_msgs.msg import TransformStamped
from tf2_ros import TransformBroadcaster
import tf_transformations
import math

class PosePublisher(Node):
    def __init__(self):
        super().__init__('py_robot_client_node')
        self.get_logger().info('Pose Publisher node has been started')

        # --- Service client setup ---
        self.client = self.create_client(GetTwoPoses, 'get_two_poses')
        while not self.client.wait_for_service(timeout_sec=1.0):
            if not rclpy.ok():
                self.get_logger().error('Interrupted while waiting for service.
                ↪ Exiting.')
                return
            self.get_logger().info('Waiting for service to become available...')
        self.req = GetTwoPoses.Request()

        # --- Robot ID ---
        self.robot_id = 1

        # --- Robot state ---
        self.x = 0.0
        self.y = 0.0
        self.yaw = 0.0
        # self.wheel_angle = 0.0

        # --- Movement parameters ---
        self.step_size = 0.01       # meters per update
        self.yaw_step = math.pi / 180.0  # radians per update
        # self.wheel_step = math.pi / 180.0  # radians per update

        # --- State machine stage ---
        # 0: rotate to pickup
        # 1: move to pickup
        # 2: rotate to pickup final yaw
        # 3: rotate to delivery
        # 4: move to delivery
        # 5: rotate to delivery final yaw
        # 6: request new poses
        self.stage = 0

        # --- Fetch initial poses ---
        self.get_new_poses()
```

```
        # --- Publishers ---
        self.robot_pub = self.create_publisher(RobotStatus, 'robot_status', 10)
        # self.joint_pub = self.create_publisher(JointState, 'joint_states', 10)
        # self.broadcaster = TransformBroadcaster(self)

        # --- Timer for periodic updates (~30 Hz) ---
        self.timer = self.create_timer(0.033, self.publish)

    def get_new_poses(self):
        """Call the get_two_poses service."""
        self.req.robot_id = self.robot_id
        future = self.client.call_async(self.req)
        future.add_done_callback(self.handle_service_response)

    def handle_service_response(self, future):
        """Store pickup & delivery poses."""
        resp = future.result()
        if resp is None:
            self.get_logger().error('Failed to call service get_two_poses')
            return

        # Store poses
        self.pickup_pose = resp.pickup_pose
        self.delivery_pose = resp.delivery_pose

        self.pickup_yaw = self.pickup_pose.theta
        self.delivery_yaw = self.delivery_pose.theta

        # Pre-compute yaw needed to face each goal
        self.pickup_to_yaw = math.atan2(
            self.pickup_pose.y - self.y,
            self.pickup_pose.x - self.x
        )

        self.get_logger().info(
            f'New pickup_pose: ({self.pickup_pose.x:.2f}, {self.pickup_pose.y:.2f},
            ↪ {self.pickup_pose.theta:.2f} rad)'
        )
        self.get_logger().info(
            f'New delivery_pose: ({self.delivery_pose.x:.2f}, {self.delivery_pose.y:.2f},
            ↪ {self.delivery_pose.theta:.2f} rad)'
        )

    def normalize_angle(self, angle: float) -> float:
        """Normalize an angle to [-pi, pi]."""
        while angle > math.pi:
            angle -= 2 * math.pi
        while angle < -math.pi:
            angle += 2 * math.pi
        return angle

    def reached(self, a: float, b: float, tol: float = 1e-2) -> bool:
        """Return True if a and b are within tol."""
        return abs(a - b) < tol
```

```
    def publish(self):
        # --- State machine logic ---
        if self.stage == 0:
            # 1. Rotate toward pickup
            yaw_diff = self.normalize_angle(self.pickup_to_yaw - self.yaw)
            if abs(yaw_diff) > self.yaw_step:
                self.yaw += math.copysign(self.yaw_step, yaw_diff)
                self.yaw = self.normalize_angle(self.yaw)
            else:
                self.yaw = self.pickup_to_yaw
                self.stage = 1

        elif self.stage == 1:
            # 2. Move toward pickup position
            dx = self.pickup_pose.x - self.x
            dy = self.pickup_pose.y - self.y
            dist = math.hypot(dx, dy)
            if dist > self.step_size:
                direction = math.atan2(dy, dx)
                self.x += math.cos(direction) * self.step_size
                self.y += math.sin(direction) * self.step_size
            else:
                self.x = self.pickup_pose.x
                self.y = self.pickup_pose.y
                self.stage = 2

        elif self.stage == 2:
            # 3. Rotate to final pickup yaw
            yaw_diff = self.normalize_angle(self.pickup_yaw - self.yaw)
            if abs(yaw_diff) > self.yaw_step:
                self.yaw += math.copysign(self.yaw_step, yaw_diff)
                self.yaw = self.normalize_angle(self.yaw)
            else:
                self.yaw = self.pickup_yaw
                # Prepare rotation to delivery
                self.delivery_to_yaw = math.atan2(
                    self.delivery_pose.y - self.y,
                    self.delivery_pose.x - self.x
                )
                self.stage = 3

        elif self.stage == 3:
            # 4. Rotate toward delivery
            yaw_diff = self.normalize_angle(self.delivery_to_yaw - self.yaw)
            if abs(yaw_diff) > self.yaw_step:
                self.yaw += math.copysign(self.yaw_step, yaw_diff)
                self.yaw = self.normalize_angle(self.yaw)
            else:
                self.yaw = self.delivery_to_yaw
                self.stage = 4
```

```
        elif self.stage == 4:
            # 5. Move toward delivery position
            dx = self.delivery_pose.x - self.x
            dy = self.delivery_pose.y - self.y
            dist = math.hypot(dx, dy)
            if dist > self.step_size:
                direction = math.atan2(dy, dx)
                self.x += math.cos(direction) * self.step_size
                self.y += math.sin(direction) * self.step_size
            else:
                self.x = self.delivery_pose.x
                self.y = self.delivery_pose.y
                self.stage = 5

        elif self.stage == 5:
            # 6. Rotate to final delivery yaw
            yaw_diff = self.normalize_angle(self.delivery_yaw - self.yaw)
            if abs(yaw_diff) > self.yaw_step:
                self.yaw += math.copysign(self.yaw_step, yaw_diff)
                self.yaw = self.normalize_angle(self.yaw)
            else:
                self.yaw = self.delivery_yaw
                self.stage = 6

        elif self.stage == 6:
            # 7. Completed delivery, request new goals
            self.get_logger().info('Completed delivery. Requesting new goals...')
            self.get_new_poses()
            self.stage = 0

        # --- Publish the current Robot Status ---
        robot_msg = RobotStatus()
        robot_msg.timestamp = self.get_clock().now().to_msg()
        robot_msg.robot_id = self.robot_id
        robot_msg.task_stage = self.stage
        robot_msg.pose.x = self.x
        robot_msg.pose.y = self.y
        robot_msg.pose.theta = self.yaw
        robot_msg.align_yaw = True
        self.robot_pub.publish(robot_msg)

        # --- Publish joint states ---
        # self.wheel_angle += 4 * self.wheel_step
        # if abs(self.wheel_angle) > math.pi:
        #     self.wheel_angle *= -1

        # js = JointState()
        # js.header.stamp = self.get_clock().now().to_msg()
        # js.name = [
        #     'left_front_joint',
        #     'left_back_joint',
        #     'right_front_joint',
        #     'right_back_joint'
        # ]
```

```
        # js.position = [
        #     self.wheel_angle,
        #     self.wheel_angle,
        #     -self.wheel_angle,
        #     -self.wheel_angle
        # ]
        # self.joint_pub.publish(js)

        # --- Publish transform for RViz ---
        # t = TransformStamped()
        # t.header.stamp = self.get_clock().now().to_msg()
        # t.header.frame_id = 'odom'
        # namespace = self.get_namespace().strip('/')
        # t.child_frame_id = f'{namespace}/base_footprint'
        # t.transform.translation.x = self.x
        # t.transform.translation.y = self.y
        # t.transform.translation.z = 0.0
        # q_tf = tf_transformations.quaternion_from_euler(0, 0, self.yaw)
        # t.transform.rotation.x = q_tf[0]
        # t.transform.rotation.y = q_tf[1]
        # t.transform.rotation.z = q_tf[2]
        # t.transform.rotation.w = q_tf[3]
        # self.broadcaster.sendTransform(t)

def main(args=None):
    rclpy.init(args=args)
    node = PosePublisher()
    try:
        rclpy.spin(node)
    except Exception as e:
        node.get_logger().error(f'Exception caught: {e}')
    finally:
        node.destroy_node()
        rclpy.shutdown()

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

7.5.2 Integrating the Python Robot Client Node into the Package

Modify `setup.py` of your Python package (`s7_py_client_robot`) to add an executable for the robot client node:

```
entry_points={
    'console_scripts': [
        "client_robot1_exe = s7_py_client_robot.client_robot1:main"
    ],
},
```

7.5.3 Summary of Python Robot Client Node Key Identifiers

- `client_robot1.py`: Name of the source file for the Python robot client node.

- py_robot_client_node: Name of the Python robot client node as defined in the constructor of the Python class.
- client_robot1_exe: Name of the executable defined in setup.py.
- get_two_poses: Name of the service that the Python client node calls.

7.5.4 Launch File Definition

Within the s7_py_robot_client/launch directory, create (or open) a file named clients_robots_launch.py with the following structure.

```
import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    # robot1: RobotStatus publisher node
    robotstatus1 = Node(
        package='s7_py_client_robot',
        executable='client_robot1_exe',
        name='robot1',
        namespace='robot1',
        remappings=[  # <<< this makes /robot1/get_two_poses → /get_two_poses
            ('get_two_poses', '/get_two_poses'),
        ],
        output='screen'
    )

    return LaunchDescription([
        robotstatus1
    ])
```

This launch file starts the robot client node with the following features:

- Adds the robot1 namespace to the robot client node.
- Adds the robot1 name to the robot client node instead of py_robot_client_node, as previously defined in the constructor of the Python class.
- Remaps the /robot1/get_two_poses service name to /get_two_poses to correctly call the pose service.

7.6 Package Setup: s7_py_robot_task_monitoring

In this Section, we present the setup of the s7_py_robot_task_monitoring package for the robot state publishers of one robot and the two cylinders that represent the pickup and delivery poses, respectively. The source files of these publishers are named, labeled, and namespaced as robot1, cylinder1, and cylinder2, respectively, including the corresponding node in Python, URDF model, and configuration in the robot_network_launch.py and setup.py files.

On the other hand, in our ROS 2 course GitHub repository, you can find the corresponding source files and configuration for other three robot state publishers named, labeled, and namespaced as robot2, robot3, and robot4: and the source files and configuration for other six robot state

publishers named, labeled, and namespaced as from cylinder3 to cylinder8, to represent the respective pickup and delivery poses of each additional robot.

7.6.1 Creating a Pose Sub-Pub Node with Python: Pickup and Delivery Poses

Setting Up the Python Node

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

```
$ cd ros2_ws/src/s7_py_robot_task_monitoring/s7_py_robot_task_monitoring
```

Replace s7_py_robot_task_monitoring with your actual package name.

Creating the Python Node File:
Create a new Python file for your pose sub-pub node, for example, cylinders.py:

```
$ touch cylinders.py
```

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

```
$ ls
```

You should now see the cylinders.py file.

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 cylinders.py file for editing.

Implementing the Node Code:
Below is an example implementation of a ROS 2 pose sub-pub node in Python:

Pose Sub-Pub in Python:

```python
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Pose2D
from geometry_msgs.msg import TransformStamped
from tf2_ros import TransformBroadcaster
import tf_transformations

class CylinderVisualizer(Node):
    def __init__(self):
        super().__init__('py_cylinder_visualizer_node')
        self.get_logger().info("Python Cylinder Visualizer node has been started")

        # TF broadcaster for all cylinders
        self.broadcaster = TransformBroadcaster(self)

        # --- Number of robots ---
        self.robot_num = 1

        # Create subscribers for pickup & delivery for robots 1-4
        self.subs = [] # keep subscriptions alive
        for rid in range(1, self.robot_num + 1):
            ns = f'/robot{rid}'
            sub_pick = self.create_subscription(
                Pose2D, f'{ns}/pickup_pose',
                lambda msg, r=rid: self.pose_callback(msg, r, 'pickup'),
                10)
            sub_del = self.create_subscription(
                Pose2D, f'{ns}/delivery_pose',
                lambda msg, r=rid: self.pose_callback(msg, r, 'delivery'),
                10)
            self.subs += [sub_pick, sub_del]

    def pose_callback(self, msg: Pose2D, robot_id: int, kind: str):
        """
        Broadcast a TF frame for a cylinder based on:
        - robot_id: 1-4
        - kind: 'pickup' or 'delivery'
        """
        # Determine cylinder index: pickup1→1, delivery1→2, pickup2→3, etc.
        base = (robot_id - 1) * 2
        cyl_idx = base + (1 if kind == 'pickup' else 2)

        t = TransformStamped()
        t.header.stamp = self.get_clock().now().to_msg()
        t.header.frame_id = 'odom'
        t.child_frame_id = f'cylinder{cyl_idx}/base_footprint'

        # Translation from Pose2D
        t.transform.translation.x = msg.x
        t.transform.translation.y = msg.y
        t.transform.translation.z = 0.0
```

```
        # Rotation from yaw
        q = tf_transformations.quaternion_from_euler(0.0, 0.0, msg.theta)
        t.transform.rotation.x = q[0]
        t.transform.rotation.y = q[1]
        t.transform.rotation.z = q[2]
        t.transform.rotation.w = q[3]

        # Publish to tf2
        self.broadcaster.sendTransform(t)
        self.get_logger().info(f"Broadcasted cylinder{cyl_idx}: ({msg.x:.2f},
        ↪ {msg.y:.2f}, {msg.theta:.2f})")

def main(args=None):
    rclpy.init(args=args)
    node = CylinderVisualizer()
    try:
        rclpy.spin(node)
    except Exception as e:
        node.get_logger().error(f'Exception caught: {e}')
    finally:
        node.destroy_node()
        rclpy.shutdown()

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

7.6.2 Integrating the Python Pose Sub-Pub Node into the Package

Modify `setup.py` of your Python package (`s7_py_robot_task_monitoring`) to add an executable for the pose sub-pub node:

```
entry_points={
    'console_scripts': [
        "cylinders_exe = s7_py_robot_task_monitoring.cylinders:main"
    ],
},
```

7.6.3 Summary of Python Pose Sub-Pub Node Key Identifiers

- `cylinders.py`: Name of the source file for the Python pose sub-pub node.
- `py_cylinder_visualizer_node`: Name of the Python pose sub-pub node as defined in the constructor of the Python class.
- `cylinders_exe`: Name of the executable defined in `setup.py`.
- `/robot1/pickup_pose`: Name of the topic that the Python node subscribes to get the pickup pose for the `robot1`.
- `/robot1/delivery_pose`: Name of the topic that the Python node subscribes to get the delivery pose for the `robot1`.

7.6.4 Creating a Status Sub-Pub Node with Python: Robot Status

Setting Up the Python Node

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

```
$ cd ros2_ws/src/s7_py_robot_task_monitoring/s7_py_robot_task_monitoring
```

Replace `s7_py_robot_task_monitoring` with your actual package name.

Creating the Python Node File:
Create a new Python file for your status sub-pub node, for example, `robots.py`:

```
$ touch robots.py
```

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

```
$ ls
```

You should now see the `robots.py` file.

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 `robots.py` file for editing.

Implementing the Node Code:
Below is an example implementation of a ROS 2 status sub-pub node in Python:

Status Sub-Pub in Python:

```python
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from s7_robot_network_interface.msg import RobotStatus
from sensor_msgs.msg import JointState
from geometry_msgs.msg import TransformStamped
from tf2_ros import TransformBroadcaster
import tf_transformations
import math
```

```python
class RobotVisualizer(Node):
    def __init__(self):
        super().__init__('py_robot_visualizer_node')
        self.get_logger().info("Python Robot Visualizer node has been started")

        # --- Robot wheel state ---
        self.wheel_angle = 0.0

        # --- Movement parameters ---
        self.wheel_step = math.pi / 180.0  # radians per update

        # --- Number of robots ---
        self.robot_num = 1

        # Create subscribers to robot_status topic for each robot
        self.subs = [] # keep subscriptions alive
        for rid in range(1, self.robot_num + 1):
            topic = f'/robot{rid}/robot_status'
            # Subscribe to '/robotN/robot_status'
            sub = self.create_subscription(
                RobotStatus, topic,
                lambda msg, r=rid: self.RobotPose_callback(msg, r),
                10)
            self.subs.append(sub)

        # Create publishers to joint_states topic for each robot
        self.pubs = [] # keep publications alive
        for rid in range(1, self.robot_num + 1):
            topic = f'/robot{rid}/joint_states'
            # Publish to '/robotN/joint_states'
            pub = self.create_publisher(JointState, topic, 10)
            self.pubs.append(pub)

        # --- Timer for periodic updates (~30 Hz) ---
        self.timer = self.create_timer(0.033, self.publish)

        # TF broadcaster for all robots
        self.broadcaster = TransformBroadcaster(self)

    def RobotPose_callback(self, msg: RobotStatus, robot_id: int):
        """
        Broadcast 'odom'→'robotN/base_footprint' transform
        based on Pose2D field from RobotStatus message.
        """
        t = TransformStamped()
        t.header.stamp = msg.timestamp # self.get_clock().now().to_msg()
        t.header.frame_id = 'odom'
        t.child_frame_id = f'robot{robot_id}/base_footprint'

        # Translation from Pose2D
        t.transform.translation.x = msg.pose.x
        t.transform.translation.y = msg.pose.y
        t.transform.translation.z = 0.0
```

```
        # Rotation from yaw
        q = tf_transformations.quaternion_from_euler(0.0, 0.0, msg.pose.theta)
        t.transform.rotation.x = q[0]
        t.transform.rotation.y = q[1]
        t.transform.rotation.z = q[2]
        t.transform.rotation.w = q[3]

        # Publish transform to tf2
        self.broadcaster.sendTransform(t)
        self.get_logger().debug(
            f"Broadcasted robot{robot_id} at ({msg.pose.x:.2f}, {msg.pose.y:.2f},
            ↪ {msg.pose.theta:.2f})"
        )

    def publish(self):
        # --- Publish joint states ---
        for robot_id in range(1, self.robot_num + 1):
            self.wheel_angle += 4 * self.wheel_step
            if abs(self.wheel_angle) > math.pi:
                self.wheel_angle *= -1

            js = JointState()
            js.header.stamp = self.get_clock().now().to_msg()
            js.name = [
                'left_front_joint',
                'left_back_joint',
                'right_front_joint',
                'right_back_joint'
            ]
            js.position = [
                self.wheel_angle,
                self.wheel_angle,
                -self.wheel_angle,
                -self.wheel_angle
            ]
            self.pubs[robot_id-1].publish(js)

def main(args=None):
    rclpy.init(args=args)
    node = RobotVisualizer()
    try:
        rclpy.spin(node)
    except Exception as e:
        node.get_logger().error(f'Exception caught: {e}')
    finally:
        node.destroy_node()
        rclpy.shutdown()

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

7.6.5 Integrating the Python Status Sub-Pub Node into the Package

Modify setup.py of your Python package (s7_py_robot_task_monitoring) to add an executable for the status sub-pub node after any existing ROS 2 node entries (e.g., cylinders_exe):

```
entry_points={
    'console_scripts': [
        "cylinders_exe = s7_py_robot_task_monitoring.cylinders:main",
        "robots_exe = s7_py_robot_task_monitoring.robots:main",
    ],
},
```

7.6.6 Summary of Python Status Sub-Pub Node Key Identifiers

- robots.py: Name of the source file for the Python status sub-pub node.
- py_robot_visualizer_node: Name of the Python status sub-pub node as defined in the constructor of the Python class.
- robots_exe: Name of the executable defined in setup.py.
- /robot1/robot_status: Name of the topic (of RobotStatus.msg type) that the Python node subscribes to get the status from the robot1 client.
- /robot1/joint_states: Name of the topic in which the Python node publishes wheel joint stats from the robot1 client.

7.6.7 Launch File Definition

Within the s7_py_task_monitoring/launch directory, create (or open) a file named robot_network_launch.py with the following structure.

```
import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    # Path to package
    pkg_share = get_package_share_directory('s7_py_robot_task_monitoring')

    # Paths to URDF files
    cylinder1_urdf = os.path.join(pkg_share, 'urdf', 'cylinder1.urdf')
    cylinder2_urdf = os.path.join(pkg_share, 'urdf', 'cylinder2.urdf')

    robot1_urdf = os.path.join(pkg_share, 'urdf', 'robot1.urdf')
```

```
    # Read URDF files
    with open(cylinder1_urdf, 'r') as infp:
        cylinder1_description = infp.read()

    with open(cylinder2_urdf, 'r') as infp:
        cylinder2_description = infp.read()

    with open(robot1_urdf, 'r') as infp:
        robot1_description = infp.read()

    # cylinder1 (pickup pose for Robot1) robot_state_publisher
    cylinder1_robot1_desc = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        # name='cylinder1_state_publisher',
        namespace='cylinder1',
        parameters=[{'robot_description': cylinder1_description}],
        output='screen'
    )
    # cylinder2 (delivery pose for Robot1) robot_state_publisher
    cylinder2_robot1_desc = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        # name='cylinder2_state_publisher',
        namespace='cylinder2',
        parameters=[{'robot_description': cylinder2_description}],
        output='screen'
    )

    # cylinders pose subscriber-publisher node
    cylinders_tf = Node(
        package='s7_py_robot_task_monitoring',
        executable='cylinders_exe',
        name='cylinders',
        output='screen'
    )

    # robot1 robot_state_publisher
    robot1_desc = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        # name='robot1_state_publisher',
        namespace='robot1',
        parameters=[{'robot_description': robot1_description}],
        output='screen'
    )

    # robots pose subscriber-publisher node
    robots_tf = Node(
        package='s7_py_robot_task_monitoring',
        executable='robots_exe',
        name='robots',
        output='screen'
    )
```

```
    return LaunchDescription([
        cylinder1_robot1_desc,
        cylinder2_robot1_desc,
        robot1_desc,
        cylinders_tf,
        robots_tf,
    ])
```

This launch file starts five nodes:

- The cylinder1/robot_state_publisher node (cylinder1_robot1_desc), which publishes the URDF model and the transformations of cylinder1 that represents the pickup pose for the robot1 client.
- The cylinder2/robot_state_publisher node (cylinder2_robot1_desc), which publishes the URDF model and the transformations of cylinder2 that represents the delivery pose for the robot1 client.
- The robot1/robot_state_publisher node (robot1_desc), which publishes the URDF model and the transformations of robot1 client.
- The cylinders node (cylinders_tf), responsible for subscribing and publishing the pose values from the pickup and delivery cylinders for the robot1 client.
- The robots node (robots_tf), responsible for subscribing and publishing the status from robot1 client, its joint state values, and the overall robot geometry.

7.6.8 Creating a Status Subscriber Node with Python: Robot Status Logger

Setting Up the Python Node

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

```
$ cd ros2_ws/src/s7_py_robot_task_monitoring/s7_py_robot_task_monitoring
```

Replace s7_py_robot_task_monitoring with your actual package name.

Creating the Python Node File:
Create a new Python file for your status subscriber node, for example, robot_status_logger.py:

```
$ touch robot_status_logger.py
```

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

```
$ ls
```

You should now see the robot_status_logger.py file.

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 `robot_status_logger.py` file for editing.

Implementing the Node Code:
Below is an example implementation of a ROS 2 status subscriber node in Python:

Status Subscriber in Python:

```python
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from s7_robot_network_interface.msg import RobotStatus
from datetime import datetime, timezone
try:
    from zoneinfo import ZoneInfo  # Available in Python 3.9+ (For Foxy/Ubuntu 22.04
    ↪ with Python 3.10)
except ImportError:
    from backports.zoneinfo import ZoneInfo  # For Python < 3.9 (For Humble/Ubuntu 20.04
    ↪ with Python 3.8)
import math

class RobotStatusLogger(Node):
    def __init__(self):
        super().__init__('py_robot_status_logger_node')
        self.get_logger().info("Python Robot Status Logger node has been started")

        # Robot ID → ID & color label map
        self.rid_label = {
            1: "1 - Blue",
            2: "2 - Green",
            3: "3 - Pink",
            4: "4 - Violet",
        }

        # Stage → human-readable description map
        self.stage_desc = {
            0: "Rotating to pickup point",
            1: "Moving to pickup point",
            2: "Rotating to pickup final yaw",
            3: "Rotating to delivery point",
            4: "Moving to delivery point",
            5: "Rotating to delivery final yaw",
            6: "Requesting new points",
            7: "Fault: stuck or error",
            8: "No data received"
        }

        # Threshold for stale data (cycles)
        self.STALE_THRESHOLD = 30
```

```
        # Default "no data" template
        self.NO_DATA_STAGE = 8
        self.NO_DATA_ENTRY = lambda rid: {
            'id': self.rid_label[rid],
            'time': '--:--:--.--',
            'align': 'N/A',
            'pose': '(  -.--,  -.--,   -.--°)',
            'stage': rid * 0 + self.NO_DATA_STAGE,  # always 8
            'desc': self.stage_desc[self.NO_DATA_STAGE]
        }

        # Initialize statuses and stale counters
        self.statuses = {rid: self.NO_DATA_ENTRY(rid) for rid in range(1, 5)}
        self.no_update_counts = {rid: 0 for rid in range(1, 5)}

        # 2. Subscribe to /robotN/robot_status for N=1..4
        self.subs = []
        for rid in range(1, 5):
            topic = f'/robot{rid}/robot_status'
            sub = self.create_subscription(
                RobotStatus, topic,
                lambda msg, r=rid: self.status_callback(msg, r),
                10
            )
            self.subs.append(sub)

        # 3. Timer to print periodic updates (~30 Hz)
        self.create_timer(0.033, self.print_table)


    def status_callback(self, msg: RobotStatus, robot_id: int):
        # 1. Format timestamp as HH:MM:SS.ss
        ts = msg.timestamp.sec + msg.timestamp.nanosec * 1e-9
        dt = datetime.fromtimestamp(ts,
        ↪  tz=timezone.utc).astimezone(ZoneInfo("America/Mexico_City"))
        timestamp_str = dt.strftime('%H:%M:%S.%f')[:-4]  # two decimals

        # 2. Alignment?
        align_str = "True" if msg.align_yaw else "False"

        # 3. Pose2D as (x,y,t°)
        x, y = msg.pose.x, msg.pose.y
        theta_deg = math.degrees(msg.pose.theta)
        pose_str = f"({x:6.2f},{y:6.2f},{theta_deg:7.2f}°)"

        # 4. Stage and description
        stage = msg.task_stage
        desc = self.stage_desc.get(stage, "Unknown stage")

        # Update status entry
        self.statuses[robot_id] = {
            'id': self.rid_label.get(robot_id),
            'time': timestamp_str,
            'align': align_str,
```

```
            'pose': pose_str,
            'stage': stage,
            'desc': desc
        }

        # Reset stale counter
        self.no_update_counts[robot_id] = 0

    def print_table(self):
        # 0. Record the local "snapshot" time
        now_local = datetime.now(ZoneInfo("America/Mexico_City"))
        snap_str = now_local.strftime('%Y-%m-%d %H:%M:%S.%f')[:-4]

        # 1. Emit a line showing when we took this snapshot
        header = f"--- Robot status snapshot at local time: {snap_str} ---"
        lines = [f"{header:^112s}"]

        # 2. Fields
        fields = "|  Robot ID  |  Timestamp  | Alignment? |     Pose2D (x,y,t°)      |
        ↪  Task Stage |        Task Description        |"
        lines.append(fields)

        # 3. Rows for robots 1-4, guaranteed presence
        for rid in sorted(self.statuses):
            row = self.statuses[rid]
            line = (
                f"| {row['id']:<10s} | {row['time']:^11s} | {row['align']:^10s} "
                f"| {row['pose']:^18s} | {row['stage']:^10d} | {row['desc']:<30s} |"
            )
            lines.append(line)

        # 4. Print once
        self.get_logger().info("\n" + "\n".join(lines))

        # 5. Increment counters for robots without updates
        for rid in self.no_update_counts:
            self.no_update_counts[rid] += 1
            # If a robot has gone stale, reset to NO_DATA
            if self.no_update_counts[rid] >= self.STALE_THRESHOLD:
                self.statuses[rid] = self.NO_DATA_ENTRY(rid)
                self.no_update_counts[rid] = 0

def main(args=None):
    rclpy.init(args=args)
    node = RobotStatusLogger()
    try:
        rclpy.spin(node)
    finally:
        node.destroy_node()
        rclpy.shutdown()

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

7.6.9 Integrating the Python Status Subscriber Node into the Package

Modify `setup.py` of your Python package (`s7_py_robot_task_monitoring`) to add an executable for the status subscriber node after any existing ROS 2 node entries (e.g., `cylinders_exe` and `robots_exe`):

```
entry_points={
    'console_scripts': [
        "cylinders_exe = s7_py_robot_task_monitoring.cylinders:main",
            "robots_exe = s7_py_robot_task_monitoring.robots:main",
            "robot_status_logger_exe =
             ↪  s7_py_robot_task_monitoring.robot_status_logger:main",
    ],
},
```

7.6.10 Summary of Python Status Subscriber Node Key Identifiers

- `robot_status_logger.py`: Name of the source file for the Python status subscriber node.
- `py_robot_status_logger_node`: Name of the Python status subscriber node as defined in the constructor of the Python class.
- `robot_status_logger_exe`: Name of the executable defined in `setup.py`.
- `/robot1/robot_status`: Name of the topic (of `RobotStatus.msg` type) that the Python node subscribes to get the status from the `robot1` client.

7.7 Building the Packages and Running the Nodes

After making the above edits and saving all the changes, go to the root of your workspace (e.g., `/ros2_ws`) and build the packages:

```
$ colcon build --packages-select s7_robot_network_interface
$ colcon build --packages-select s7_py_task_manager
$ colcon build --packages-select s7_py_client_robot
$ colcon build --packages-select s7_py_robot_task_monitoring
```

Then, in five terminals, source the workspace environment in each, and proceed as follows:

- In **Terminal 1**, run the pose server node:

  ```
  $ source install/setup.bash
  $ ros2 run s7_py_task_manager task_manager_exe
  ```

- In **Terminal 2**, launch the robot client node:

  ```
  $ source install/setup.bash
  $ ros2 launch s7_py_client_robot client_robots_launch.py
  ```

- In **Terminal 3**, launch the pose sub-pub (pickup and delivery cylinders) and status sub-pub (robot status) nodes:

  ```
  $ source install/setup.bash
  $ ros2 launch s7_py_robot_task_monitoring robot_network_launch.py
  ```

- In **Terminal 4**, start RViz with the configuration file stored in the `robot_task_monitoring` package:

```
$ source install/setup.bash
$ rviz2 -d src/s7_py_robot_task_monitoring/rviz/robot_network.rviz
```

- In **Terminal 5**, run the status subscriber node to log the robot status:

```
$ source install/setup.bash
$ ros2 run s7_py_robot_task_monitoring robot_status_logger_exe
```

Additionally, open two terminals for diagnostics and introspection tools:

- **Terminal 6:** Use the `view_frames` tool from the `tf2_tools` package to visualize the current coordinate frame hierarchy (tree):

```
$ ros2 run tf2_tools view_frames
```

- **Terminal 7:** Launch the `rqt_graph` tool to view the current ROS 2 node and topic graph:

```
$ rqt_graph
```

7.8 Practice Assignment: `tf2` Library (Robot Network)

In this assignment, you will run the ROS 2 robot network implemented in Python, as described in Sections 7.4, 7.5, and 7.6. Capture evidence of successful execution using two screenshots as PNG files and one PDF file, and submit them in the designated assignment area on Canvas (within the ROS 2 module). The filenames must follow this format:

`FirstnameLastname_evidence_sY_Z.png`

where `sY` correspond to the session number, and `Z` is the evidence number. For example, correct filenames would be:

`EduardoDavila_evidence_s7_1.png`,
`EduardoDavila_evidence_s7_2.pdf`,
`EduardoDavila_evidence_s7_3.png`.

7.8.1 Executing ROS 2 Robot Status Publishers

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

```
$ colcon build --packages-select s7_robot_network_interface
$ colcon build --packages-select s7_py_task_manager
$ colcon build --packages-select s7_py_client_robot
$ colcon build --packages-select s7_py_robot_task_monitoring
```

Then, in five terminals, source the workspace environment in each, and proceed as follows:

- In **Terminal 1**, run the pose server node:

```
$ source install/setup.bash
$ ros2 run s7_py_task_manager task_manager_exe
```

- In **Terminal 2**, launch the robot client node:

```
$ source install/setup.bash
$ ros2 launch s7_py_client_robot client_robots_launch.py
```

- In **Terminal 3**, launch the pose sub-pub (pickup and delivery cylinders) and status sub-pub (robot status) nodes:

```
$ source install/setup.bash
$ ros2 launch s7_py_robot_task_monitoring robot_network_launch.py
```

- In **Terminal 4**, start RViz with the configuration file stored in the `robot_task_monitoring` package:

```
$ source install/setup.bash
$ rviz2 -d src/s7_py_robot_task_monitoring/rviz/robot_network.rviz
```

- In **Terminal 5**, run the status subscriber node to log the robot status:

```
$ source install/setup.bash
$ ros2 run s7_py_robot_task_monitoring robot_status_logger_exe
```

You should observe that:

1. The task_manager node process the service requests from the robot1 client node.
2. The cylinder1/robot_state_publisher and the cylinder2/robot_state_publisher nodes publish the corresponding cylinder URDF models (for visualization in RViz) that represents the pickup and delivery poses, respectively, for the robot1 client node.
3. The robot1/robot_state_publisher node publishes the robot1 URDF model and the coordinate transforms for visualization in RViz.
4. The cylinders node subscribes and publishes the pose values from the pickup and delivery cylinders for the robot1 client, to broadcast its coordinate transforms for visualization in RViz.
5. The robots node subscribes and publishes the status from robot1 client, its joint state values, and the overall robot geometry to broadcast its coordinate transforms for visualization in RViz.
6. The robot_status_logger node constantly prints the robot status if its subscription receives data.

7.8.2 Visualizing the frames with view_frames

To observe the current coordinate frame hierarchy, use the image_frames tool from the tf2_tools package. Then, in **Terminal 6**, execute the following command (without the need to enter the workspace directory):

```
$ ros2 run tf2_tools view_frames
```

This tool will generate a PDF file in the current directory of the terminal. The file provides a graphical representation of the current coordinate frame tree published by the robot_state_publisher node from robot1 URDF model.

7.8.3 Visualizing the ROS Graph with rqt_graph

To verify that all the nodes are correctly connected, use the rqt_graph tool. Then, in **Terminal 7**, execute the following command (without the need to enter the workspace directory):

```
$ ros2 run rqt_graph rqt_graph
```

or simply:

```
$ rqt_graph
```

This will visualize the ROS 2 graph, showing topics connecting the robot network nodes, as described above.

7.8.4 Submission Instructions

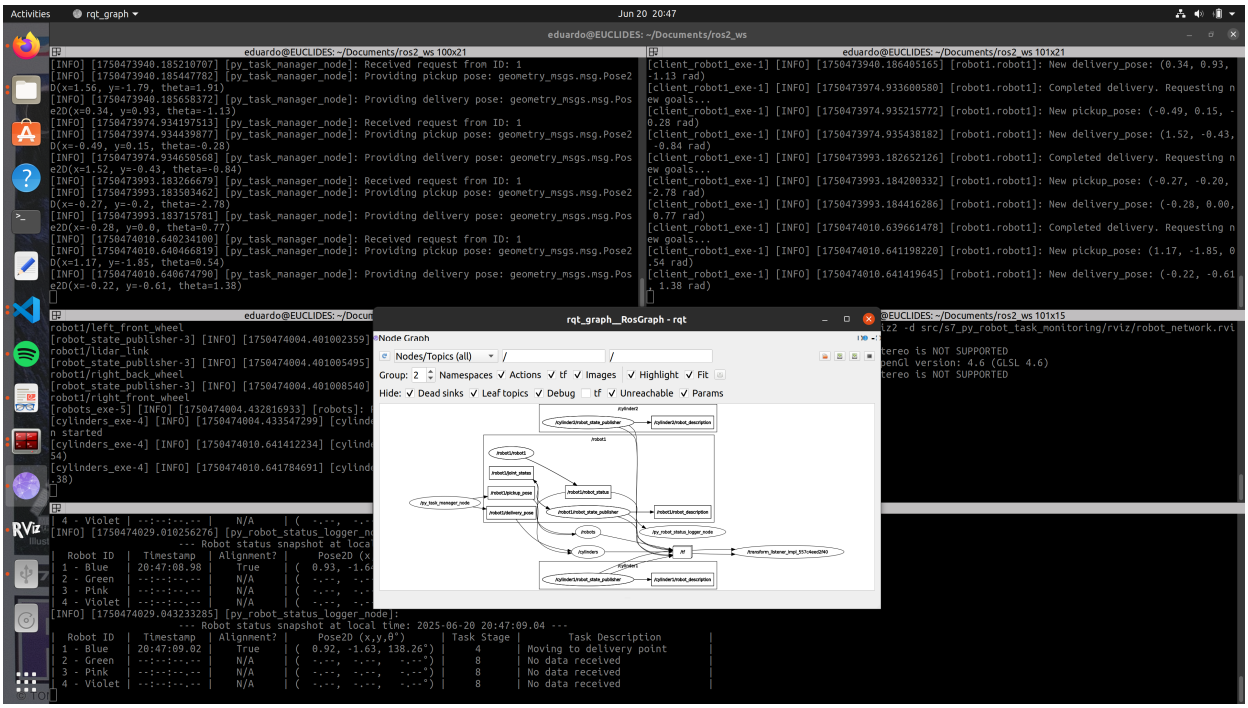


Figure 7.2: Example screenshot showing terminal outputs and the ROS 2 graph of the robot network in action.

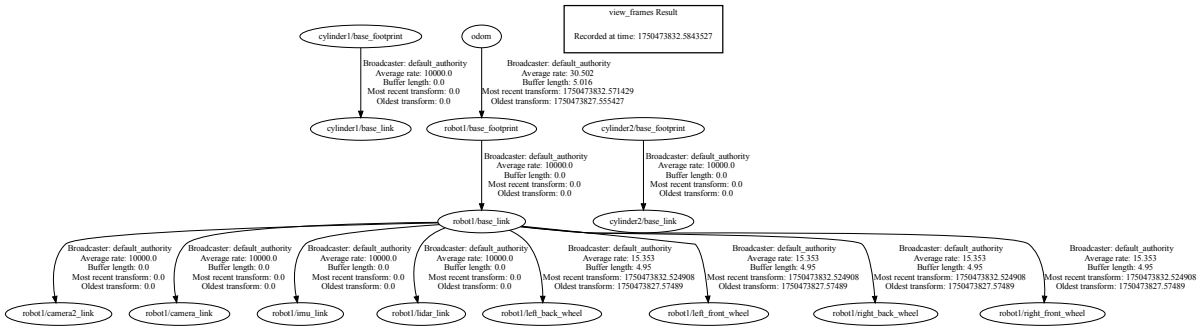


Figure 7.3: PDF showing the graphical representation of the current coordinate frame tree.

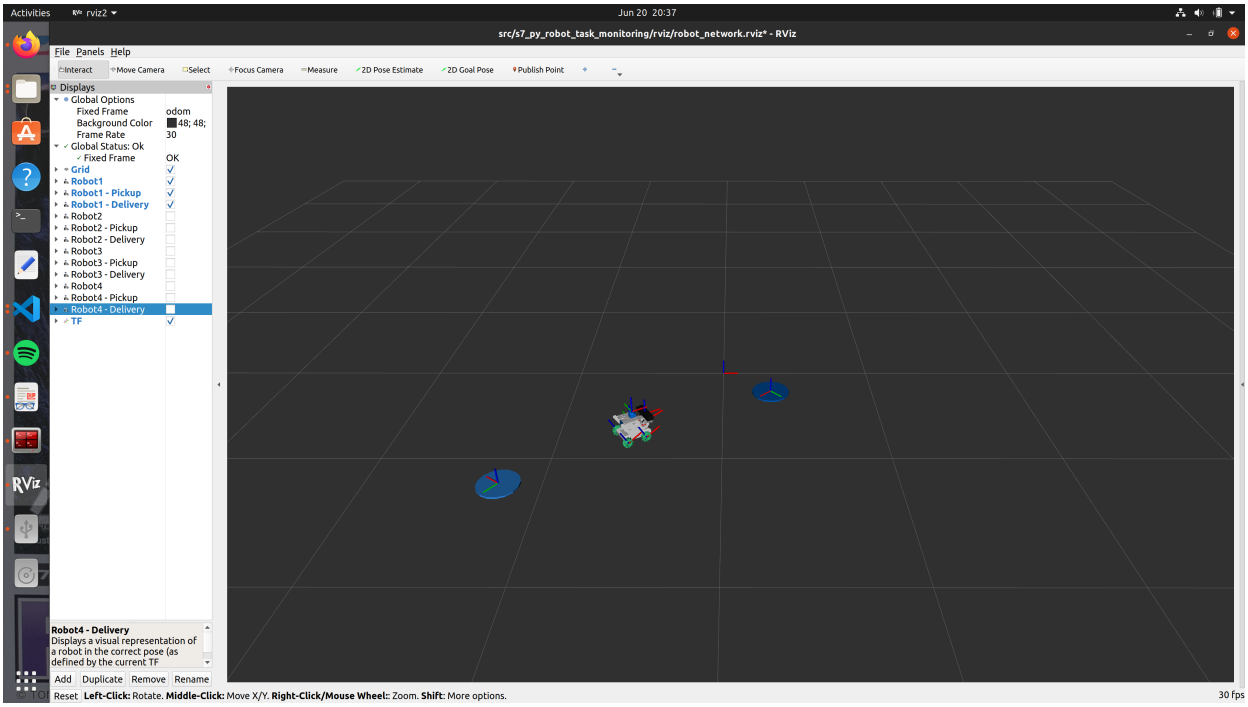


Figure 7.4: Example screenshot showing the visualization of the RDK X3 `Robot1` in RViz.

1. Run the ROS 2 robot network nodes as described above.
2. As first evidence, capture a **screenshot showing the terminal outputs and the ROS 2 graph** from either the C++ or Python nodes, as described in Sections 7.8.1 and 7.8.3. See Figure 7.2 for reference.
3. As second evidence, locate the PDF file generated by the `view_frames` tool from the `tf2_tools` package, as described in Section 7.8.2. See Figure 7.3 for reference.
4. As third evidence, capture a **screenshot showing the visualization of the RDK X3** `Robot1` **in RViz**. See Figure 7.4 for reference.
5. Save the screenshots as PNG files.
6. Name the files following this format: `FirstnameLastname_evidence_sX_Y.png` (replace `X` with the session number and `Y` with the evidence number).
7. Submit the files as specified by the course guidelines on Canvas.

Note: Your machine's username and hostname must be visible in the screenshots to verify the authenticity of your submission, as shown in Figure 7.2. Any submission that appears copied, unclear, or altered will be considered invalid and may receive a score of 0.

CHAPTER 8

Creating and Loading Occupancy Grid Maps (SLAM)

Author: Dr. Eduardo de Jesús Dávila Meza.
EduardoDavila-AI-PhD

Abstract - All the source files referenced in this chapter are available in our ROS 2 course GitHub repository, located in the `ros2_ws/src` folder, corresponding to the ROS 2 packages with the `s8` prefix.

8.1 Introduction

Simultaneous localization and mapping (SLAM) is a advanced technique where the robot **simultaneously builds a map of its environment** while keeping track of its position within that map, as shown in Figure 8.1. Unlike odometry, which relies solely on internal sensor data, **SLAM incorporates external environmental data** to correct drift and improve accuracy.

Key Features of SLAM:

- **Map building:** It creates a map of the environment (either a 2D grid or a 3D model) as the robot moves.
- **Localization:** It tracks the robot's position within that map.
- **Sensor fusion:** SLAM combines data from **multiple sensors** (e.g., LiDAR, cameras, depth sensors) to estimate both the environment and the robot's location accurately.
- **Loop closure:** SLAM detects when the robot has revisited a previously seen location and corrects errors from odometry drift.

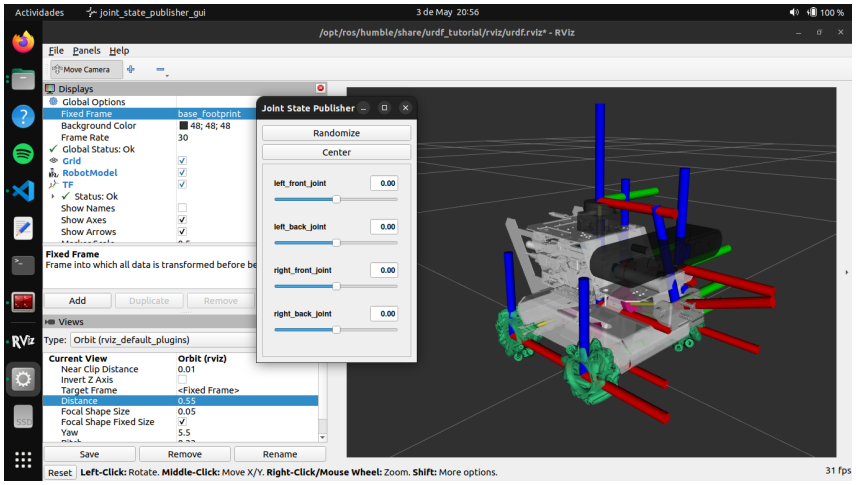


Figure 8.1: Visualization of the RDK X3 Robot in RViz building an occupancy grid map by using a LiDAR sensor.

Applications:

- Used in autonomous navigation (e.g., self-driving cars, mobile robots).
- Essential for robots that need to operate in large or unstructured environments where GPS or other external positioning systems are not available.
- Common in mapping tasks, such as creating floor plans or 3D models of environments.

8.1.1 Occupancy Grid Map

In the context of ROS 2, "creating and loading occupancy grid maps" refers to the processes of generating and utilizing structured representations of an environment, which are crucial for enabling autonomous navigation in mobile robots.

Creating Grid Maps

Grid maps are two-dimensional representations of an environment, divided into a grid where each cell contains information about that specific area. There are two primary types:

1. **Occupancy Grid Maps:** These maps indicate whether a cell is occupied, free, or unknown. They are typically generated using SLAM techniques. For instance, the `slam_toolbox` package in ROS 2 Humble allows for the creation of occupancy grid maps by processing sensor data as the robot explores its environment.
2. **Multi-layered Grid Maps:** These provide more detailed information by storing multiple data layers per cell, such as elevation, surface normals, or traversability. The `grid_map` library facilitates the creation and management of such maps, which are particularly useful for robots operating in complex terrains .

Loading Grid Maps

Once a grid map is created, it needs to be loaded into the robot's navigation system for use in tasks like path planning and obstacle avoidance. In ROS 2 Humble:

- The **Map Server** in the Navigation2 (Nav2) stack is responsible for loading and providing access to occupancy grid maps. It can load maps from files during startup or dynamically at runtime using the `load_map` service.
- For multi-layered grid maps, the `grid_map` library offers tools to load and manage these maps, enabling advanced navigation strategies that consider various environmental factors.

Practical Application

In a typical workflow:

1. **Map Creation:** A robot explores its environment using sensors (like LiDAR or cameras), and a SLAM algorithm processes this data to create a grid map.
2. **Map Loading:** The generated map is loaded into the robot's navigation system using the Map Server or appropriate tools from the `grid_map` library.
3. **Navigation:** The robot uses the loaded map to plan paths, avoid obstacles, and navigate effectively within the environment.

Understanding and effectively utilizing grid maps in ROS 2 Humble is essential for developing robust autonomous navigation capabilities in mobile robots.

8.2 Prerequisites

8.2.1 Workspace and Package Creation

Before getting started, make sure 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:

- Python package to contain the pose publisher and occupancy grid map nodes nodes:

```
$ ros2 pkg create s8_py_slam --build-type ament_python --dependencies rclpy
↪  geometry_msgs sensor_msgs tf2_ros tf_transformations nav_msgs --license
↪  Apache-2.0 --description "SLAM simulation package"
```

Feel free to replace `s8_py_slam` with a name of your preference.

The parameter (`.yaml`), frame (`.pdf`), launch (`.py`), map (`.yaml` and `.pgm`), mesh (`.STL`), RViz (`.rviz`), URDF (`.urdf`), and world (`.world`) files should be organized into directories named `config`, `frames`, `launch`, `maps`, `meshes`, `rviz`, `urdf`, and `worlds`, respectively. Create these directories in the Python package:

```
$ mkdir config frames launch maps meshes rviz urdf worlds
```

8.2.2 Installing Required Packages

Before proceeding, ensure that the following library and packages are installed. Update your package list (apt repository) and install them with:

```
$ sudo apt update
$ sudo apt install ros-humble-gazebo*
$ sudo apt install ros-humble-turtlebot3*
$ sudo apt install ros-humble-slam-toolbox
$ sudo apt install rox-humble-nav2-map-server
$ sudo apt install ros-humble-nav2-lifecycle-manager
$ pip3 install backports.zoneinfo[tzdata] # For Python < 3.9 (For Foxy/Ubuntu 20.04 with
↪ Python 3.8)
$ pip3 install zoneinfo # For Python 3.9+ (For Humble/Ubuntu 22.04 with Python 3.10)
```

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

```
$ echo "source /usr/share/gazebo/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
$ cat ~/.bashrc # Optionally to verify the sourcing command was correctly added
```

Additionally, from the root of your workspace, use `rosdep` to check for any other missing dependencies before building:

```
$ rosdep install -i --from-path src --rosdistro humble -y
```

8.3 Creating a ROS 2 Pose Publisher Node with Python

8.3.1 Setting Up the Python Node

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

```
$ cd ros2_ws/src/s8_py_slam/s8_py_slam
```

Replace `s8_py_slam` with your actual package name.

Creating the Python Node File:
Create a new Python file for your pose publisher node, for example, `pose_publisher.py`:

```
$ touch pose_publisher.py
```

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

```
$ ls
```

You should now see the `pose_publisher.py` file.

8.3.2 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 `pose_publisher.py` file for editing.

Implementing the Node Code:

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

Pose Publisher in Python:

```python
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from tf2_ros import TransformListener, Buffer
from geometry_msgs.msg import PoseStamped
from geometry_msgs.msg import Point

class PosePublisher(Node):
    def __init__(self):
        super().__init__('py_pose_publisher_node')
        self.tf_buffer = Buffer()
        self.tf_listener = TransformListener(self.tf_buffer, self)
        self.pose_pub = self.create_publisher(PoseStamped, '/pose_stamped', 10)
        self.timer = self.create_timer(0.033, self.publish_pose)

    def publish_pose(self):
        try:
            now = rclpy.time.Time()
            trans = self.tf_buffer.lookup_transform('odom', 'base_footprint', now)
            pose_msg = PoseStamped()
            pose_msg.header.stamp = self.get_clock().now().to_msg()
            pose_msg.header.frame_id = 'map'
            position = Point()
            position.x = trans.transform.translation.x
            position.y = trans.transform.translation.y
            position.z = trans.transform.translation.z
            pose_msg.pose.position = position
            pose_msg.pose.orientation = trans.transform.rotation
            self.pose_pub.publish(pose_msg)
        except Exception as e:
            self.get_logger().warn(f'Could not transform: {e}')
```

```
def main():
    rclpy.init()
    node = PosePublisher()
    try:
        rclpy.spin(node)
    except Exception as e:
        node.get_logger().error(f'Exception caught: {e}')
    finally:
        node.destroy_node()
        rclpy.shutdown()

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

8.3.3 Integrating the Python Pose Sever Node into the Package

Modify `setup.py` of your Python package (`s8_py_slam`) to add an executable for the pose publisher node:

```
entry_points={
    'console_scripts': [
        "pose_publisher_exe = s8_py_slam.pose_publisher:main"
    ],
},
```

8.3.4 Summary of Python Pose Publisher Node Key Identifiers

- `pose_publisher.py`: Name of the source file for the Python pose publisher node.
- `py_pose_publisher_node`: Name of the Python pose publisher node as defined in the constructor of the Python class.
- `pose_publisher_exe`: Name of the executable defined in `setup.py`.
- `/pose_stamped`: Name of the topic provided by the Python publisher node.

8.4 Launch File Definition

8.4.1 Spawning Robot

Within the `s8_py_slam/launch` directory, create (or open) a file named `spawn_robot_launch.py` with the following structure:

```python
import os
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch_ros.actions import Node
from ament_index_python.packages import get_package_share_directory

def generate_launch_description():
    # Path to package
    pkg_share = get_package_share_directory('s8_py_slam')

    # Path to the URDF file
    # robot_plugin or cylinder_plugin
    urdf_plugin_path = os.path.join(pkg_share, 'urdf', 'robot_plugin.urdf')

    # Launch file for Gazebo
    gazebo_launch = os.path.join(
        get_package_share_directory('gazebo_ros'), 'launch', 'gazebo.launch.py')

    # Path to your custom world file
    # robot_world or tec_warehouse
    world_path = os.path.join(pkg_share, 'worlds', 'tec_warehouse.world')

    return LaunchDescription([
        # 1) Start Gazebo with the specified world and verbosity
        IncludeLaunchDescription(
            PythonLaunchDescriptionSource(gazebo_launch),
            launch_arguments={
                'world': world_path,
                # 'verbose': 'true'
            }.items(),
        ),

        # 2) Spawn the robot model
        Node(
            package='gazebo_ros',
            executable='spawn_entity.py',
            arguments=[
                '-entity', 'robot_lidar',
                '-file', urdf_plugin_path,
                '-x', '0.5',
                '-y', '0.5',
                '-z', '90.0'
            ],
            output='screen'
        ),
    ])
```

This launch file starts Gazebo with the following features:

- Starts Gazebo trough the `gazebo.launch.py` launch file from the `gazebo_ros` package.
- Loads the URDF model (RDK X3 robot or cylinder robot) in the Gazebo environment by using the `spawn_entity.py` node from the `gazebo_ros` package, with the initial position at $(x = 0.5, y = 0.5, z = 90.0)$ Cartesian coordinates.

- Loads the world model (`robot_world` or `tec_warehouse`) in the Gazebo environment.

8.4.2 Displaying Robot

Within the `s8_py_slam/launch` directory, create (or open) a file named `display_robot_launch.py` with the following structure:

```
import os
from launch import LaunchDescription
from ament_index_python.packages import get_package_share_directory
from launch_ros.actions import Node

def generate_launch_description():
    # Path to package
    pkg_share = get_package_share_directory('s8_py_slam')

    # Path to the URDF file
    # robot or cylinder_plugin
    urdf_path = os.path.join(pkg_share, 'urdf', 'robot.urdf')

    # Path to RViz configuration file
    rviz_config_path = os.path.join(pkg_share, 'rviz', 'slam.rviz')

    # Read URDF files
    with open(urdf_path, 'r') as infp:
        robot_desc = infp.read()

    return LaunchDescription([
        # 1) Publish the URDF to /robot_description
        Node(
            package='robot_state_publisher',
            executable='robot_state_publisher',
            name='robot',
            output='screen',
            parameters=[{'robot_description': robot_desc,
                         'use_sim_time': True
                         }],
        ),

        # 2) RViz2
        Node(
            package='rviz2',
            executable='rviz2',
            name='rviz2',
            arguments=['-d', rviz_config_path],
            output='screen',
        ),
    ])
```

This launch file starts two nodes:

- The `robot` node (`robot_state_publisher`), which publishes the URDF model and the transformations of either RDK X3 robot or cylinder robot.

- Starts RViz trough the `rviz2` node from the `rviz2` package, loading the `slam.rviz` configuration file.

8.5 Building the Package and Running the Node

After making the above edits and saving all the changes, go to the root of your workspace (e.g., `/ros2_ws`) and build the package:

```
$ colcon build --packages-select s8_py_slam
```

Then, in five terminals, source the workspace environment in each, and proceed as follows:

- In **Terminal 1**, spawn the robot in the Gazebo environment through the `spawn_robot_launch.py` launch file:

  ```
  $ source install/setup.bash
  $ ros2 launch s8_py_slam spawn_robot_launch.py
  ```

- In **Terminal 2**, display the robot in the RViz environment through the `display_robot_launch.py` launch file:

  ```
  $ source install/setup.bash
  $ ros2 launch s8_py_slam display_robot_launch.py
  ```

- In **Terminal 3**, start the SLAM node from the `slam_toolbox` package:

  ```
  $ source install/setup.bash
  $ ros2 launch slam_toolbox online_async_launch.py use_sim_time:=True
  ```

- In **Terminal 4**, run the pose publisher node:

  ```
  $ source install/setup.bash
  $ ros2 run s8_py_slam pose_publisher_exe
  ```

- In **Terminal 5**, preferably at a separate window, run the teleop node to control the robot with the keyboard:

  ```
  $ source install/setup.bash
  $ ros2 run teleop_twist_keyboard teleop_twist_keyboard
  ```

  Now, pressing the indicated keys will move the robot in the desired direction and velocity.

Additionally, open two terminals for diagnostics and introspection tools:

- **Terminal 6:** Use the `view_frames` tool from the `tf2_tools` package to visualize the current coordinate frame hierarchy (tree):

  ```
  $ ros2 run tf2_tools view_frames
  ```

- **Terminal 7:** Launch the `rqt_graph` tool to view the current ROS 2 node and topic graph:

  ```
  $ rqt_graph
  ```

8.6 Practice Assignment: `tf2` Library (Robot Network)

In this assignment, you will run the ROS 2 SLAM simulation, as described in Sections 8.3 and 8.4. Capture evidence of successful execution using three screenshots as PNG files and one PDF file, and submit them in the designated assignment area on Canvas (within the ROS 2 module). The filenames must follow this format:

`FirstnameLastname_evidence_sY_Z.png`

where `sY` correspond to the session number, and `Z` is the evidence number. For example, correct filenames would be:

`EduardoDavila_evidence_s8_1.png`,
`EduardoDavila_evidence_s8_2.pdf`,
`EduardoDavila_evidence_s8_3.png`,
`EduardoDavila_evidence_s8_4.png`.

8.6.1 Executing ROS 2 SLAM Simulimulation

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

```
$ colcon build --packages-select s8_py_slam
```

Then, in five terminals, source the workspace environment in each, and proceed as follows:

- In **Terminal 1**, spawn the robot in the Gazebo environment through the `spawn_robot_launch.py` launch file:

  ```
  $ source install/setup.bash
  $ ros2 launch s8_py_slam spawn_robot_launch.py
  ```

- In **Terminal 2**, display the robot in the RViz environment through the `display_robot_launch.py` launch file:

  ```
  $ source install/setup.bash
  $ ros2 launch s8_py_slam display_robot_launch.py
  ```

- In **Terminal 3**, start the SLAM node from the `slam_toolbox` package:

  ```
  $ source install/setup.bash
  $ ros2 launch slam_toolbox online_async_launch.py use_sim_time:=True
  ```

- In **Terminal 4**, run the pose publisher node:

  ```
  $ source install/setup.bash
  $ ros2 run s8_py_slam pose_publisher_exe
  ```

- In **Terminal 5**, preferably at a separate window, run the teleop node to control the robot with the keyboard:

  ```
  $ source install/setup.bash
  $ ros2 run teleop_twist_keyboard teleop_twist_keyboard
  ```

  Now, pressing the indicated keys will move the robot in the desired direction and velocity.

You should observe that:

1. The robot URDF model is spawn inside the simulated environment in Gazebo, along with the `libgazebo_ros_ray_sensor.so` plugin that simulates a LiDAR sensor.
2. The `robot` (`robot_state_publisher`) node publishes the URDF model and the coordinate transforms for visualization in RViz, as simulated in Gazebo.
3. The occupancy grid map is displayed through the `/map` topic in the Map display in RViz, and is being created while the robot is moved across the entire simulated environment in Gazebo.

8.6.2 Visualizing the frames with `view_frames`

To observe the current coordinate frame hierarchy, use the `image_frames` tool from the `tf2_tools` package. Then, in **Terminal 6**, execute the following command (without the need to enter the workspace directory):

```
$ ros2 run tf2_tools view_frames
```

This tool will generate a PDF file in the current directory of the terminal. The file provides a graphical representation of the current coordinate frame tree from the robot URDF model.

8.6.3 Visualizing the ROS Graph with `rqt_graph`

To verify that all the nodes are correctly connected, use the `rqt_graph` tool. Then, in **Terminal 7**, execute the following command (without the need to enter the workspace directory):

```
$ ros2 run rqt_graph rqt_graph
```

or simply:

```
$ rqt_graph
```

This will visualize the ROS 2 graph, showing topics connecting the robot network nodes, as described above.

8.6.4 Submission Instructions

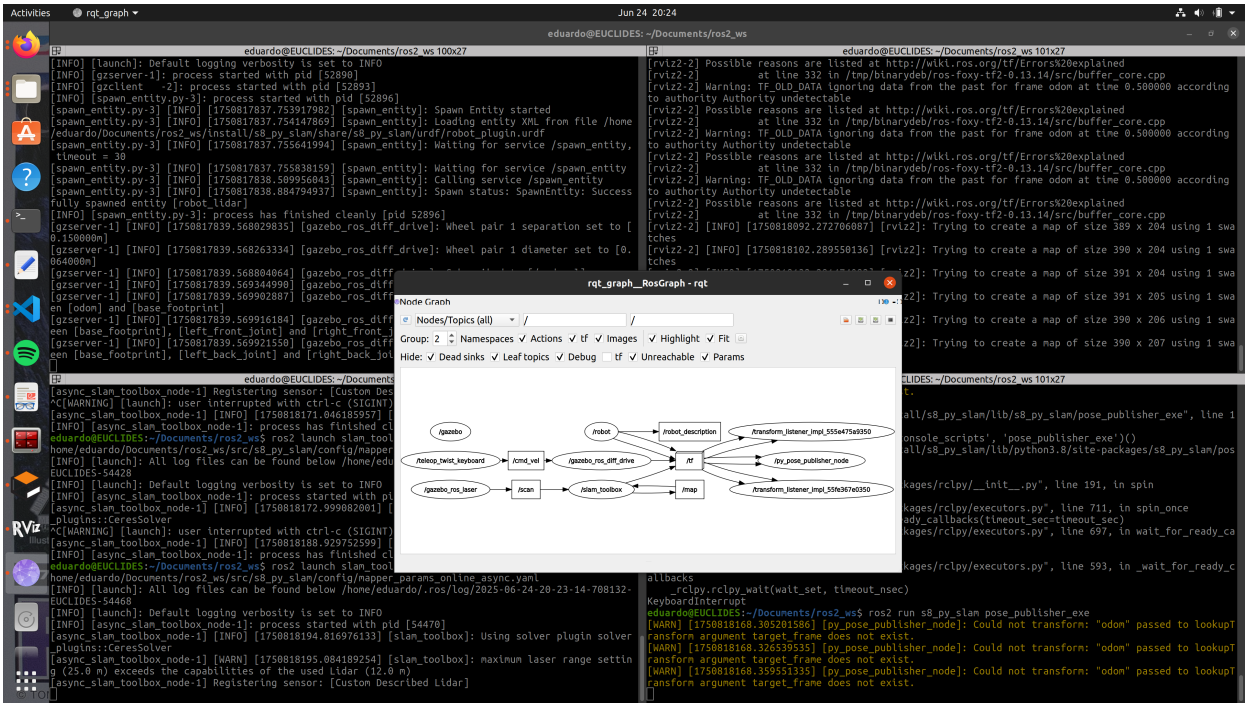


Figure 8.2: Example screenshot showing terminal outputs and the ROS 2 graph of the SLAM simulation in action.

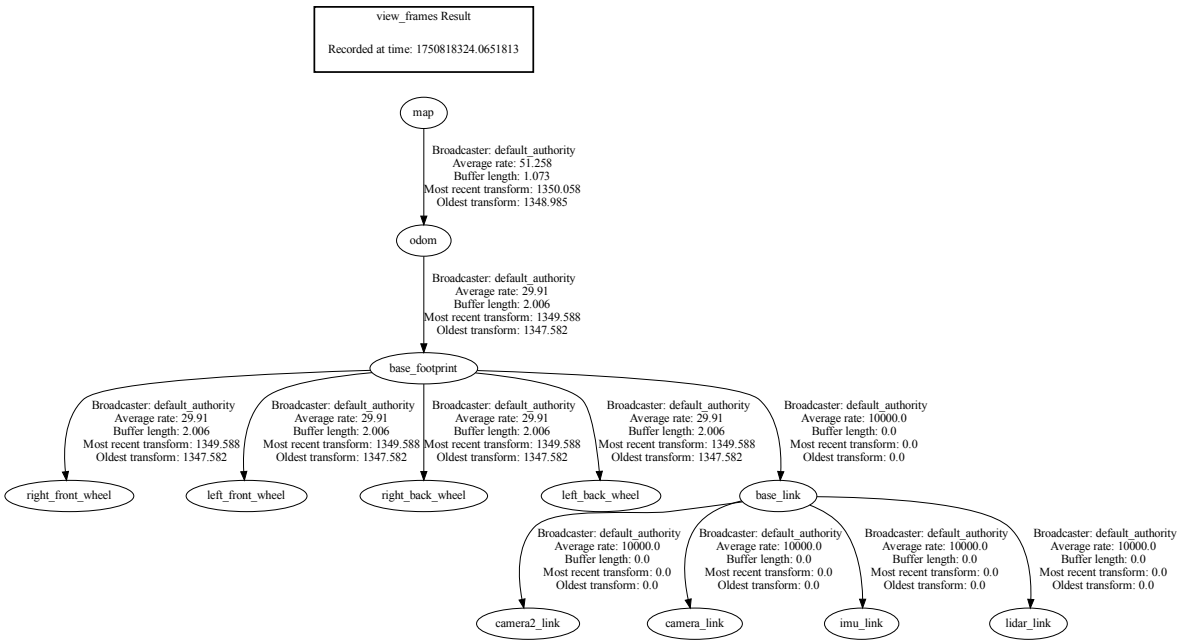


Figure 8.3: PDF showing the graphical representation of the current coordinate frame tree.

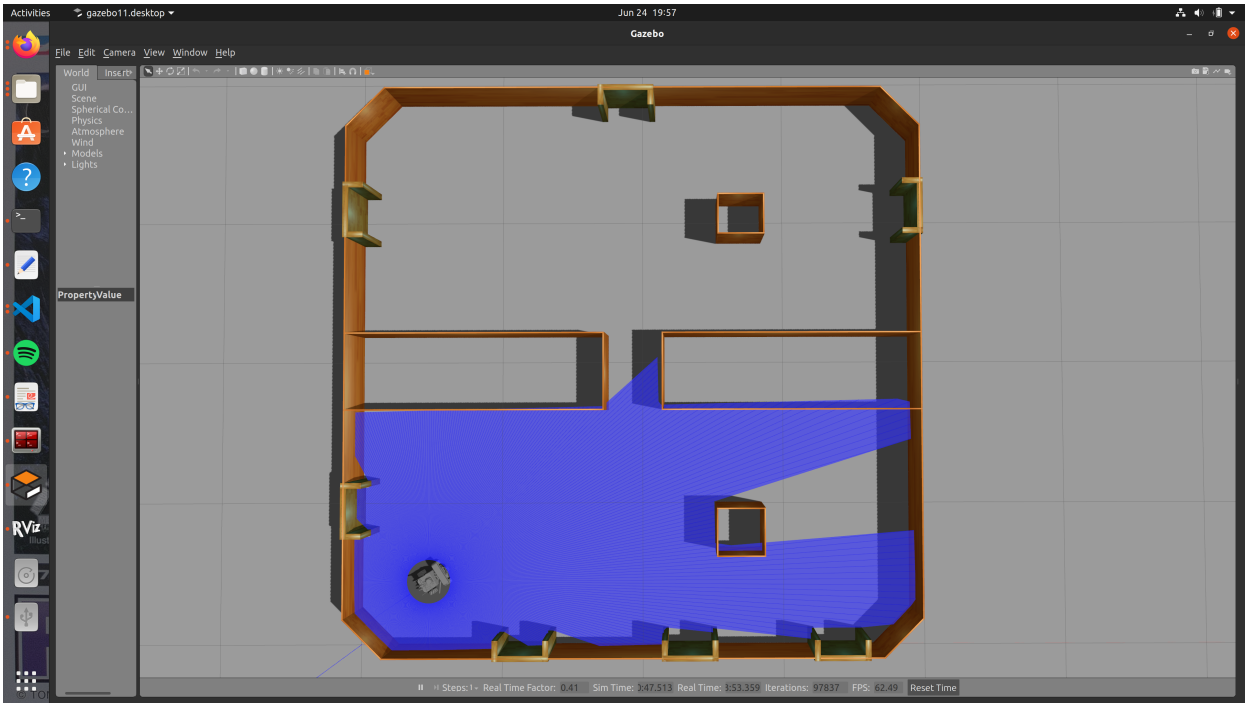


Figure 8.4: Example screenshot showing the simulation of the RDK X3 Robot in Gazebo.

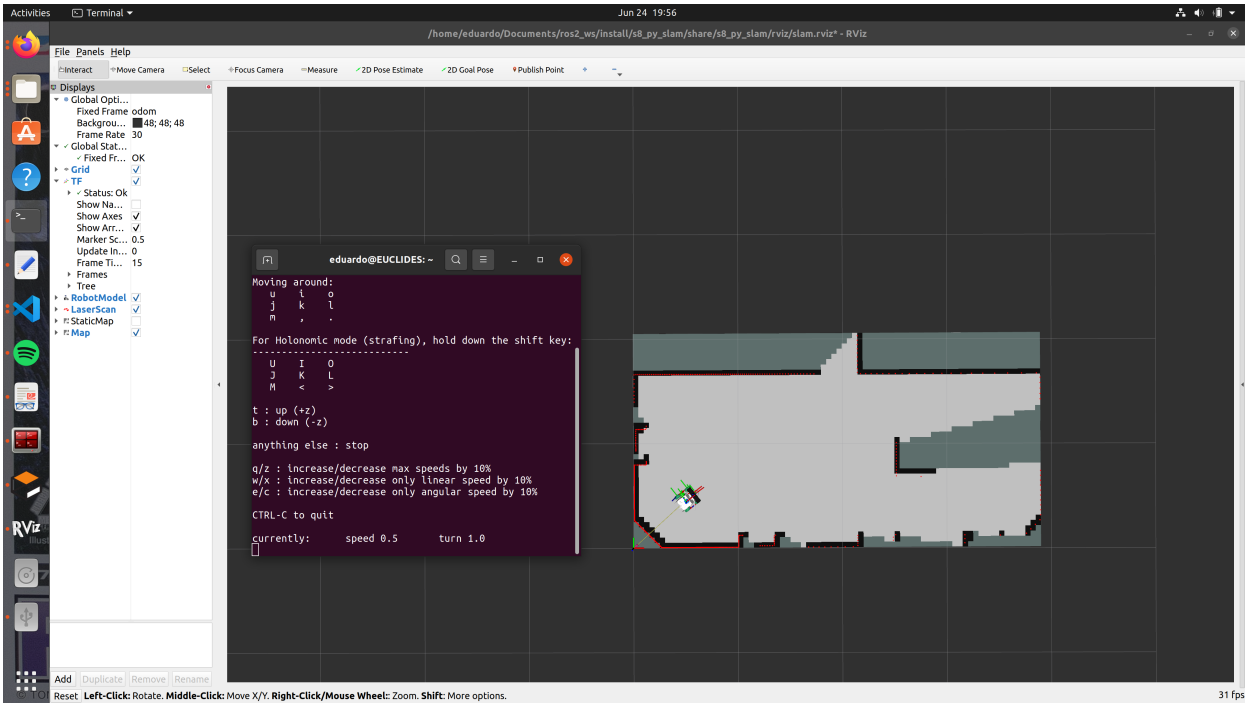


Figure 8.5: Example screenshot showing the visualization of the RDK X3 Robot in RViz.

1. Run the ROS 2 SLAM simulation as described above.
2. As first evidence, capture a **screenshot showing the terminal outputs and the ROS 2 graph** from either the C++ or Python nodes, as described in Sections 8.6.1 and 8.6.3. See Figure 8.2 for reference.
3. As second evidence, locate the PDF file generated by the `view_frames` tool from the `tf2_tools` package, as described in Section 8.6.2. See Figure 8.3 for reference.
4. As third evidence, capture a **screenshot showing the simulation of the RDK X3 Robot in Gazebo**. See Figure 8.4 for reference.
5. As fourth evidence, capture a **screenshot showing the visualization of the RDK X3 Robot in RViz**. See Figure 8.5 for reference.
6. Save the screenshots as PNG files.
7. Name the files following this format: `FirstnameLastname_evidence_sX_Y.png` (replace `X` with the session number and `Y` with the evidence number).
8. Submit the files as specified by the course guidelines on Canvas.

Note: Your machine's username and hostname must be visible in the screenshots to verify the authenticity of your submission, as shown in Figure 8.2. Any submission that appears copied, unclear, or altered will be considered invalid and may receive a score of 0.

APPENDIX A

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
`sudo apt update`	Updates the list of available packages and their versions.
`sudo apt upgrade`	Installs the latest versions of all installed packages.
`sudo apt install <pkg_name>`	Installs the specified package.
`sudo apt install ./<file>.deb`	Installs a package from a local `.deb` file.
`sudo apt install ros-humble-desktop`	Installs the full desktop version of ROS 2 Humble.
`sudo apt install ros-dev-tools`	Installs development tools for ROS 2.
`sudo apt install ros-humble-turtlesim`	Installs the `turtlesim` package for ROS 2 Humble.
`sudo apt install python3-colcon-common-extensions`	Installs common extensions for `colcon` to facilitate package building.
`sudo apt install terminator`	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
`source /opt/ros/humble/setup.bash`	Sets up the environment for ROS 2 Humble in the current terminal session.
`source /usr/share/colcon_argcomplete/hook/colcon-argcomplete.bash`	Sets up the autocompletion for `colcon` in the current terminal session.
`echo "source/opt/ros/humble/setup.bash" >> ~/.bashrc`	Adds the ROS 2 Humble environment setup to the `.bashrc` file for future terminal sessions.
`echo "source/usr/share/colcon_argcomplete/hook/colcon-argcomplete.bash" >> ~/.bashrc`	Enables autocompletion for `colcon` in the `.bashrc` file for future terminal sessions.
`source ~/.bashrc`	Reloads the `.bashrc` file to apply changes without restarting the terminal.
`cat ~/.bashrc`	Displays the contents of the `.bashrc` file.
`gedit ~/.bashrc`	Opens the `.bashrc` file in the `gedit` text editor.

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

A.3 Workspace Creation and Building

Command	Description
`mkdir -p <ws_name>/src`	Creates the workspace directory and its `src` subdirectory.
`cd <ws_name>`	Navigates to the workspace directory.
`colcon build`	Builds all packages in the workspace.
`colcon build --symlink-install`	Builds packages using symbolic links, useful for development purposes.
`source ./install/setup.bash`	Sets up the environment for the built workspace in the current terminal session.
`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
`cd <ws_name>/src`	Navigates to the workspace's `src` directory.
`ros2 pkg create <pkg_name> --build-type ament_cmake --dependencies rclcpp std_msgs --license Apache-2.0 --description "Your package description here"`	Creates a new C++ package with specified dependencies, license, and description.
`ros2 pkg create <pkg_name> --build-type ament_python --dependencies rclpy std_msgs --license Apache-2.0 --description "Your package description here"`	Creates a new Python package with specified dependencies, license, and description.
`ls`	Lists the files and directories in the current directory.
`cd ..`	Navigates one directory up.
`rosdep install -i --from-path src --rosdistro humble -y`	In the root of the workspace, checks for missing dependencies before building.
`colcon build --packages-select <pkg_name>`	Builds only the `<pkg_name>` package.
`colcon build --packages-select <pkg_name> --symlink-install`	Builds the `<pkg_name>` package with symbolic link installation.
`cd src/<pkg_name>`	Navigates to the `<pkg_name>` directory.
`code package.xml`	Opens the `package.xml` file in Visual Studio Code.
`code CMakeLists.txt`	Opens the `CMakeLists.txt` file in Visual Studio Code.
`code setup.py`	Opens the `setup.py` file in Visual Studio Code.
`cd <ws_name>/src/<pkg_name>/src`	Navigates to the `src` subdirectory of a C++ package.
`cd <ws_name>/src/<pkg_name>/<pkg_name>`	Navigates to the `<pkg_name>` subdirectory of a Python package.
`touch <filename>.cpp`	Creates a new C++ source file named `<filename>`.
`touch <filename>.py`	Creates a new Python source file named `<filename>`.
`cd ../../..`	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
`ros2 run <pkg_name> <node_name>`	Runs the specified node from the specified package.
`ros2 run <pkg_name> <node_name> <args>`	Runs the specified node from the specified package with specified arguments.
`ros2 service call <service_name> <service_type> <args>`	Calls the service `<service_name>` from the specified package with specified arguments.
`ros2 node list`	Lists all active nodes in the ROS 2 system.
`ros2 node info <node_name>`	Displays detailed information about the specified node.
`ros2 topic list`	Lists all active topics in the ROS 2 system.
`ros2 topic info <topic_name>`	Displays detailed information about the specified topic.
`ros2 topic echo <topic_name>`	Echoes the messages of a specified topic.
`ros2 topic pub <topic_name> <msg_type> <args>`	Publishes a message to the specified topic.
`ros2 topic hz topic_name`	Displays the message frequency of a specified topic.
`ros2 interface show <pkg_name>/msg/<custom_msg_name>`	Shows the field structure of a custom message type in the specified package.
`ros2 interface show <pkg_name>/srv/<custom_srv_name>`	Shows the request/response structure of a custom service type in the specified package.
`ros2 run rqt_graph rqt_graph rqt_graph`	Runs the `rqt_graph` tool to visualize the ROS 2 nodes and topics.
`ros2 run rqt_service_caller rqt_service_caller`	Runs the `rqt_service_caller` 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 B

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++

```cpp
#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

```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++

```cpp
#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

```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++

```cpp
#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

```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.

APPENDIX C

Configuring ROS 2 Package Paths for URDF Visualizer in VS Code

Author: Dr. Eduardo de Jesús Dávila Meza.
EduardoDavila-AI-PhD

To correctly visualize URDF or Xacro files that utilize `package://` URIs (Uniform Resource Identifiers), it is essential to configure the URDF Visualizer extension in Visual Studio Code to recognize the paths of your ROS 2 packages. This ensures that all resources referenced in your robot description files are accurately located and rendered.

C.1 Steps to Add a ROS 2 Package Path

1. Open your project folder in Visual Studio Code.
2. Navigate to the Extensions view by clicking on the Extensions icon in the Activity Bar or pressing `Ctrl+Shift+X`.
3. Search for URDF Visualizer and ensure it is installed.
4. Click on the gear icon (Manage) next to the URDF Visualizer extension and select **Extension Settings**.
5. In the settings, locate the **Urdf-visualizer: Packages** section.
6. Click on **Edit in settings.json** to open the `settings.json` file.
7. Within the `urdf-visualizer.packages` object, add your package path without removing existing entries. You can specify:
   - A relative path:
     ```
     "package_name": "${workspaceFolder}/src/package_name"
     ```
   - An absolute path:
     ```
     "package_name": "/absolute/path/to/package_name"
     ```

8. Save the `settings.json` file and close it.
9. To visualize your URDF or Xacro file:
    - Open the file in VS Code.
    - Click on the eye icon in the top-right corner.
    - Alternatively, press `Ctrl+Shift+P` to open the Command Palette, type `URDF:Preview URDF/Xacro`, and select it.
10. If the URDF Visualizer is already open, click on the reload button to apply the new settings.

C.2 Additional Notes

- If your URDF or Xacro file references multiple packages, ensure all are listed in the `urdf-visualizer.packages` configuration.
- Use absolute paths if your packages are located outside the current workspace.
- The extension supports both URDF and Xacro files, provided the package paths are correctly configured.

For more detailed information, refer to the official documentation: URDF Visualizer Extension.