IEEE-HKN Workshop
Introduction to ROS 2

Cheng Jia Wei Andy

Contents

Chapter 1

Acknowledgements

I would like to thank my friends and seniors at Bumblebee for teaching me whatever I know today and IEEE-HKN for the opportunity to have taught this workshop. And you, dear reader/participant, for taking the time and effort to go through this material. My hope is that by the end of it, you would have learnt something new and feel somewhat inspired and curious to look more into this on your own, with "this" being whatever topic that interests you! ♥

Chapter 2

Pre-requisites

Please ensure that you have the following set up before proceeding with the remaining material

- Ubuntu OS or WSL2 with Ubuntu (if running on Windows)
  WSL2 installation instructions can be found here
  I recommend using the command line to install WSL2
- ROS2 Humble
  Installation instructions for Ubuntu can be found here
  I recommend installing `ros-humble-desktop` to avoid any potential complications

Chapter 3

Motivation and Yap

Before diving into the details, let's take a moment to understand why we might want to use ROS2. What's in it for us, and what's the point of learning it? Depending on who you ask, ROS2 can be seen as either a blessing or a challenge. At its core, ROS2 provides abstraction—a way to simplify complex processes by focusing only on what's necessary for us to achieve our goals.

So, what is abstraction? Simply put, it means saying, "I don't need to understand all the intricate details; I just need it to work, and I need to know how to interact with it." Let's use a simple analogy to make this clearer.

3.0.1 A Practical Analogy: You and a Friend

Imagine you're trying to tell a friend where you are. Your friend is far away, eagerly waiting to hear from you. In this scenario, let's introduce some basic ROS2 terminology that we'll explore more deeply later:

- **You** are the **publisher** because you're actively sharing information (your location).
- **Your friend** is the **subscriber** because they're listening for that information.

Now, let's say you shout your location at the top of your lungs. If your friend still can't hear you, you've hit a communication barrier. But then, you both realize you have phones. You text your location to your friend, who sees the message and instantly knows where you are.

3.0.2 Abstraction Through Phones

In this example, the **phone** acts as the abstraction mechanism. Most of us don't know exactly how phones transmit messages—what happens behind the scenes to convert text into data and send it through the network. But we don't need to know. All we care about is that the phone works and how we can use it to send or receive messages. That's abstraction in action: hiding the complex details while letting us focus on the result.

3.0.3 Abstraction in ROS2

Similarly, in ROS2, **you (the publisher)** would send your location to a specific **topic** (another key term we'll discuss later). **Your friend (the subscriber)** would listen to that topic to get the information. ROS2 handles the entire communication process for you—you don't need to know exactly how it works under the hood. All you need is to understand what it does and how to make it work for your needs.

This abstraction makes life easier. ROS2 takes care of the heavy lifting so you can focus on solving the problems that matter. However, unlike using a phone, where most of us are already familiar with how to use it, ROS2 requires you (as a developer or user) to learn how to interact with it and leverage its features.

This simple communication process is just one part of what ROS2 can do. In the coming chapters, we'll explore the basics of ROS2, step by step, to help you get started on your own journey.

Chapter 4

Workspaces (for your code, not you)

To use ROS2 effectively, we need to understand how to organize our code. Most of it lives in a **workspace**; essentially a special folder where ROS2 projects are built and managed. While it's just a regular directory, it's structured to support ROS2 development, making it a key part of working with ROS2. The following images are of a ROS2 workspace containing Python packages (more on this later) `simple_publisher` and `simple_subscriber`.

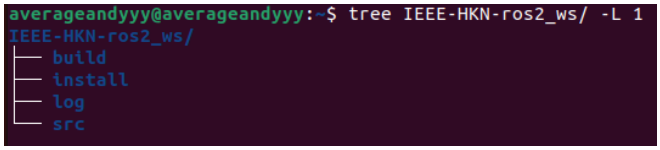

Figure 4.1: Base level of the workspace

Figure 4.2: Structure of `src` where most your written code will live

```
# Create workspace directory
mkdir ros2_ws
# Go into workspace and create src directory
cd ros2_ws
mkdir src
```

Congratulations! You've just created your workspace! (I did say they were just folders) Now, let's move on to creating packages—these are like folders but with a bit more structure to hold your code and other essentials.

Chapter 5

Packages (not from Shopee)

Packages are a way to logically organize your code. For example, you might have a package called `camera_drivers` for interfacing with cameras and another called `ml` for machine learning tasks. Typically, a package is significant enough to justify its own Git repository. Let's start by creating a package that contains a simple publisher.

```
# In the /src directory, run the following in the terminal
source /opt/ros/humble/setup.bash
ros2 pkg create --build-type ament_python simple_publisher
```

And you're done! Remember when I said that a package typically warrants its own Git repository? For today's workshop, given its size and simplicity, you might want to make the workspace the repository instead for easy reference in the future.

Chapter 6

I Wish To Stop Shouting

6.1 Writing a Publisher

Here's the code to create a simple publisher!

```python
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class SimplePublisher(Node):
    def __init__(self):
        super().__init__('simple_publisher') # name of node
        self.publisher = self.create_publisher(String, 'topic', 10) # to change topic name
        timer_period = 0.5 # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)

    def timer_callback(self):
        msg = String()
        msg.data = 'Hello World!'
        self.publisher.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)

def main(args=None):
    rclpy.init(args=args)
    simple_publisher = SimplePublisher()
    rclpy.spin(simple_publisher)

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

Python is a programming language that supports object-oriented programming (OOP). This means it allows us to use classes and objects to model our code, either based on real-world counterparts (if they exist) or abstract concepts. With OOP, we can translate these ideas into code that can run and function on a computer.

Here, we created a `SimplePublisher` class that inherits from the `Node` class. In ROS2, a `Node` is formally the smallest unit of computation. Informally, it's just something that performs useful tasks. While we don't always need to understand how it works internally, we focus on how to use it to achieve our goals. `SimplePublisher` is a more specialized node designed to handle the specific task of publishing.

The `self.create_publisher` function is part of the `Node` class, developed by the experts behind ROS2. Although we may not know the exact mechanics of how it creates a publisher internally, understanding how to use it is enough. This function takes three parameters: the message type (e.g., `std_msgs/String`, which is different from Python's basic string), the name of the topic to publish to, and the queue size, which determines how many messages can be held at a time.

Next, we have `self.create_timer`, which takes two parameters: the timer period (in seconds) and a callback function. This creates a timer that periodically invokes the callback function at the specified interval. In our case, the callback publishes a string to the topic and logs the published string.

Finally, we include some standard boilerplate code to run the node. The `rclpy.spin` function keeps the node alive and running, ensuring it can continue performing its tasks.

6.2 Building the Package

Before running the node, we need to set up the package for building. Start by modifying `package.xml` to look like the following:

```
<?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>simple_publisher</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="averageandyyy@gmail.com">averageandyyy</maintainer>
  <license>TODO: License declaration</license>

  <!-- Added dependencies -->
  <depend>rclpy</depend>
  <depend>std_msgs</depend>

  <test_depend>ament_copyright</test_depend>
  <test_depend>ament_flake8</test_depend>
  <test_depend>ament_pep257</test_depend>
  <test_depend>python3-pytest</test_depend>

  <export>
    <build_type>ament_python</build_type>
  </export>
</package>
```

Adding dependencies ensures the necessary tools and libraries are available when building the package. If any are missing, the build process will fail, which is helpful because the code wouldn't work without them. After this, update `setup.py` to look like the following:

```
from setuptools import find_packages, setup

package_name = 'simple_publisher'

setup(
    name=package_name,
    version='0.0.0',
    packages=find_packages(exclude=['test']),
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='averageandyyy',
    maintainer_email='averageandyyy@gmail.com',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
```

```
    entry_points={
        'console_scripts': [
            # Added entrypoint
            'yapper = simple_publisher.publisher:main',
        ],
    },
)
```

Invoking `yapper` from the terminal later executes the `main` function in the code, which starts the node. In this example, the `main` function is in the same file as the class definition. However, in larger projects, class definitions are usually placed in separate files for better organization. To finally build the package, navigate to your workspace, just outside of `src` and run the following

```
# Install the builder (only happens once, no need to run ever again after install)
sudo apt install python3-colcon-common-extensions

# Build (just) the package
colcon build --packages-select simple_publisher

# To build all packages at once
colcon build
```

6.3 Running the Publisher

From the workspace directory, run the following to the publisher in action

```
# Run this every time after a build, sets up environment to run own packages in terminal
source install/setup.bash

# Format: ros2 run package_name entrypoint_name
ros2 run simple_publisher yapper
```

Chapter 7

Please Stop Shouting I Am Subscribed

Try repeating the steps above, but this time, for a subscriber instead. In the workshop, I created a new package just for the subscriber alone to demonstrate the process again but you definitely could put the subscriber in the same directory or even file where the publisher was defined. Try it out and experiment! The code for the subscriber can be found here

After setting up the subscriber, trying running both simultaneously to see the publisher and subscriber in action. We've just demonstrated a very simple example of a typical publisher/subscriber framework that only involved strings. Such a framework can very easily extended to more complicated tasks that require synchronisation and communication, allowing to compose and string together additional components to form more complicated tasks.

Chapter 8

Say Hi to Thomas in `turtlesim`

`turtlesim` is a simulation package developed by the folks over at ROS2. At its core, it is just a 2D simulation involving a turtle that can move across a 2D plane aka your screen. We'll now see how we can run the simulation and interact with it.

```
# On one terminal, run
ros2 run turtlesim turtlesim_node

# On another, run
ros2 run turtlesim turtle_teleop_key
# This allows us to use our keyboard and move the turtle!
```

Feel free to take some time and experiment! Move the turtle around and see what it can or cannot do.

8.1 Moving Thomas

I've decided to name our turtle friend Thomas, feel free to name yours whatever you'd like. As we have tried, we've managed to make Thomas move through our keyboard inputs but real turtles move on their own, not according to keyboard strokes. Let's try and make Thomas move on this own instead!

Just like any other software engineering project, we need to understand:

1. Our working environment, which dictates our constraints and things we need to both work with and work around.
2. Our desired outcome, how we want our turtle to move.

Now, as mentioned earlier, we weren't the ones who made Thomas and so we need to do our homework to learn how Thomas works underneath the hood and what specific components we need to interact with to invoke our desired outcome.

```
# On one terminal run the following while Thomas is being simulated
ros2 topic list
# This shows us the list of active topics that are available while Thomas is running

# On more terminals, run the following
ros2 topic info /turtle1/pose
ros2 topic echo /turtle1/pose
# The first shows us the message type and the second shows us the actual message being
# communicated in real-time

# Run the following next
ros2 topic info /turtle1/cmd_vel
ros2 topic info /turtle1/cmd_vel
# Run all the above while teleop is operational to see how the values change as
```

```
# we move Thomas around
```

Intuitively and through observation, we see that `pose` tells us where the turtle is on the screen and the angle by which it is facing. `cmd_vel` then tells us how fast and the direction aka the velocity that Thomas is moving at. We see that these messages/values change depending on the keystrokes that we invoke, which tells us quite a fair bit! Before we look further into that, let's think about how we want Thomas to move.

While we can think of and make Thomas move as a point mass, we want to instead make Thomas move in a way that makes intuitive sense, much like a turtle! Given a desired destination, we want to

1. Rotate Thomas to face the desired destination
2. Move towards the desired destination
3. Once at the desired destination, rotate once more to fast the desired orientation

In the rest of the workshop, we fleshed out more of these requirements and how we combined our requirements together with the constraints posed by Thomas such as the topics that we have to work with and eventually arrived at code that made Thomas move autonomously. The following pages are spaces that are hopefully filled with drawings and annotations that reflect the development process that we shared together. The full code for Thomas' autonomous movement can be seen here.

As a note for myself, we should have covered

1. Where is Thomas right now?
2. Where does Thomas need to go?
3. When should Thomas move? When should Thomas stop?
4. How do I tell Thomas to move?
5. Rotational and vector math, cover a few examples to show that the math works and is correct
6. Document development process: I am clueless and noob, but how did I become less noob?
7. Introduction into PID: P component and observe how Thomas slows down as he approaches his goal
8. General software engineering processes

Chapter 9

Space 1 for Workshop Development

Chapter 10

Space 2 for Workshop Development

Chapter 11

Space 3 for Workshop Development

Chapter 12

Closing Words

Thank you dear reader/participant for having taken the time to go through the material or attend the workshop in person. I hope you had found the session fruitful and hopefully learnt a thing or two about ROS. In coming up with the workshop material, I had hoped to strike a balance between a beginner friendly introduction to the uninitiated and a simple yet non-trivial step up to showcase what ROS can do, in hopes of inspiring you, the reader, to look more into and perhaps use ROS in your projects, if suitable.

Past workshops that I've attended typically left me feeling disappointed with a sense of "That's it?". If you felt that you were a victim of yet another underwhelming workshop, or on the contrary, felt overwhelmed by the content of the workshop, I strongly encourage you to reach out via **e1155277@u.nus.edu** and let me know how I can improve or any changes I can make or add to the workshop material to hopefully make it more engaging. Your feedback is, seriously (no cap), appreciated.

If you'd like to see more of ROS2, do check out their documentation page or let me know if you'd like to see this expanded to cover more topics! Thank you again for showing up or going through this material! ♡