Introduction to Robot Operating System 2 (ROS 2)

EE5111/EE5061

Selected Topics in Industrial Control & Instrumentation

Department of Electrical & Computer Engineering
National University of Singapore

October, 2025

Contents

Chapter 1

Introduction

1.1 About ROS 1

Instead of going directly into ROS 2, its origin, ROS 1 (or just ROS for short), will be briefly introduced here for the better comprehension. **ROS is an open-source, meta-operating system for your robot**. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers. Figure 1.1 is the logo of ROS.

Figure 1.1: ROS Logo

So ROS is a meta-operating system for robots, but you may wonder:

'What is a meta operating system?'

There are many kinds of operating systems (OS). The most commonly used OS like Windows or Linux are **general purpose OS**. Different from these OS, ROS is a **Meta-OS**, which describes a system that performs processes such as scheduling, loading, monitoring and error handling by utilizing a virtualization layer between applications and distributed computing resources.

1.1.1 Advantages and Disadvantages

Because of its particularity, ROS has many advantages as well as disadvantages:

Advantages:

- ROS is an open-source project, which means everyone can make contributions, and for now, it has been with a certain influence. Many famous industrial robot companies such as ABB and KUKA are supporting ROS (KUKA-ROS and ABB-ROS Packages). Besides, some software like MATLAB and LabVIEW also support ROS. Therefore, it is pretty easy to test your own control algorithms with the help of Matlab and KUKA-ROS via the ROS interface.

- ROS also supports different kinds of programming languages like C++ and Python. A prototype model can be easily completed with the help of ROS with any commonly used programming language.
- Many GUI tools in ROS are convenient for users. We can visualize the track of a virtual moving vehicle and we can even simulate the gravity and friction with the help of ROS.

Disadvantages:

- However, ROS is always based on general-purpose operating systems. This kind of operating system is not designed for industrial purposes, so most of them are not real-time, which means ROS is not real-time for control algorithms.

1.2 Entering ROS 2

Since ROS was started in 2007, a lot has changed in the robotics and ROS community. The goal of the ROS 2 project is to adapt to these changes, leveraging what is great about ROS 1 and improving what is not [2]. The ROS 2 ecosystem consists of these four pivotal elements:

1. Tools: ROS 2 offers a suite of developer tools, which cover a wide range of functionalities and accelerate application development. The tools include launch, introspection, debugging, visualization, plotting, logging, and playback.
2. Capabilities: ROS 2 provides advanced features and better supports for the real-time computing.
3. Data Distribution Service (DDS): ROS 2 facilitates communication between nodes via ROS Middleware Interface, which provide real-time communication, scalability, performance enhancement, and security benefits via Data Distribution Service .
4. Ecosystem: The ROS 2 ecosystem is supported by a community of developers, researchers, and industry professionals. The community-driven effort ensures continuous improvement and innovation.

Specifically compared to ROS 1, there are some improvements of ROS 2. Initially, ROS 2 uses Data Distribution Service (DDS) as its underlying communication middleware, which is designed for **real-time systems** and offers higher efficiency, reliability, and low latency compared to ROS 1's XML-RPC. DDS provides a rich set of Quality of Service (QoS) policies that allow you to configure communication behavior to **meet specific requirements**. ROS 2 is designed to be more scalable and reliable than ROS 1, as **it doesn't rely on a central master node**. It allows you to run multiple nodes and processes on the same or different machines, making it easier to distribute tasks and improve performance. ROS 2 supports **a wider range of programming languages** than ROS 1, including C++, Python, Java, C#, and Rust. This makes it more accessible to developers with different programming backgrounds. ROS 2 has a **simplified API** that is easier to learn and use than ROS 1, and the **improved tooling and documentation**, making it easier to develop and maintain robotic applications. Finally yet significantly, ROS 2 is designed to be **supported for a longer period** than ROS 1, ensuring that your robotic applications will continue to be relevant and supported in the future.

1.2.1 Objectives of ROS 2

The primary objective of using ROS 2 is to provide a flexible, scalable, and efficient framework for building complex robotic systems. Here are some more specific goals:

1. Modularity and reusability: ROS 2 promotes a modular approach to robot development, allowing developers to create reusable components that can be easily integrated into different systems.
2. Real-time performance: ROS 2 is designed to handle real-time tasks efficiently, ensuring that robots can respond quickly and reliably to their environment.
3. Scalability: ROS 2 can be used to build systems of varying sizes, from small personal robots to large industrial automation systems.

In essence, ROS 2 aims to simplify the development of robotic applications by providing a standardized platform for communication, coordination, and control.

1.3 File System for ROS 2

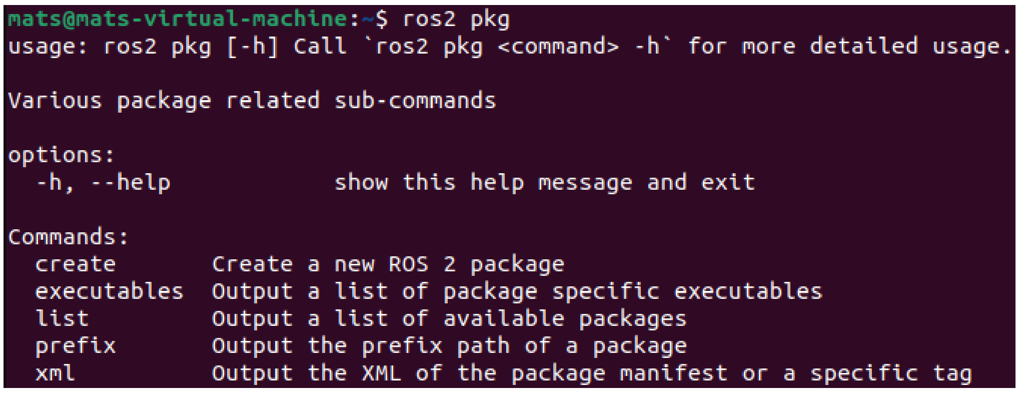
ROS 2 utilizes a hierarchical file system structure to organize and manage robot-related resources. This structure is designed to be intuitive and scalable, accommodating various robot configurations and applications. Here are some basic concepts.

- Root directory serves as the top-level workspace for your ROS 2 projects. It is typically named '`ws`' but can be customized.
- Source directory '`src`' contains source code for your custom ROS 2 packages, where each package is represented as a separate directory within `src`.
- Build directory '`build`' stores the build files generated during the compilation process.
- Install directory '`install`' contains the installed files of your built packages, and it is automatically created when you run the '`colcon build`' command. These files are typically executable and can be used by other applications.
- Log directory '`log`' stores log files generated during the execution of ROS 2 nodes for debugging and troubleshooting.

1.3.1 File System Tools

ROS 2 code is orgainized via '**Packages**'. To install or share the code, one must have them in a package. With packages, the ROS 2 work can be share and utilized with ease by all ROS 2 users. Different from ROS 1, package creation in ROS 2 uses '`ament`' as its build system, and '`colcon`' as its build tool. You can create a package using either CMake or Python, which are officially supported, though other build types also exist.

An important tool for file system management in ROS 2 is '`ros2 pkg`', which contains various package-related sub-commands. By calling '`ros2 pkg`' in the terminal, the detailed explanation about this tool is displayed as shown in Figure 1.2, including '`create`', '`executables`', '`list`', '`prefix`', and '`xml`'. To know more details for any sub-command, just call '`ros2 pkg <command> -h`'.



```
mats@mats-virtual-machine:~$ ros2 pkg
usage: ros2 pkg [-h] Call `ros2 pkg <command> -h` for more detailed usage.

Various package related sub-commands

options:
  -h, --help            show this help message and exit

Commands:
  create       Create a new ROS 2 package
  executables  Output a list of package specific executables
  list         Output a list of available packages
  prefix       Output the prefix path of a package
  xml          Output the XML of the package manifest or a specific tag
```

Figure 1.2: Sub-commands under `ros2 pkg`.

1.3.2 Package Tools

Some important tools for the ROS 2 package are introduced here. Initially, to create a ROS 2 package in the workspace by yourself, the command '`ros2 pkg create`' is used. To build packages in the workspace, just call the command '`colcon build`'.

Note that putting packages in a workspace is especially valuable because you can build many packages at once by running colcon build in the workspace root. Otherwise, you would have to build each package individually.

First, install the colcon extensions.

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

Here is an example that shows how to setup an empty workspace named '<WSName>' (select your own name, or 'ros2' by default).

Example: Initializing a ROS 2 workspace

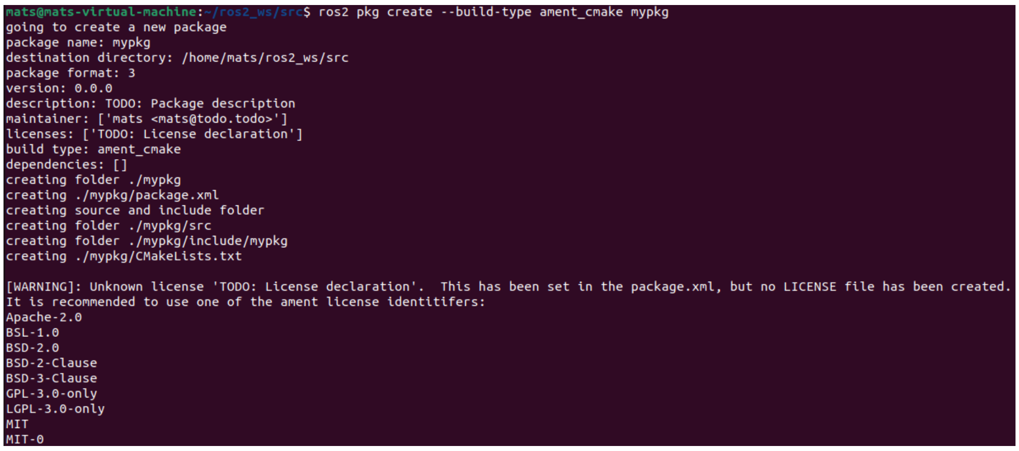
```
$ mkdir -p ~/<WSName>_ws/src
$ cd ~/<WSName>_ws
$ colcon build
```

In this empty workspace, go and create your first ROS 2 package named '<PKGName>' (again free to select on your own) with the build type as 'ament_cmake', for ROS 2 nodes in Cpp. The process is shown in Figure 1.3

Example: Creating your first ROS 2 package

```
$ cd ~/<WSName>_ws/src
$ ros2 pkg create --build-type ament_cmake <PKGName>
```

You can always go back to the root level of your workspace to build the packages, like



```
mats@mats-virtual-machine:~/ros2_ws/src$ ros2 pkg create --build-type ament_cmake mypkg
going to create a new package
package name: mypkg
destination directory: /home/mats/ros2_ws/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['mats <mats@todo.todo>']
licenses: ['TODO: License declaration']
build type: ament_cmake
dependencies: []
creating folder ./mypkg
creating ./mypkg/package.xml
creating source and include folder
creating folder ./mypkg/src
creating folder ./mypkg/include/mypkg
creating ./mypkg/CMakeLists.txt

[WARNING]: Unknown license 'TODO: License declaration'.  This has been set in the package.xml, but no LICENSE file has been created.
It is recommended to use one of the ament license identitifers:
Apache-2.0
BSL-1.0
BSD-2.0
BSD-2-Clause
BSD-3-Clause
GPL-3.0-only
LGPL-3.0-only
MIT
MIT-0
```

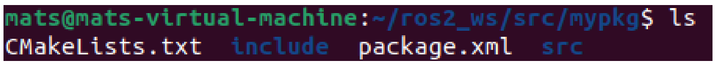
Figure 1.3: Create the ROS 2 package with type 'ament_cmake'

this newly created one. Alternatively, here shows a method to only build the specific package.

Example: Build a specific ROS 2 package

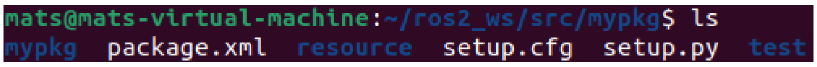
```
$ cd ~/<WSName>_ws
$ colcon build --packages-select <PKGName>
```

Inside the newly created package with type 'ament_cmake', the contents are shown in Figure 1.4. Another build-type 'ament_python' is available as well for creating the node using Python. With this build-type, the package contents would be as in Figure 1.5 below. In this class, we will demonstrate the coding with Cpp.



```
mats@mats-virtual-machine:~/ros2_ws/src/mypkg$ ls
CMakeLists.txt  include  package.xml  src
```

Figure 1.4: A new package with build-type 'ament_cmake'.



```
mats@mats-virtual-machine:~/ros2_ws/src/mypkg$ ls
mypkg  package.xml  resource  setup.cfg  setup.py  test
```

Figure 1.5: A new package with build-type 'ament_python'.

The 'src' directory (or the directory with the same name as the package name for python) is where all your custom Python nodes will go.

1.4 Important Concepts

This section explains some of the most frequently used ROS 2 terms. Use this section as a ROS 2 glossary as many terms might be new to you. When there are unfamiliar terms, you are suggested look over the definition and then move on. You will become more familiar with the concepts as you engage with examples later.

1.4.1 Overall Concepts

Here are a series of core ROS 2 concepts that make up what is referred to as the 'ROS 2 graph', a network of ROS 2 elements that process data together.

Package

A package is the basic unit for ROS 2, based on which the application is developed. A package contains either nodes or a configuration file to launch other packages. The package also contains all the files necessary for running the package, including ROS dependency libraries, datasets, and configuration files.

Node

Each node in ROS should be responsible for a single purpose, and each node can send and receive data from other nodes via topics, services, actions, or parameters. A robotic system contains many nodes working together.

Topics

ROS 2 breaks complex systems down into nodes. Topics are vital in that they are one of the main way to connect nodes for exchanging messages. Topics offer great flexibility. A node may publish data to any number of topics and simultaneously have subscriptions to any number of topics. Likewise, a topic can hear from any number of publisher(s) and deliver message to any number of subscriber(s) as shown in Figure 1.6 below.

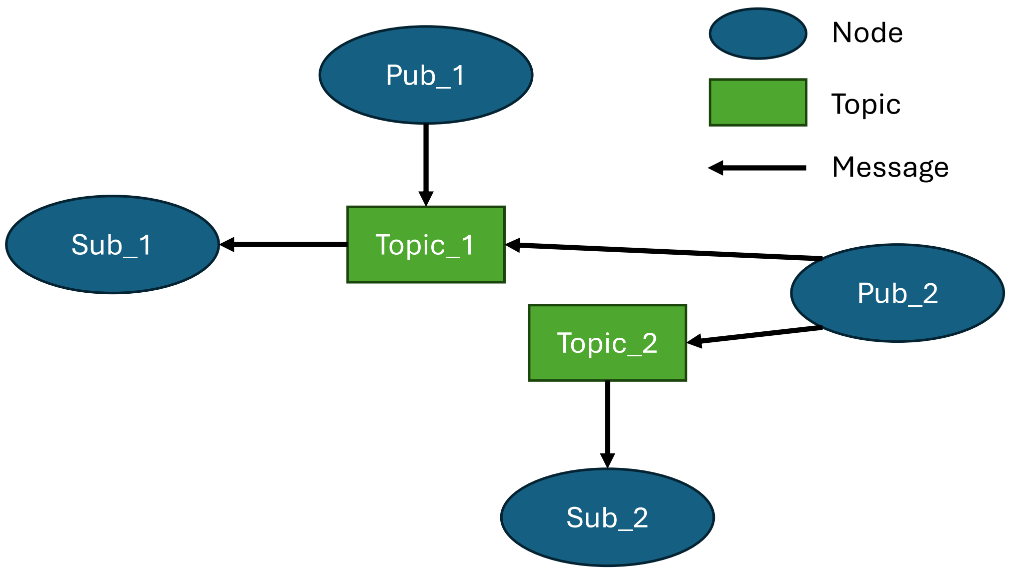


Figure 1.6: Diagram of ROS Topic connecting ROS Nodes via messages.

1.5 Example: Coding a Publisher and Subscriber

Here, let us try an simple example from the ROS 2 official tutorial [2]. Double check your Ubuntu and ROS 2 has been set up with the correct version.

OS version: **Ubuntu 22.04.5 LTS (Jammy Jellyfish)**,
ROS 2 version: **Humble Hawksbill**.

Wrong version may lead to errors in this session. On how to install and setup the OS and ROS 2, please refer to the Appendix at the end of this file.

1.5.1 Publisher Setup

Initially, create a package for this example with the build-type 'ament_cmake', as Cpp will be used for this example.

```
$ ros2 pkg create --build-type ament_cmake --license Apache-2. 0
    cpp_pubsub
```

Inside the 'src' folder of the newly created package (NOT the workspace), call the following commands to copy the official ROS 2 publisher file:

```
$ wget -O publisher_member_function.cpp https://raw.
    githubusercontent.com/ros2/examples/humble/rclcpp/topics/
    minimal_publisher/member_function.cpp
```

This program is briefly explained here.

The beginning part is the Cpp header. Note that '`rclcpp/rclcpp.hpp`' is for the common use of the ROS 2 system.

```
#include "rclcpp/rclcpp.hpp"
```

The node class '`MinimalPublisher`' by inheriting from `rclcpp::Node`.

```
class MinimalPublisher : public rclcpp::Node
```

Inside this class, there are the public constructor and the private function. The public constructor names the node ‘`minimal_publisher`’ and initializes ‘`count_`’ to 0. Inside the constructor, the publisher is initialized, and ‘`timer_`’ is initialized such that ‘`timer_callback`’ function to be executed twice per second.

```
public:
  MinimalPublisher()
  : Node("minimal_publisher"), count_(0)
  {
    publisher_ = this -> create_publisher<std_msgs::msg::String>("
        topic", 10);
    timer_ = this -> create_wall_timer(
    500ms, std::bind(&MinimalPublisher::timer_callback, this));
  }
```

Next, in the private function ‘`timer_callback`’, the messages are published. Here, the ‘`RCLCPP_INFO`’ will print the published message to the console.

```
private:
  void timer_callback()
  {
    auto message = std_msgs::msg::String();
    message.data = "Hello, world! " + std::to_string(count_++);
    RCLCPP_INFO(this -> get_logger(), "Publishing: '%s'", message.
        data.c_str());
    publisher_ -> publish(message);
  }
```

At the end of the program is the ‘`main`’ function that runs for ROS 2 initialization, looping, and ending.

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

After comprehending this program, we go back to edit the dependencies. Go back one level via `cd ..` and we can see the ‘CMakeLists.txt’ and ‘package.xml’. Initially, in the ‘package.xml’ (use editor such as `gedit package.xml`), add the two dependencies '`rclcpp`' and '`std_msgs`' after line 10, and now the line 10 to line 15 in the file should be like this.

Amendment of 'package.xml' from line 10 to line 15

```
10 <buildtool_depend>ament_cmake</buildtool_depend>
11 <depend>rclcpp</depend>
12 <depend>std_msgs</depend>
13
14 <test_depend>ament_lint_auto</test_depend>
15 <test_depend>ament_lint_common</test_depend>
```

Next, open 'CMakeLists.txt' and add the following contents after 'find_package' in line 9, which results in the following:

Amendment of 'CMakeLists.txt' from line 8 to line 16

```
 8 # find dependencies
 9 find_package(ament_cmake REQUIRED)
10 find_package(rclcpp REQUIRED)
11 find_package(std_msgs REQUIRED)
12 add_executable(talker src/publisher_member_function.cpp)
13 ament_target_dependencies(talker rclcpp std_msgs)
14 install(TARGETS
15   talker
16   DESTINATION lib/${PROJECT_NAME})
```

1.5.2 Subscriber Setup

After preparing the publisher, now let us set up the subscriber. Similarly, in the package src folder, obtain the prepared subscriber node.

```
wget -O subscriber_member_function.cpp https://raw.
    githubusercontent.com/ros2/examples/humble/rclcpp/topics/
    minimal_subscriber/member_function.cpp
```

Check this subscriber node, and you can see the structure is roughly the same as the publisher node. However, a significant difference between the publisher and the subscriber is that there is no timer now. This is because there is no need to hear from the publisher following the same frequency. Instead, the subscriber just need to be ready to receive the message, which is realized in the main function by spinning the '`MinimalSubscriber`' node.
After adding the subscriber node, again add the subscribe as the executable in the

‘CMakeLists.txt’. The final ‘CMakeLists.txt’ should look like this.

Final version of ‘CMakeLists.txt’ from line 8 to line 23

```
 8 # find dependencies
 9 find_package(ament_cmake REQUIRED)
10 find_package(rclcpp REQUIRED)
11 find_package(std_msgs REQUIRED)
12 add_executable(talker src/publisher_member_function.cpp)
13 ament_target_dependencies(talker rclcpp std_msgs)
14 install(TARGETS
15   talker
16   DESTINATION lib/${PROJECT_NAME})
17
18 add_executable(listener src/subscriber_member_function.cpp)
19 ament_target_dependencies(listener rclcpp std_msgs)
20 install(TARGETS
21   talker
22   listener
23   DESTINATION lib/${PROJECT_NAME})
```

1.5.3 Build and Run

To build and run the package, check the dependencies first:

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

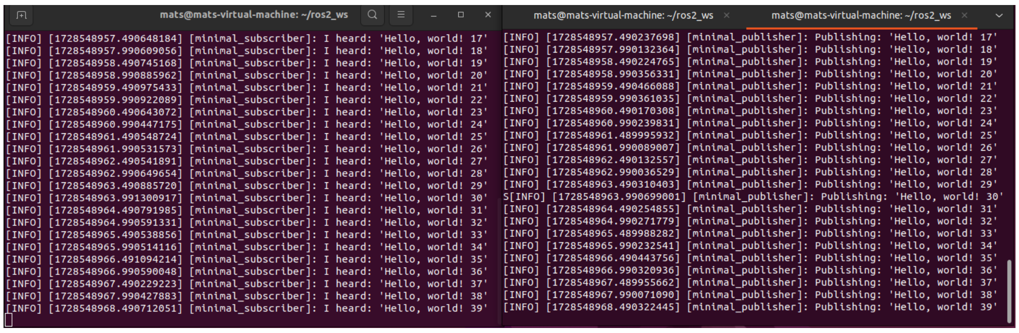
If all dependencies are installed, in the root of your workspace, build the package.

```
colcon build --packages-select cpp_pubsub
```

Open two new terminals and navigate both to ‘ros2_ws’. For both terminals, source the setup files.

```
. install/setup.bash
```

Respectively, in one terminal, run the publisher.


mats@mats-virtual-machine: ~/ros2_ws
mats@mats-virtual-machine: ~/ros2_ws
mats@mats-virtual-machine: ~/ros2_ws

```
[INFO] [1728548957.490648184] [minimal_subscriber]: I heard: 'Hello, world! 17'
[INFO] [1728548957.990609056] [minimal_subscriber]: I heard: 'Hello, world! 18'
[INFO] [1728548958.490745168] [minimal_subscriber]: I heard: 'Hello, world! 19'
[INFO] [1728548958.990885962] [minimal_subscriber]: I heard: 'Hello, world! 20'
[INFO] [1728548959.490975433] [minimal_subscriber]: I heard: 'Hello, world! 21'
[INFO] [1728548959.990922089] [minimal_subscriber]: I heard: 'Hello, world! 22'
[INFO] [1728548960.490643072] [minimal_subscriber]: I heard: 'Hello, world! 23'
[INFO] [1728548960.990447175] [minimal_subscriber]: I heard: 'Hello, world! 24'
[INFO] [1728548961.490548724] [minimal_subscriber]: I heard: 'Hello, world! 25'
[INFO] [1728548961.990531573] [minimal_subscriber]: I heard: 'Hello, world! 26'
[INFO] [1728548962.490541891] [minimal_subscriber]: I heard: 'Hello, world! 27'
[INFO] [1728548962.990649654] [minimal_subscriber]: I heard: 'Hello, world! 28'
[INFO] [1728548963.490885720] [minimal_subscriber]: I heard: 'Hello, world! 29'
[INFO] [1728548963.991300917] [minimal_subscriber]: I heard: 'Hello, world! 30'
[INFO] [1728548964.490791985] [minimal_subscriber]: I heard: 'Hello, world! 31'
[INFO] [1728548964.990591331] [minimal_subscriber]: I heard: 'Hello, world! 32'
[INFO] [1728548965.490538856] [minimal_subscriber]: I heard: 'Hello, world! 33'
[INFO] [1728548965.990514116] [minimal_subscriber]: I heard: 'Hello, world! 34'
[INFO] [1728548966.491094214] [minimal_subscriber]: I heard: 'Hello, world! 35'
[INFO] [1728548966.990590048] [minimal_subscriber]: I heard: 'Hello, world! 36'
[INFO] [1728548967.490229223] [minimal_subscriber]: I heard: 'Hello, world! 37'
[INFO] [1728548967.990427883] [minimal_subscriber]: I heard: 'Hello, world! 38'
[INFO] [1728548968.490712051] [minimal_subscriber]: I heard: 'Hello, world! 39'
```

```
[INFO] [1728548957.490237698] [minimal_publisher]: Publishing: 'Hello, world! 17'
[INFO] [1728548957.990132364] [minimal_publisher]: Publishing: 'Hello, world! 18'
[INFO] [1728548958.490224765] [minimal_publisher]: Publishing: 'Hello, world! 19'
[INFO] [1728548958.990356331] [minimal_publisher]: Publishing: 'Hello, world! 20'
[INFO] [1728548959.490466088] [minimal_publisher]: Publishing: 'Hello, world! 21'
[INFO] [1728548959.990361035] [minimal_publisher]: Publishing: 'Hello, world! 22'
[INFO] [1728548960.490170308] [minimal_publisher]: Publishing: 'Hello, world! 23'
[INFO] [1728548960.990239831] [minimal_publisher]: Publishing: 'Hello, world! 24'
[INFO] [1728548961.489995932] [minimal_publisher]: Publishing: 'Hello, world! 25'
[INFO] [1728548961.990089007] [minimal_publisher]: Publishing: 'Hello, world! 26'
[INFO] [1728548962.490132557] [minimal_publisher]: Publishing: 'Hello, world! 27'
[INFO] [1728548962.990036529] [minimal_publisher]: Publishing: 'Hello, world! 28'
[INFO] [1728548963.490310403] [minimal_publisher]: Publishing: 'Hello, world! 29'
S[INFO] [1728548963.990699001] [minimal_publisher]: Publishing: 'Hello, world! 30'
[INFO] [1728548964.490254855] [minimal_publisher]: Publishing: 'Hello, world! 31'
[INFO] [1728548964.990271779] [minimal_publisher]: Publishing: 'Hello, world! 32'
[INFO] [1728548965.489988282] [minimal_publisher]: Publishing: 'Hello, world! 33'
[INFO] [1728548965.990232541] [minimal_publisher]: Publishing: 'Hello, world! 34'
[INFO] [1728548966.490443756] [minimal_publisher]: Publishing: 'Hello, world! 35'
[INFO] [1728548966.990320936] [minimal_publisher]: Publishing: 'Hello, world! 36'
[INFO] [1728548967.489955662] [minimal_publisher]: Publishing: 'Hello, world! 37'
[INFO] [1728548967.990071090] [minimal_publisher]: Publishing: 'Hello, world! 38'
[INFO] [1728548968.490322445] [minimal_publisher]: Publishing: 'Hello, world! 39'
```

Figure 1.7: ROS 2 Publisher and Subscriber.

```
ros2 run cpp_pubsub talker
```

In the other, run the subscriber.

```
ros2 run cpp_pubsub listener
```

The listener will start printing messages to the console, starting at whatever message count the publisher is on at that time as shown in Figure 1.7. This indicates the successful communication between the two ROS nodes over the ROS topic.

Chapter 2

Simulation Experiment

2.1 Playing with Turtlesim

2.1.1 What is Turtlesim

Turtlesim is a simple simulator tool that comes with ROS (ROS 2) and is often used as an educational tool for learning the basics of ROS (ROS 2) concepts such as topics, services, and nodes. It provides a virtual 2D turtle that can be controlled to move around on the screen using ROS (ROS 2) commands. Turtlesim is ideal for beginners who are learning how to interact with robots using ROS (ROS 2) because it simulates fundamental robotic operations without the complexity of real-world hardware.

Below show some key concepts of Turtlesim:

- **Nodes**: In Turtlesim, the turtle is a node, and it interacts with other nodes via ROS communication. Nodes are executable files that perform computation.
- **Topics**: ROS topics allow nodes to communicate with each other by sending messages. For example, to move the turtle, you send velocity commands via the `/turtle1/cmd_vel` topic.
- **Services**: ROS services enable nodes to interact using request-response communication. Turtlesim includes services like `/spawn` to create new turtles or `/reset` to reset the simulation.
- **Messages**: The data exchanged between nodes are sent in the form of messages. For Turtlesim, messages like `Twist` are used to control the turtle's movement.
- **Parameters**: Turtlesim uses ROS parameters to configure and store runtime settings. For instance, the background color of the window can be set using parameters.

2.1.2 Install Turtlesim

As always, don't forget to source ROS 2 firstly in every new terminal you open (see step 2 in Appendix B).

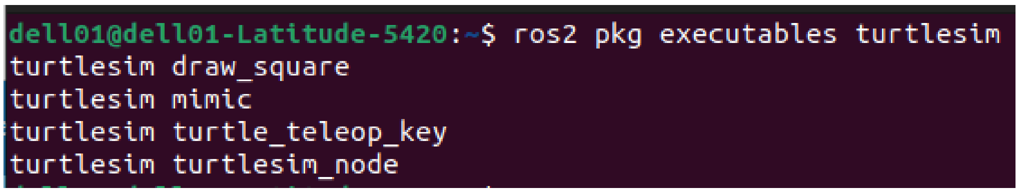
Install the turtlesim package:

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

Check that the package is installed:

```
ros2 pkg executables turtlesim
```

The above command should return a list of turtlesim's executables:



```
dell01@dell01-Latitude-5420:~$ ros2 pkg executables turtlesim
turtlesim draw_square
turtlesim mimic
turtlesim turtle_teleop_key
turtlesim turtlesim_node
```

Figure 2.1: ROS 2 Successful Installation.

2.1.3 Use Turtlesim

To start Turtlesim, enter the following command in your terminal:

```
ros2 run turtlesim turtlesim_node
```

The simulator window should appear as shown in Figure 2.2. You will also see messages from the node in the terminal.

Open a new terminal and run a new node to control the turtle in the first node:

```
ros2 run turtlesim turtle_teleop_key
```

Use the arrow keys on your keyboard to control the movement of the turtle.

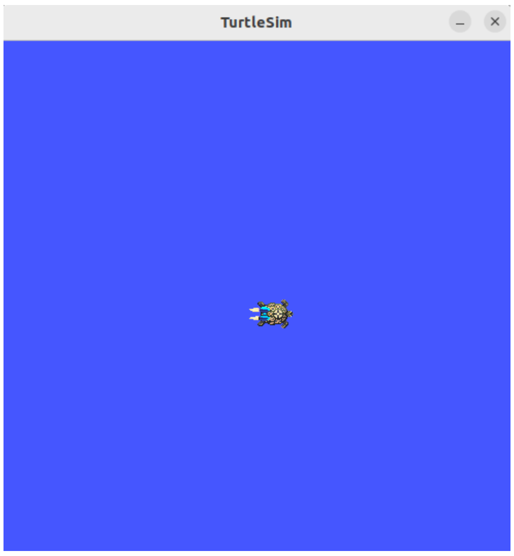


Figure 2.2: Turtlesim Simulator Window.

2.2 Introducing rqt

2.2.1 What is rqt

rqt is a graphical user interface (GUI) framework used in ROS (ROS 2) that provides a variety of tools and plugins to visualize and manage ROS-based systems. It's a highly modular tool built using Python and Qt, and it helps ROS users interact with running ROS nodes, topics, services, and parameters in a visual way.

Everything done in rqt can be done on the command line, but rqt provides a more user-friendly way to manipulate ROS 2 elements. You can also develop your own custom plugins using Python or C++.

2.2.2 Install rqt

Open a new terminal to install rqt and its plugins:

```
sudo apt update
sudo apt install '~nros-humble-rqt*'
```

To run rqt:

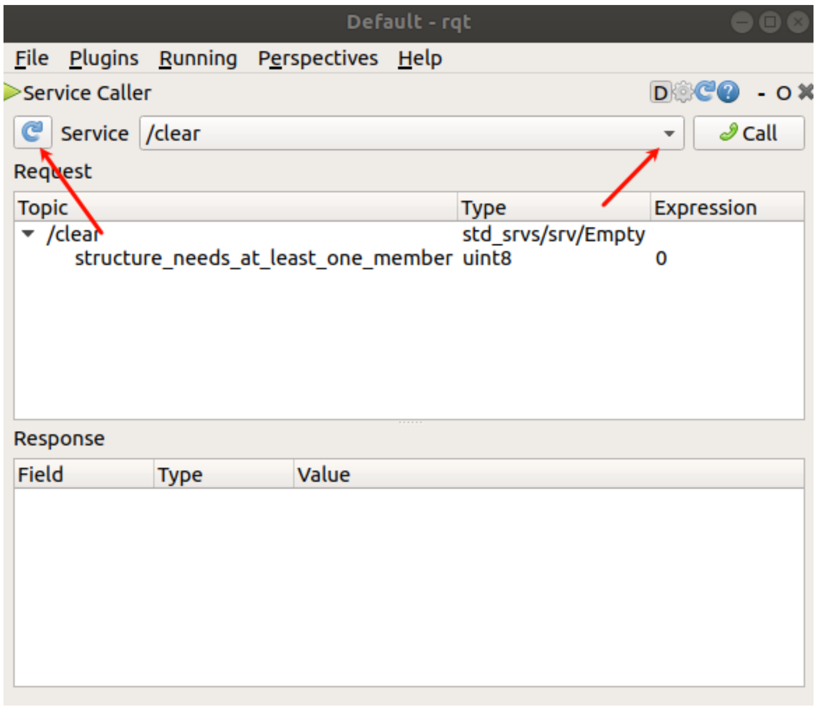
```
rqt
```

2.2.3 Use rqt

Select **Plugins** > **Services** > **Service Caller** from the menu bar at the top.

Use the refresh button to the left of the **Service** dropdown list to ensure all the services of your turtlesim node are available.

Click on the **Service** dropdown list to see turtlesim's services.



Topic	Type	Expression
/clear	std_srvs/srv/Empty	
structure_needs_at_least_one_member	uint8	0

Field	Type	Value

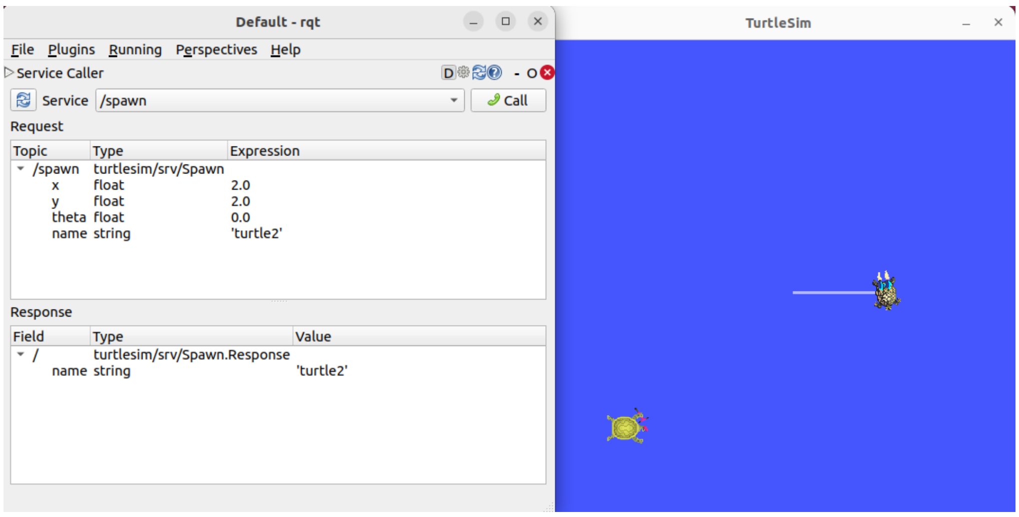
Figure 2.3: Default rqt.

Try the spawn Service

Select the `/spawn` service in the **Service** dropdown list to create another turtle in the turtlesim window.

Give the new turtle a unique name and enter some valid coordinates (e.g., 0~10) to place it.

Click the **Call** button on the upper right side of the rqt window to spawn turtle2. If the service call was successful, you should see a new turtle in the simulator window as shown in Figure 2.4.



Request

```
Topic     Type               Expression
/spawn    turtlesim/srv/Spawn
  x       float              2.0
  y       float              2.0
  theta   float              0.0
  name    string             'turtle2'
```

Response

```
Field  Type                          Value
/      turtlesim/srv/Spawn.Response
  name string                        'turtle2'
```

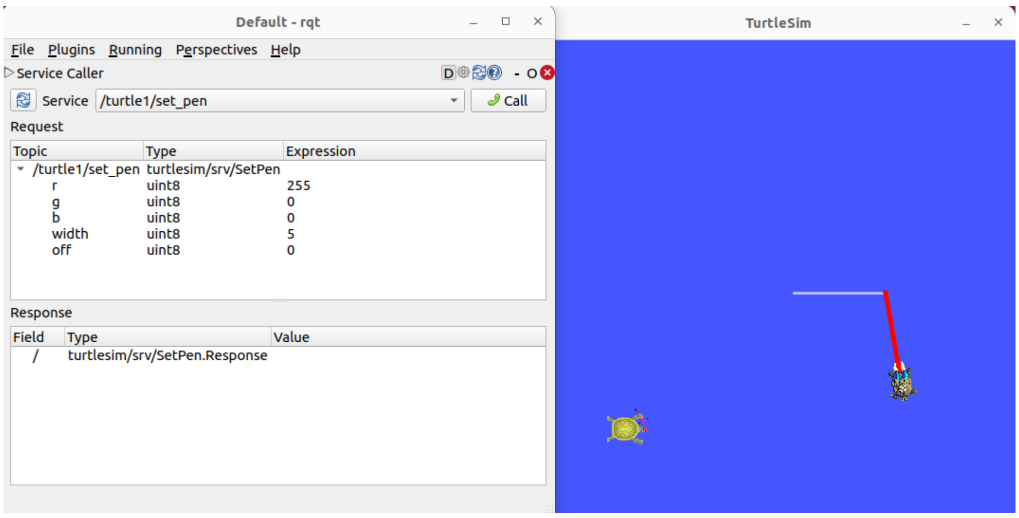
Figure 2.4: Spawn A New Turtle.

Try the set_pen service

Select the `/set_pen` service in the **Service** dropdown list to give turtle1 a unique pen.

Set color of the pen using **r**, **g** and **b** (0~255), and set the thickness of the line using **width**.

Click the **Call** button, return to the terminal where `turtle_teleop_key` is running, and press the arrow keys. If the service call was successful, you should see `turtle1`'s pen has changed as shown in Figure 2.5.



Request

```
Topic              Type                  Expression
/turtle1/set_pen   turtlesim/srv/SetPen
  r                uint8                 255
  g                uint8                 0
  b                uint8                 0
  width            uint8                 5
  off              uint8                 0
```

Response

```
Field  Type                           Value
/      turtlesim/srv/SetPen.Response
```

Figure 2.5: Set A Unique Pen for turtle1.

Remapping

To control turtle2, you need another `turtle_teleop_key` node. However, if you try to run the same command as before, you will notice that this one also controls turtle1.

Therefore, you need to remap the `cmd_vel` topic to change this behavior. Open a new terminal and run:

```
ros2 run turtlesim turtle_teleop_key --ros-args --remap turtle1/
    cmd_vel:=turtle2/cmd_vel
```

Now, you can move turtle2 in this new terminal.

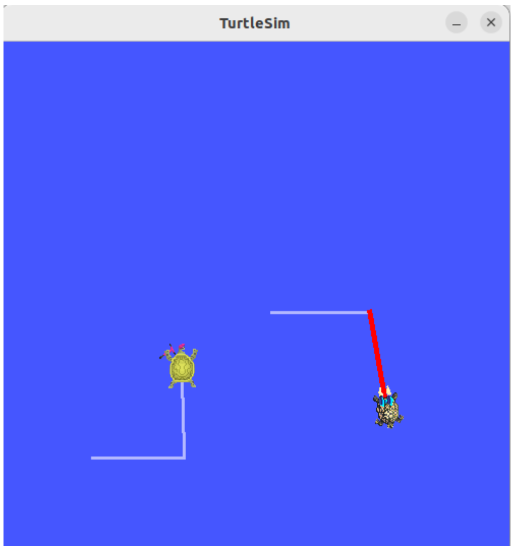


Figure 2.6: Remapping.

Close Turtlesim

To stop the simulation, you can enter `Ctrl + C` in the `turtlesim_node` terminal, and type `q` in the `turtle_teleop_key` terminals.

2.3 Additional Task

2.3.1 Task Introduction

Now, you have grasped a basic knowledge of the ROS 2 Topics, Turtlesim and rqt, let's try one advanced task using what you've just learnt.

Overview: Cooperative motion of two turtles in Turtlesim.

- **Leader Turtle:** controlled using the keyboard (via ROS 2 `urtle_teleop_key`).
- **Follower Turtle:** continuously tracks and follows the leader turtle by adjusting its velocity and direction.

This will involve publishing and subscribing to topics to control the turtles' positions.

2.3.2 Task Implementation

Step 1: Create the workspace and package

First, source your ROS 2 installation as usual. Let's use the workspace you created in the previous tutorial, `<WSName>_ws`, for a new package `turtle_coop`. Make sure you are in the `src` folder before running the package creation command.

```
cd ~/<WSName>_ws/src
ros2 pkg create --build-type ament_cmake --license Apache-2.0
    turtle_coop
```

Step 2: Add the turtle_coop.cpp, CMakeLists.txt, and package.xml into your package's directory.

Please refer to these files in appendix C and place them in the correct directories in the package.

Fill in the `TODO for student` blanks in the codes according to section 1.5.

Step 3: Build the package

Return to the root of your workspace and build only the `turtle_coop` package.

```
cd ~/<WSName>_ws
colcon build --packages-select turtle_coop
```

If you encounter errors while building your package, please remember to clean your workspace before rebuilding. Sometimes, residual build artifacts can cause issues.

```
cd ~/<WSName>_ws
rm -rf build/ install/ log/
colcon build --packages-select turtle_coop
```

Step 4: Source the workspace

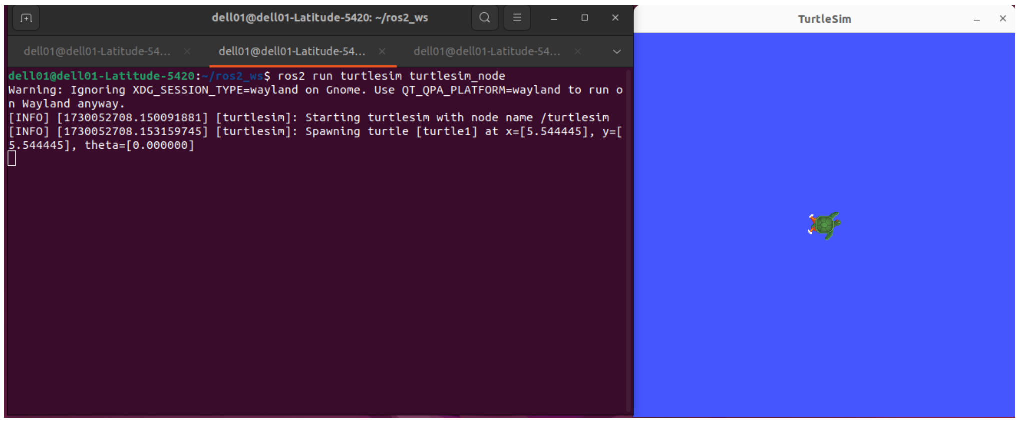
Open a new terminal, and from inside the `<WSName>_ws` directory, run the following command to source your workspace:

```
source install/local_setup.bash
```

Remember to do this firstly EVERY TIME you open a new terminal in the following steps! Otherwise ROS cannot locate your packages.

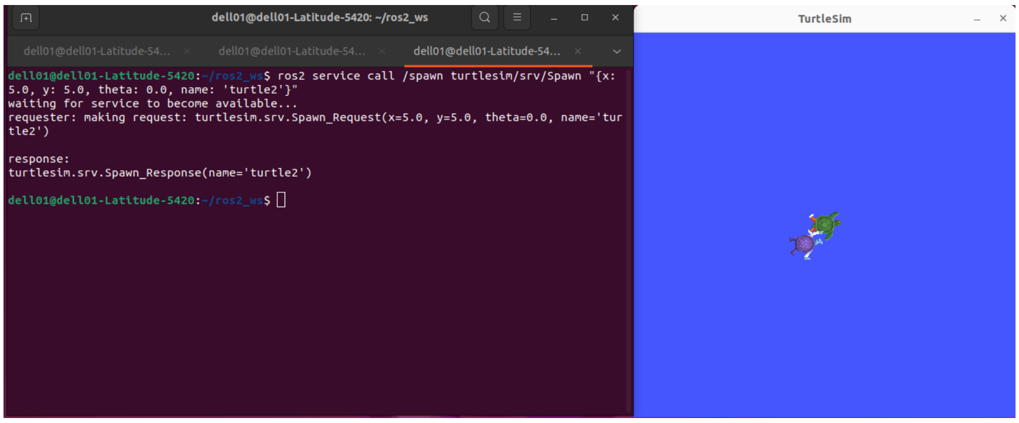
Step 5: Launch Turtlesim and the cooperative node

Terminal 1: Launch `turtlesim_node`.


dell01@dell01-Latitude-5420: ~/ros2_ws
dell01@dell01-Latitude-54...
dell01@dell01-Latitude-54...
dell01@dell01-Latitude-54...
TurtleSim

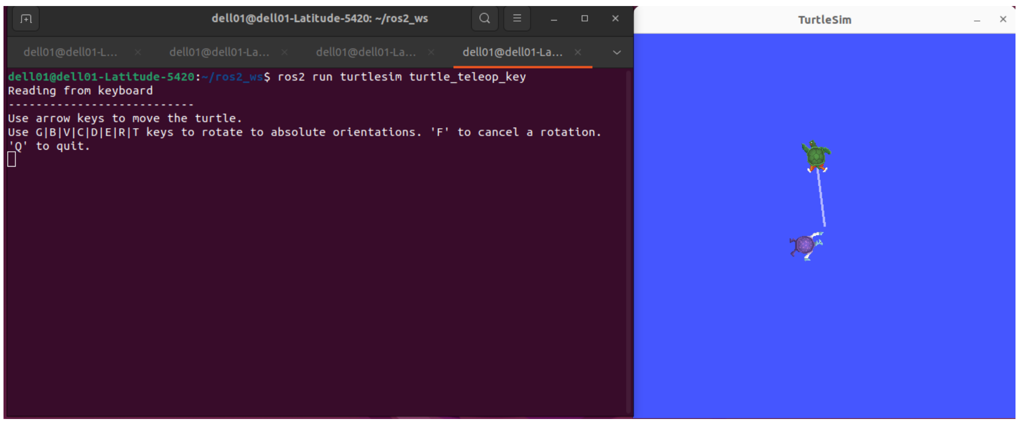
```
dell01@dell01-Latitude-5420:~/ros2_ws$ ros2 run turtlesim turtlesim_node
Warning: Ignoring XDG_SESSION_TYPE=wayland on Gnome. Use QT_QPA_PLATFORM=wayland to run o
n Wayland anyway.
[INFO] [1730052708.150091881] [turtlesim]: Starting turtlesim with node name /turtlesim
[INFO] [1730052708.153159745] [turtlesim]: Spawning turtle [turtle1] at x=[5.544445], y=[
5.544445], theta=[0.000000]
```

Terminal 2: Spawn turtle2.


dell01@dell01-Latitude-5420: ~/ros2_ws
dell01@dell01-Latitude-54...
dell01@dell01-Latitude-54...
dell01@dell01-Latitude-54...
TurtleSim

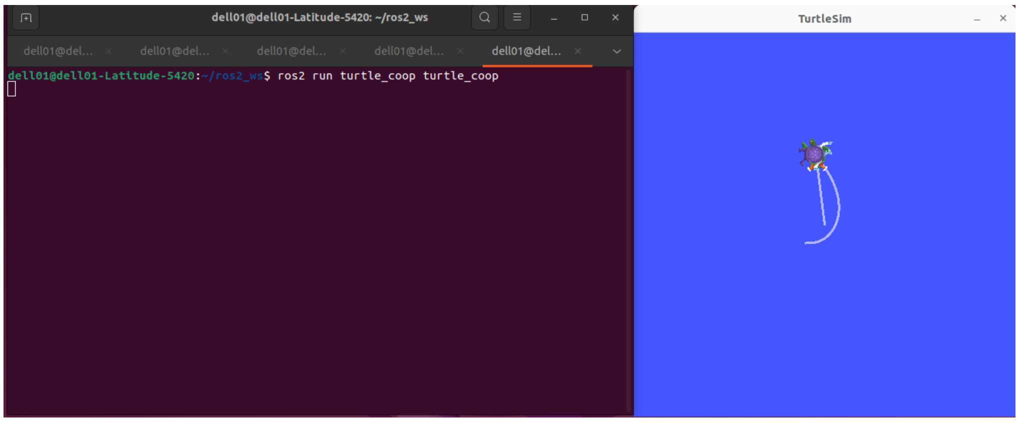
```
dell01@dell01-Latitude-5420:~/ros2_ws$ ros2 service call /spawn turtlesim/srv/Spawn "{x:
5.0, y: 5.0, theta: 0.0, name: 'turtle2'}"
waiting for service to become available...
requester: making request: turtlesim.srv.Spawn_Request(x=5.0, y=5.0, theta=0.0, name='tur
tle2')

response:
turtlesim.srv.Spawn_Response(name='turtle2')

dell01@dell01-Latitude-5420:~/ros2_ws$ 
```

Terminal 3: Control turtle1 using `turtle_teleop_key`.



```
dell01@dell01-Latitude-5420:~/ros2_ws$ ros2 run turtlesim turtle_teleop_key
Reading from keyboard
---------------------------
Use arrow keys to move the turtle.
Use G|B|V|C|D|E|R|T keys to rotate to absolute orientations. 'F' to cancel a rotation.
'Q' to quit.
```

Terminal 4: Run the `turtle_coop` node to make turtle2 follow turtle1.



```
dell01@dell01-Latitude-5420:~/ros2_ws$ ros2 run turtle_coop turtle_coop
```

By following these steps, you should be able to control turtle1 via the keyboard while turtle2 follows based on the logic defined in `turtle_coop` node.

The rqt graph is shown as below.

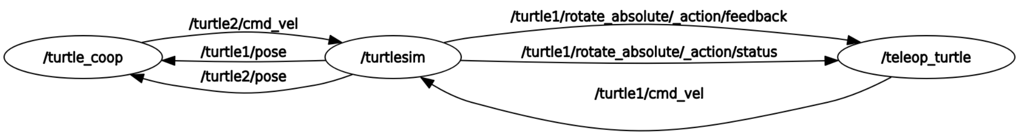


Figure 2.7: rqt graph for Turtle Cooperation

2.3.3 Improvements

You may noticed that the given controller performance is not satisfactory sometimes. Please consider for improvements if you have time. Some possible directions are given here:

1. Sometimes, the rotation of the follower turtle bot is not along the shortest rotation direction but the opposite (almost 360^o rotation). Check the code and consider

from the trajectory planning perspective what it is currently lacking and how to improve the code to solve this problem.

2. Deviations occur when two turtles are on the same horizontal line or overlapped. Consider how to avoid this problem, for example, by introducing a tolerance, adding upper bound, adding dead zone, etc.

3. The tracking of the follower to the leader is asymptotic due to the proportional control law. Consider from the control perspective on improvements to make the convergence within a finite time.

2.4 Playing with RX-150

We install the RX-150 robotic arm system according to the official tutorial.

```
sudo apt install curl
curl 'https://raw.githubusercontent.com/Interbotix/
    interbotix_ros_manipulators/main/interbotix_ros_xsarms/install
    /amd64/xsarm_amd64_install.sh' > ~/xsarm_amd64_install.sh
chmod +x ~/xsarm_amd64_install.sh
~/xsarm_amd64_install.sh -d humble
```

For the questions that appear during the installation script:

- There is no need to install the perception package (just type n and press Enter)
- There is no need to install the MATLAB-ROS API (just type n and press Enter)

After the installation is complete, we need to configure the environment variables required to run the robotic arm program. (Similarly, the following commands only need to be executed once.)

```
echo "source ~/interbotix_ws/install/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

To check whether the installation was successful, we can look at the list of Interbotix ROS packages.

```
interbotix_common_modules
interbotix_common_sim
interbotix_common_toolbox
interbotix_ros_xsarms
interbotix_ros_xsarms_examples
interbotix_ros_xseries
interbotix_tf_tools
interbotix_xs_driver
interbotix_xs_modules
interbotix_xs_msgs
interbotix_xs_ros_control
interbotix_xs_rviz
interbotix_xs_sdk
interbotix_xs_toolbox
interbotix_xsarm_control
interbotix_xsarm_descriptions
interbotix_xsarm_dual
interbotix_xsarm_joy
interbotix_xsarm_moveit
interbotix_xsarm_moveit_interface
interbotix_xsarm_ros_control
interbotix_xsarm_sim
```

2.4.1 Running the Robot Arm in Gazebo

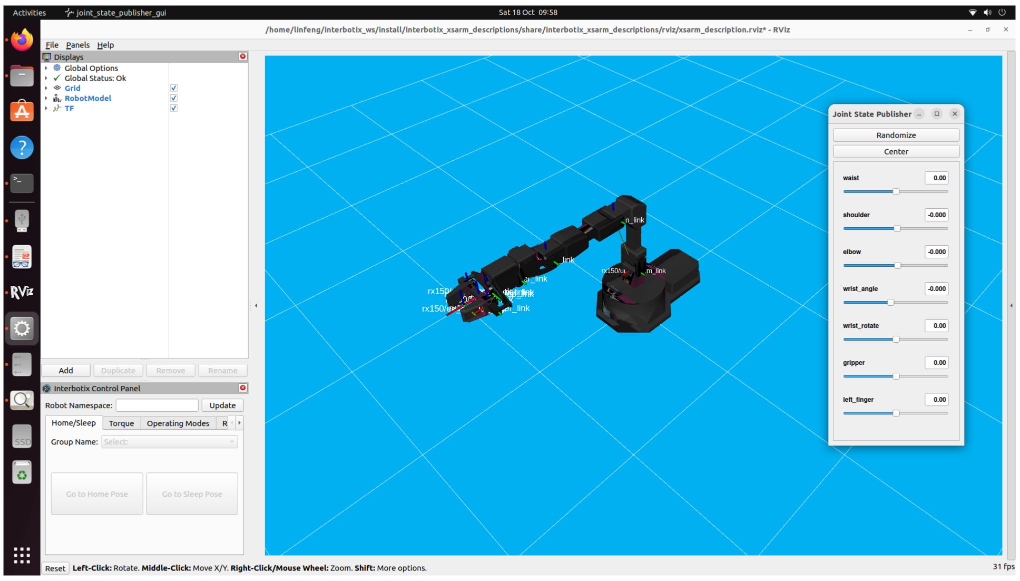
Use the following command to launch the ReactorX 150 robot arm in Gazebo:

```
ros2 launch interbotix_xsarm_descriptions xsarm_description.launch
    .py robot_model:=rx150 use_joint_pub_gui:=true
```

After launching, the user can use the graphical interface of the joint state publisher to manually control the joint angles of the robot arm for testing. In the graphical interface, you can move the sliders to change the joint positions in real time.

- `Randomize`: Randomly set all joint positions within the defined valid range.
- `Center`: Reset all joints to their default or central positions, usually zero.

This graphical interface controller is very helpful when debugging, especially when you want to test the positive direction of the joint angles. Its working principle is that when the user adjusts the sliders, the updated joint angles are published to `/joint_states`.



waist 0.00
shoulder -0.000
elbow -0.000
wrist_angle -0.000
wrist_rotate 0.00
gripper 0.00
left_finger 0.00

Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel: Zoom. Shift: More options.

Figure 2.8: Graphical interface of the joint state publisher

References

[1] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, W. Woodall, “Robot Operating System 2: Design, architecture, and uses in the wild,” Science Robotics vol. 7, May 2022.

[2] Writing a simple publisher and subscriber (C++) (Humble). Available at: https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Cpp-Publisher-And-Subscriber.html

Appendix A

Instructions for Setting up Ubuntu

Please note that the Linux version used in this EE5111/EE5061 lab session is Ubuntu 22.04.5 LTS (Jammy Jellyfish). Please refer to `https://releases.ubuntu.com/22.04/`. Below is a detailed guide for installing Ubuntu on a Windows PC or a Mac using three methods: dual boot, virtual machine (VM), and WSL2 (for Windows).

A.1 Option 1: Dual Boot

To install the dual boot system alongside Windows 10/11, please refer to `https://itsfoss.com/install-ubuntu-1404-dual-boot-mode-windows-8-81-uefi/` and `https://help.ubuntu.com/community/WindowsDualBoot`.

To install the dual boot system alongside Mac OS X, please refer to section 4 of `https://help.ubuntu.com/community/MactelSupportTeam/AppleIntelInstallation#Dual-Boot:.2520Mac.2520OSX.2520and.2520Ubuntu` and `https://help.ubuntu.com/community/DualBoot/MacOSX`.

A.2 Option 2: Virtual Machine

In computing, a virtual machine (VM) is an emulation of a computer system. The VMWare by Broadcom has become free in 2024 for personal use. To install VMware, please visit `https://knowledge.broadcom.com/external/article?articleNumber=368667`, and go for '`VMware Workstation Pro` ' for Windows or '`VMware Fusion Pro`' for MacOS. For the tutorial, please refer to
`https://knowledge.broadcom.com/external/article?articleNumber=344595`.

The VirtualBox is also recommended, which is a free and powerful product for enterprise as well as home use. To install VirtualBox, please refer to `https://www.virtualbox.org/`.

After you have installed VM, please run and create a VM (work with default settings but customize for faster response), and then install Ubuntu 22.04 on it referring to `https://ubuntu.com/tutorials/install-ubuntu-desktop` and `https://releases.ubuntu.com/22.04/`.

A.3 Option 3: Windows Subsystem for Linux (WSL2)

Please note that this option is for Windows 10/11 users. Some manuals are available at `https://ubuntu.com/wsl` and `https://learn.microsoft.com/en-us/windows/wsl/install`. The installation steps are also briefly given below.

- Step 1: Enable WSL on Windows 10/11.
  - Run Windows PowerShell as administrator.
  - Type the following command in PowerShell.

    ```
    $ wsl --install
    ```

- Step 2: Install Ubuntu 22.04 for WSL.

  ```
  $ wsl.exe --install --d Ubuntu-22.04
  ```

- Step 3: Open WSL.

  ```
  $ wsl
  ```

- Step 4: Set up Ubuntu (set a username and password).
- Step 5: Set up Ubuntu GUI for WSL.
  - Install xfce desktop on WSL by typing the following commands in the Ubuntu 22.04 terminal.

    ```
    $ sudo apt update
    $ sudo apt install xfce4 xfce4-terminal
    ```

Now, the Ubuntu 22.04 on Windows 10/11 is ready to use via WSL.

Appendix B

Instructions for Setting up ROS 2

After you installed the Ubuntu 22.04 on your PC, please follow the below guidelines to install ROS 2.

- Step 1: Install ROS 2 Humble Hawksbill (please refer to `https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debs.html`).
- Step 2: Configure environment (please refer to `https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Configuring-ROS2-Environment.html`). As mentioned in task 2, you can add the command to your shell startup if you don't want to source ROS 2 every time you open a new shell:

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

Now, everything is ready and please work on the lab session following the lab manual' on your PC.

Appendix C

Turtle Cooperation Files

File 1: turtle_coop.cpp (to fill in the blanks marked with ??)

```cpp
#include "rclcpp/rclcpp.hpp"
#include "geometry_msgs/msg/twist.hpp"
#include "turtlesim/msg/pose.hpp"
#include <cmath>

using std::placeholders::_1;

class TurtleCoop : public rclcpp::Node {
public:
    TurtleCoop() : Node("turtle_coop") {
        // Publisher for the follower turtle's velocity
        follower_pub_ = this->create_publisher<geometry_msgs::msg::Twist>("turtle2/cmd_vel", 10);

        // Subscriber for the leader turtle's pose
        leader_sub_ = this->create_subscription<turtlesim::msg::Pose>(
            "turtle1/pose", 10, std::bind(&TurtleCoop::leaderPoseCallback, this, _1));

        // Subscriber for the follower turtle's pose (Please try to fill in)
        follower_sub_ = ??;
    }

private:
    // Callback for leader turtle's pose
    void leaderPoseCallback(const turtlesim::msg::Pose::SharedPtr msg) {
        leader_pose_ = *msg;
        followLeader();
    }

    // Callback for follower turtle's pose
    void followerPoseCallback(const turtlesim::msg::Pose::SharedPtr msg) {
        follower_pose_ = *msg;
    }

    // Function to make the follower turtle follow the leader
    void followLeader() {
        double distance = std::sqrt(
            std::pow(leader_pose_.x - follower_pose_.x, 2) +
            std::pow(leader_pose_.y - follower_pose_.y, 2));

        // Compute follower orientation to track the leader (Please try to fill in)
        double angle_to_leader = ??;

        double angle_difference = angle_to_leader - follower_pose_.theta;

        // Create a twist message for controlling the follower turtle
        geometry_msgs::msg::Twist vel_msg;

        // Proportional controller for linear and angular velocities
        vel_msg.linear.x = 1.0 * distance;
        vel_msg.angular.z = 5.0 * angle_difference;

        // Publish the velocity to the follower turtle
        follower_pub_->publish(vel_msg);
    }
    // Member variables for storing poses
    turtlesim::msg::Pose leader_pose_;
    turtlesim::msg::Pose follower_pose_;

    // ROS 2 publisher and subscriber objects
    rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr follower_pub_;
    rclcpp::Subscription<turtlesim::msg::Pose>::SharedPtr leader_sub_;
    rclcpp::Subscription<turtlesim::msg::Pose>::SharedPtr follower_sub_;
};

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

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.5)
project(turtle_coop)

# if(NOT CMAKE_CXX_STANDARD)
#   set(CMAKE_CXX_STANDARD 14)
# endif()

# if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
#   add_compile_options(-Wall -Wextra -Wpedantic)
# endif()

# find dependencies
find_package(ament_cmake REQUIRED)
# uncomment the following section in order to fill in
# further dependencies manually.
# find_package(<dependency> REQUIRED)
# TODO for student
find_package(rclcpp REQUIRED)
find_package(geometry_msgs REQUIRED)
find_package(turtlesim REQUIRED)

# TODO for student
# Specify the executable and its source files
add_executable(turtle_coop src/turtle_coop.cpp)
# Link libraries
ament_target_dependencies(turtle_coop rclcpp geometry_msgs turtlesim)

# TODO for student
# Install executables
install(TARGETS
  turtle_coop
  DESTINATION lib/${PROJECT_NAME}
)

# if(BUILD_TESTING)
#   find_package(ament_lint_auto REQUIRED)
#   # the following line skips the linter which checks for copyrights
#   # comment the line when a copyright and license is added to all source files
#   set(ament_cmake_copyright_FOUND TRUE)
#   # the following line skips cpplint (only works in a git repo)
#   # comment the line when this package is in a git repo and when
#   # a copyright and license is added to all source files
#   set(ament_cmake_cpplint_FOUND TRUE)
#   ament_lint_auto_find_test_dependencies()
# endif()

# Mark the package as ament package
ament_package()
```

package.xml

```xml
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd" schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>turtle_coop</name>
  <version>0.0.0</version>
  <!-- TODO for student -->
  <description>Examples of coordinating two turtles in turtlesim</description>
  <maintainer email="xuyilan@u.nus.edu">xuyilan</maintainer>
  <license>Apache-2.0</license>

  <buildtool_depend>ament_cmake</buildtool_depend>
  <build_depend>rclcpp</build_depend>
  <build_depend>geometry_msgs</build_depend>
  <build_depend>turtlesim</build_depend>

  <!-- TODO for student -->
  <!-- <depend>rclcpp</depend>
  <depend>geometry_msgs</depend>
  <depend>turtlesim</depend> -->

  <exec_depend>rclcpp</exec_depend>
  <exec_depend>geometry_msgs</exec_depend>
  <exec_depend>turtlesim</exec_depend>

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

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```