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

Chapter 1

ROS 2 Environment Setup and Development Tools

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

1.1 Introduction to ROS 2

1.1.1 Basic Concepts

The Robot Operating System (ROS) is not an actual operating system but a flexible *framework* for writing robot software. It provides a collection of software libraries, tools, and conventions to simplify the task of creating complex and robust robot applications. From drivers and state-of-the-art algorithms to powerful developer tools, ROS has the open source tools you need for your next robotics project. Some of its main advantages include:

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

1.1.2 Some ROS Distributions

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

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

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

1.1.3 ROS Distribution for the Course

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

Figure 1.1: ROS 2 Humble Hawksbill Logo.

1.2 Installation of Ubuntu 22 and ROS 2 Humble Hawksbill

1.2.1 Installing Ubuntu 22.04.5 LTS (Jammy Jellyfish)

1. **Download the Universal USB Installer**
   Visit the official Universal USB Installer website to download the tool. Follow the instructions provided on the website or refer to a YouTube tutorial for creating a bootable USB.
2. **Download the Ubuntu ISO**
   Download the Ubuntu 22.04.5 LTS (Jammy Jellyfish) ISO from the official Ubuntu releases page.
3. **Create a Bootable USB Drive**
   Use the Universal USB Installer tool with the downloaded ISO to create a bootable USB stick. Follow the on-screen prompts to properly set up the drive.
4. **Install Ubuntu on Your Machine**
   Boot your computer from the USB drive. Follow the Ubuntu installation wizard to install Ubuntu 22.04.5 LTS and configure your system settings as needed.

1.2.2 Installing ROS 2 Humble Hawksbill

Recommended Method: Using Debian Packages

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

Instructions:

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

```
$ locale  # check for UTF-8

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

$ locale  # verify settings
```

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

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

Now, add the ROS 2 GPG key with apt:

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

Then, add the repository to your sources list:

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

Install ROS 2 Packages
Update your apt repository caches after setting up the repositories:

```
$ sudo apt update
```

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

```
$ sudo apt upgrade
```

> ⚠ **Warning:**
> Due to early updates in Ubuntu 22.04, it is important that `systemd` and `udev`-related packages are updated before installing ROS 2. Installing ROS 2's dependencies on a freshly installed system without upgrading can trigger the removal of critical system packages. Please refer to ros2/ros2#1272 and Launchpad #1974196 for more information.

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

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

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

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

Environment Setup – Sourcing the Setup Script
Set up your environment by sourcing the following file (replace `.bash` with your shell if not using bash):

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

Alternative: Building from Source

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

Sourcing the Setup Script

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

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

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

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

1.3 Try Some Examples

1.3.1 Talker-Listener

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

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

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

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

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

1.3.2 Turtlesim: Publish and Move a Turtle

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

First, ensure that the package is installed:

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

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

```
$ ros2 run turtlesim turtlesim_node
```

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

```
$ ros2 run turtlesim turtle_teleop_key
```

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

1.4 Configuring the Development Environment

1.4.1 Installing Visual Studio Code

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

1. **Using the Ubuntu Software Center**
   (a) Open the *Ubuntu Software* application from the applications menu.
   (b) In the search bar, type *Visual Studio Code*.
   (c) Locate *Visual Studio Code* in the search results and click *Install*.
2. **Using the Official .deb Package**
   (a) Visit the official VS Code download page: https://code.visualstudio.com/.
   (b) Click on the *.deb* package suitable for Debian/Ubuntu.
   (c) Once downloaded, open a terminal and navigate to the directory containing the downloaded file.
   (d) Install the package using:
   ```
   $ sudo apt install ./<file>.deb
   ```
   Replace `<file>` with the actual filename. This method ensures that VS Code is added to your system repositories and receives updates automatically.

1.4.2 Installing Recommended VS Code Extensions

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

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

To install these extensions:

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

1.4.3 Installing Terminator for Multiple Terminals

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

1. Open a terminal.
2. Update the package list:
```
$ sudo apt update
```
3. Install Terminator:
```
$ sudo apt install terminator
```
4. Launch Terminator from the applications menu, by typing `terminator` in the terminal, or by pressing `Ctrl+Alt+T`.

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

1.5 `colcon` Configuration

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

1.5.1 Installing `colcon`

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

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

1.5.2 Enabling `colcon` Argument Completion

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

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

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

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

1.6 Practice Assignment: Running ROS 2 Examples

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

`FirstnameLastname_evidence_wXsY.png`,

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

`EduardoDavila_evidence_w1s1.png`.

1.6.1 Examples to Try

Talker-Listener

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

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

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

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

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

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

Turtlesim: Publish and Move a Turtle

Another example is provided by the `turtlesim` package:

- In Terminal 3, launch the turtlesim node:

```
$ ros2 run turtlesim turtlesim_node
```

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

```
$ ros2 run turtlesim turtle_teleop_key
```

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

1.6.2 Submission Instructions

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

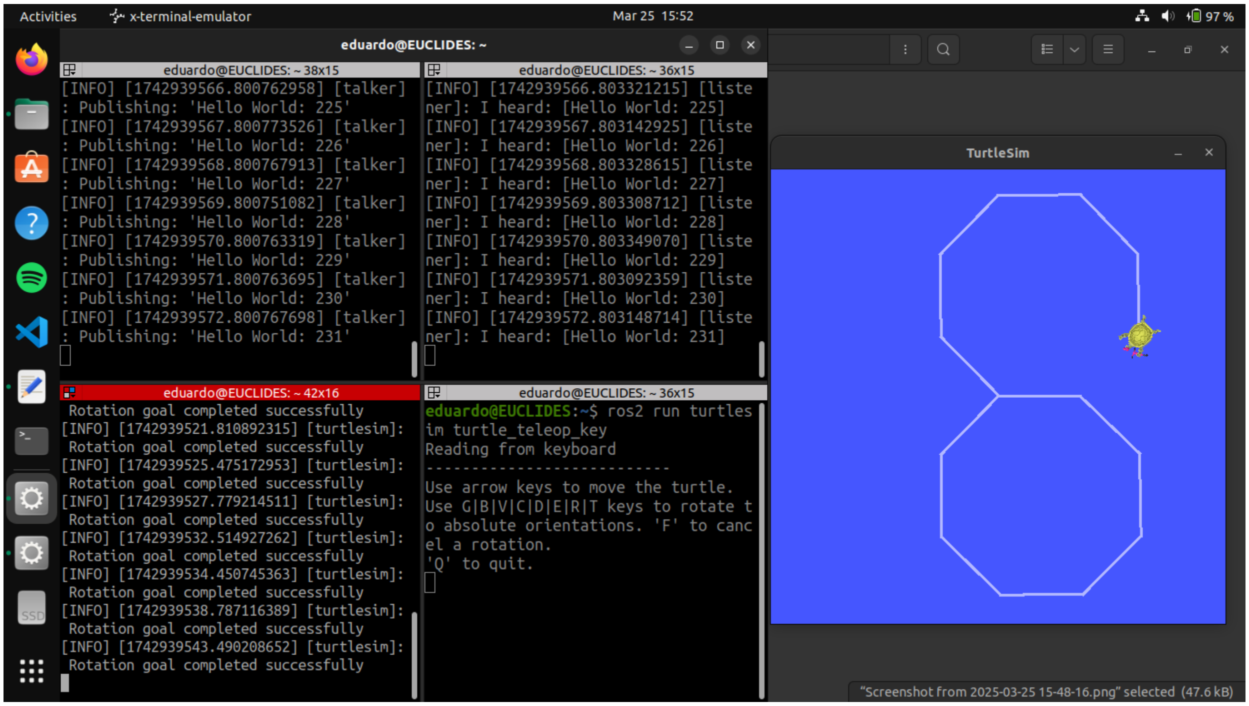

Figure 1.2: Example screenshot showing terminal outputs from both the Talker-Listener example and the Turtlesim node example in action.

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

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

Chapter 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., `ros2 bag record`) can subscribe to topics without interrupting existing communication.

Strongly-Typed Messages

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

1. Strict Data Types: Each field in a message adheres to a predefined data type.
   ```
   uint32 field1
   string field2
   ```
   In this example, `field1` must always be an unsigned 32-bit integer, and `field2` must always be a string.
2. Well-Defined Semantics: Message contents adhere to clear conventions. For example, an IMU message includes a 3-dimensional angular velocity vector, where each component is explicitly defined in radians per second. This ensures consistency and prevents misinterpretation of the transmitted data.

2.2 Workspace Creation and Setup

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

2.2.1 Creating the Workspace

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

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

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

2.2.2 Building the Workspace

Navigate to the workspace directory:

```
$ cd ros2_ws
```

Build the workspace using `colcon`:

```
$ colcon build
```

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

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

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

2.2.3 Sourcing the Workspace Environment

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

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

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

For more details on `colcon` and workspace management, refer to the ROS 2 Colcon tutorial.

2.3 Creating a ROS 2 Package for C++ Nodes

2.3.1 Navigating to the `src` Directory

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

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

2.3.2 Creating the C++ Package

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

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

Let's break down the components of this command:

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

2.3.3 Building the Package

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

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

Alternatively, to build all packages in the workspace:

```
$ colcon build
```

2.3.4 Examining the Package Contents

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

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

You should see the following key files and directories:

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

2.3.5 Detailed Examination of Key Package Files

`package.xml`

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

- `<name>`: Specifies the package's name.
- `<version>`: Indicates the current version of the package.

- `<description>`: Provides a brief overview of the package's purpose.
- `<maintainer>`: Contains contact information for the package maintainer.
- `<license>`: Details the licensing information.
- `<depend>`: Lists build-time and runtime dependencies required by the package.

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

```
$ code package.xml
```

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

`CMakeLists.txt`

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

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

```
$ code CMakeLists.txt
```

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

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

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

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

2.4 Creating a ROS 2 Package for Python Nodes

2.4.1 Navigating to the `src` Directory

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

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

2.4.2 Creating the Python Package

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

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

Let's break down the components of this command:

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

2.4.3 Building the Package

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

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

Alternatively, to build all packages in the workspace:

```
$ colcon build
```

2.4.4 Examining the Package Contents

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

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

You should see the following key files and directories:

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

2.4.5 Detailed Examination of Key Package Files

`package.xml`

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

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

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

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

```
$ code package.xml
```

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

`setup.py`

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

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

```
$ code setup.py
```

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

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

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

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

2.5 Creating a ROS 2 Publisher Node with C++

2.5.1 Setting Up the C++ Node

Navigating to the Package Directory

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

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

Replace `my_cpp_package` with your actual package name.

Creating the C++ Node File

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

```
$ touch publisher_cpp.cpp
```

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

```
$ ls
```

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

2.5.2 Editing the C++ Node

Opening the Workspace in VS Code

Navigate back to the root of your workspace:

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

Then, open the workspace in Visual Studio Code:

```
$ code .
```

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

Implementing the Node Code

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

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

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

using namespace std::chrono_literals;

/* This example creates a subclass of Node and uses std::bind() to register a
* member function as a callback from the timer. */

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

```
    private:
        void timer_callback()
        {
        auto message = std_msgs::msg::String();
        message.data = "Hello, this is Eduardo from the C++ Publisher: " +
 ↪  std::to_string(count_++);
        RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_str());
        publisher_->publish(message);
        }
        rclcpp::TimerBase::SharedPtr timer_;
        rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
        size_t count_;
};

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

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

Breaking Down the Publisher Node Code

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

1. Includes and Namespace

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

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

using namespace std::chrono_literals;
```

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

2. Defining the Node Class

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

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

3. Timer Callback Method

```
    private:
        void timer_callback()
        {
        auto message = std_msgs::msg::String();
        message.data = "Hello, this is Eduardo from the C++ Publisher: " +
 ↪  std::to_string(count_++);
        RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_str());
        publisher_->publish(message);
        }
        rclcpp::TimerBase::SharedPtr timer_;
        rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
        size_t count_;
};
```

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

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

4. The Main Function

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

- `rclcpp::init` initializes the ROS 2 communication.
- `rclcpp::spin` keeps the node active and processes callbacks.
- `rclcpp::shutdown` cleans up resources when the node is stopped.

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

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

2.5.3 Integrating the Publisher Node into the Package

Modifying `CMakeLists.txt`

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

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

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

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

Breaking down each line:

- `find_package(ament_cmake REQUIRED)` locates the ament build system.
- `find_package(rclcpp REQUIRED)` and `find_package(std_msgs REQUIRED)` ensure that the required dependencies are found.
- `add_executable(node_publisher_cpp src/publisher_cpp.cpp)` creates an executable from your source file.
- `ament_target_dependencies(node_publisher_cpp rclcpp std_msgs)` links the required libraries.
- The `install` command specifies where the executable should be installed.

Building the Package

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

```
$ colcon build
```

Alternatively, to build only the C++ package:

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

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

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

or, more specifically:

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

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

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

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

```
$ cd build/my_cpp_package
```

or:

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

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

```
$ ./node_publisher_cpp
```

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

Sourcing the Environment

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

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

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

2.5.5 Running the ROS 2 Publisher Node with ROS 2

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

```
$ ros2 run my_cpp_package node_publisher_cpp
```

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

2.5.6 Summary of Key Identifiers

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

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

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

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

2.6 Creating a ROS 2 Subscriber Node with C++

2.6.1 Setting Up the C++ Node

Navigating to the Package Directory

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

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

Replace `my_cpp_package` with your actual package name.

Creating the C++ Node File

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

```
$ touch subscriber_cpp.cpp
```

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

```
$ ls
```

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

2.6.2 Editing the C++ Node

Opening the Workspace in VS Code

Navigate back to the root of your workspace:

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

Then, open the workspace in Visual Studio Code:

```
$ code .
```

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

Implementing the Node Code

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

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

using std::placeholders::_1;

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

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

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

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

Breaking Down the Subscriber Node Code

1. Includes and Namespace

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

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

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

2. Defining the Node Class

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

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

3. Topic Callback Method

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

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

4. The Main Function

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

- `rclcpp::init`: Initializes the ROS 2 communication.
- `rclcpp::spin`: Keeps the node active and processes callbacks.
- `rclcpp::shutdown`: Gracefully shuts down ROS 2, cleaning up all resources when the node stops.
- `return 0;`: Returns 0 to indicate successful program termination.

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

2.6.3 Integrating the Subscriber Node into the Package

Modifying `CMakeLists.txt`

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

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

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

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

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

Breaking down each line:

- `find_package(ament_cmake REQUIRED)` locates the ament build system.

- `find_package(rclcpp REQUIRED)` and `find_package(std_msgs REQUIRED)` ensure that the required dependencies are found.
- `add_executable(node_subscriber_cpp src/subscriber_cpp.cpp)` creates an executable from your source file.
- `ament_target_dependencies(node_subscriber_cpp rclcpp std_msgs)` links the required libraries.
- The `install` command specifies where the executable should be installed.

Building the Package

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

```
$ colcon build
```

Alternatively, to build only the C++ package:

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

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

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

or, more specifically:

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

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

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

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

```
$ cd build/my_cpp_package
```

or:

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

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

```
$ ./node_subscriber_cpp
```

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

Sourcing the Environment

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

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

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

2.6.5 Running the ROS 2 Subscriber Node with ROS 2

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

```
$ ros2 run my_cpp_package node_subscriber_cpp
```

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

2.6.6 Summary of Key Identifiers

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

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

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

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

2.7 Creating a ROS 2 Publisher Node with Python

2.7.1 Setting Up the Python Node

Navigating to the Package Directory

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

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

Replace `my_py_package` with your actual package name.

Creating the Python Node File

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

```
$ touch publisher_py.py
```

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

```
$ ls
```

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

Making the Python File Executable

Ensure the new Python file is executable:

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

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

2.7.2 Editing the Python Node

Opening the Workspace in VS Code

Navigate back to the root of your workspace:

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

Then, open the workspace in Visual Studio Code:

```
$ code .
```

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

Implementing the Node Code

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

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

class PyPublisher(Node):

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

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

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

    py_publisher = PyPublisher()

    rclpy.spin(py_publisher)

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

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

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

Breaking Down the Publisher Node Code

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

1. Shebang and Imports

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

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

2. Defining the Node Class

```
class PyPublisher(Node):

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

- `class PyPublisher(Node):` defines a new node by subclassing `Node`.
- In the `__init__` method:

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

3. Timer Callback Method

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

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

4. The Main Function

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

    py_publisher = PyPublisher()

    rclpy.spin(py_publisher)

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

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

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

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

2.7.3 Running the Publisher Node Standalone (Optional)

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

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

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

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

to run it as an executable, or

```
$ python3 publisher_py.py
```

to run it as a Python script.

2.7.4 Integrating the Publisher Node into the Package

Modifying `setup.py`

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

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

Breaking down the components of this specification:

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

Building the Package

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

```
$ colcon build
```

Alternatively, to build only the Python package:

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

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

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

or, more specifically:

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

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

Sourcing the Environment

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

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

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

2.7.5 Running the ROS 2 Publisher Node with ROS 2

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

```
$ ros2 run my_py_package node_publisher_py
```

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

2.7.6 Summary of Key Identifiers

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

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

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

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

2.8 Creating a ROS 2 Subscriber Node with Python

2.8.1 Setting Up the Python Node

Navigating to the Package Directory

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

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

Replace `my_py_package` with your actual package name.

Creating the Python Node File

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

```
$ touch subscriber_py.py
```

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

```
$ ls
```

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

Making the Python File Executable

Ensure the new Python file is executable:

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

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

2.8.2 Editing the Python Node

Opening the Workspace in VS Code

Navigate back to the root of your workspace:

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

Then, open the workspace in Visual Studio Code:

```
$ code .
```

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

Implementing the Node Code

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

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

class PySubscriber(Node):

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

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

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

    py_subscriber = PySubscriber()

    rclpy.spin(py_subscriber)

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

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

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

2.8.3 Breaking Down the Subscriber Node Code

1. Shebang and Imports

```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 Subscriber Node Class

```python
class PySubscriber(Node):

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

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

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

3. The Listener Callback Method

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

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

4. The Main Function

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

    py_subscriber = PySubscriber()

    rclpy.spin(py_subscriber)

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

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

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

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

2.8.4 Running the Subscriber Node Standalone (Optional)

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

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

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

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

to run it as an executable, or

```
$ python3 subscriber_py.py
```

to run it as a Python script.

2.8.5 Integrating the Subscriber Node into the Package

Modifying setup.py

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

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

Breaking down the components of this specification:

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

Building the Package

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

```
$ colcon build
```

Alternatively, to build only the Python package:

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

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

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

or, more specifically:

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

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

Sourcing the Environment

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

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

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

2.8.6 Running the ROS 2 Subscriber Node with ROS 2

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

```
$ ros2 run my_py_package node_subscriber_py
```

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

2.8.7 Summary of Key Identifiers

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

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

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

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

2.9 Practice Assignment: Running ROS 2 Nodes

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

`FirstnameLastname_evidence_wXsY.png`,

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

`EduardoDavila_evidence_w1s2.png`.

2.9.1 Executing ROS 2 Publishers and Subscribers

Running the Python Publisher and Subscriber

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

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

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

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

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

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

Running the C++ Publisher and Subscriber

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

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

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

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

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

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

Visualizing the ROS Graph with `rqt_graph`

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

```
$ ros2 run rqt_graph rqt_graph
```

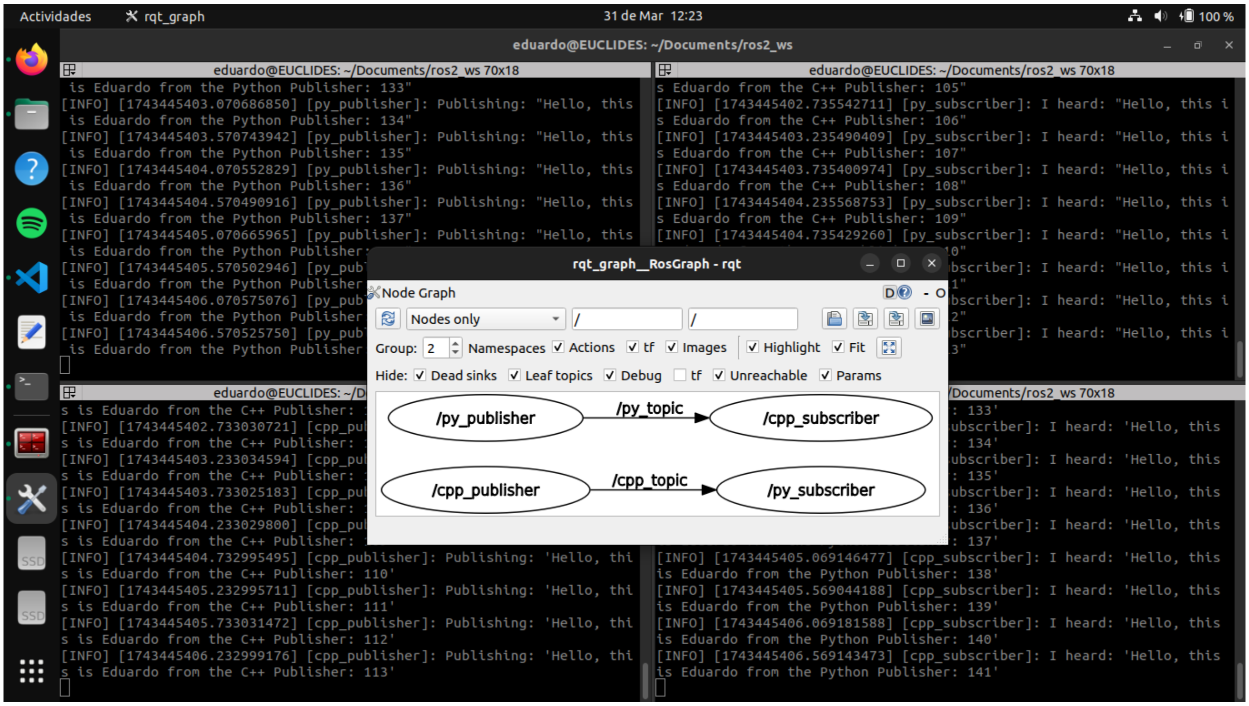

Figure 2.1: Example screenshot showing terminal outputs from the py_publisher, py_subscriber, cpp_publisher, and cpp_subscriber nodes in action.

or simply:

```
$ rqt_graph
```

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

2.9.2 Submission Instructions

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

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