Tinkerers' Lab
IIT Hyderabad

Mechatronics Summer School 2025

Prepared by:

The Senior Mechatronics Team

Jaideep Nirmal AJ, *Head*
Abhijith Raj, *Senior Core*
Satyavada Sri Harini, *Senior Core*
Taha Mohiuddin, *Senior Core*

Tinkerers' Lab, IIT Hyderabad
Between A, B, G, H Hostel Block, IIT Hyderabad
IITH Main Road, Near NH-65, Sangareddy, Telangana 502285

Tinkerers' Lab
IIT Hyderabad

.

Module 1
ROS 101: Getting Started

Contents

1 A Welcome Note

Welcome to the Mechatronics Edition of Tinkerers' Laboratory (IIT Hyderabad) Summer School 2025! This summer school is designed to provide a solid foundation in ROS2 - The Robot Operating System.

We will start from the basics, some theory and get up to a point where you can build an intelligent robot with ROS, all by yourself. Do note that this summer school requires a good amount of effort on your part. Be sure to try out everything mentioned in the book to get a bigger picture of this dynamic tool. In case you feel some things here just don't make sense, then we suggest you to watch YouTube videos from either *Articulated Robotics* (goated) or *Robotics Back-End* channel.

Note that a significant amount of effort has been put to generate this document. Currently, this is exclusive content meant to be circulated within the Tinkerers' Lab IITH team only. Kindly refrain from sharing with people outside Tinkerers' Lab IITH.

2 The Basics

2.1 What is ROS?

Robot Operating System (ROS or ros) is an open-source robotics *middleware* suite.

ROS is a framework designed to simplify the development of robotic systems. It provides a collection of tools, libraries, and conventions for building modular and reusable robotic software. ROS operates using a distributed computing architecture, where individual software processes (called nodes) communicate via a publish-subscribe messaging system or services. This architecture allows developers to integrate sensors, actuators, and control systems seamlessly, enabling efficient communication and coordination between components.

When you build a robot, you would want to focus on making sure the robot performs the task you are building it for. But even for the smallest of the robots, there are several behind-the-scenes tasks that are necessary to be done, e.g. task scheduling, memory management, etc. But these tasks are so "low-level" or close to the processor that it no longer is a "robotics" job. Yet, these are necessary to make the robot work as intended.

ROS comes to the rescue here. ROS's philospohy is simple: *Do not reinvent the wheel.*

ROS has several inbuilt tools which make programming a robot easy. Being open source, the global ROS community offers numerous pre-built packages for functionalities like navigation, computer vision, and motion planning. This exponentially decreases the development time, giving roboticists more time to focus on the necessary parts of their project.

Figure 1: Behold, the ROS logo

2.2 FAQs at this point

1. Should I learn a new programming language (sigh)?
   No. ROS is just a framework. It supports programming in two languages - C++ and - everybody's favorite - Python. You can do as much with Pyton as with C++, there are mostly no sides to pick.

2. But why should I care about this?
   ROS is the de facto industry standard in Robotics Development. It is widely used in academic research and autonomous vehicles. It is a must-have skill for those who are see themselves in the Robotics Industry in the future (*And c'mon, tis gonna look nice on your CV*)

3. What's the *OS* in R*OS* about?
   ROS is not an actual Operating System. Its just a bunch of software that makes developing a robot easy.

4. Do I need a robot to run ROS?
   No. ROS runs really well on Linux systems. A Linux machine is all you need.

5. Wait, I dont get this
   Robots are just computers with some moving parts. And the robotics industry loves Linux. So ROS runs really well on Linux systems. The nearest that we, at TL, can get with a Robot is with a Raspberry Pi. R-Pi's can run Linux and we run ROS on Linux. And do wonders.

2.3 The Virtual World

Anyone who has a decent experience with coding knows how messy debugging a program can get.One can usually not afford to build a robot (which is usually expensive) and end up wrecking it because of a some bug in the code.

This is where simulations become helpful. Before actaully building the robot, we simulate it. *Gazebo* is the widely used simulation software for ROS applications. Gazebo gives the ability to build custom worlds and spawn robots into it. It even emulates the sensor data (camera, LiDAR, etc) giving more life to the simulation.

ROS works very well with Gazebo. This is very handy as one need not tweak their code in order for it to run on the simulator. The code that runs in a Gazebo world will run just as fine in the real world.

3 History of ROS

To those who are interested

The Robot Operating System (ROS) originated as a project at Stanford University's Artificial Intelligence Laboratory in 2007, where it was developed to support the STAIR (Stanford AI Robot) project. The goal was to create a standardized, open-source middleware framework for robotic software development that could work across diverse hardware platforms.

In 2008, the development of ROS shifted to Willow Garage, a robotics research lab in Menlo Park, California. Willow Garage played a pivotal role in nurturing ROS and released its first stable version, ROS 1.0, in 2010. This version introduced a modular and distributed architecture that allowed developers to build robotic systems by combining reusable software modules. Following Willow Garage's closure in 2014, the Open Source Robotics Foundation (OSRF) took over the stewardship of ROS.

Over the years, ROS gained widespread adoption in both academia and industry due to its open-source nature, adaptability, and robust community support. However, limitations in real-time processing, scalability, and security led to the development of ROS 2, which was first released in 2014 as a next-generation platform to address these challenges.

ROS 2 addresses many of the limitations of ROS 1 by introducing features essential for modern robotics applications:

- Real-time capabilities for time-critical tasks.
- Scalability for large-scale systems.
- Enhanced security features.
- Decentralized architecture that eliminates single points of failure.

While ROS 1 remains widely used in legacy systems and research projects, its final distribution (Noetic) will only be supported until 2025. Therefore, transitioning to ROS 2 is recommended for long-term projects and cutting-edge applications

Distro	Release date	Logo	EOL date
Iron Irwini	May 23rd, 2023		November 2024
Humble Hawksbill	May 23rd, 2022		May 2027
Galactic Geochelone	May 23rd, 2021		December 9th, 2022
Foxy Fitzroy	June 5th, 2020		June 20th, 2023
Eloquent Elusor	November 22nd, 2019		November 2020
Dashing Diademata	May 31st, 2019		May 2021
Crystal Clemmys	December 14th, 2018		December 2019
Bouncy Bolson	July 2nd, 2018		July 2019
Ardent Apalone	December 8th, 2017		December 2018
beta3	September 13th, 2017		December 2017
beta2	July 5th, 2017		September 2017
beta1	December 19th, 2016		Jul 2017

Figure 2: ROS2 Releases, the ones in green are in support (as of May 2025)

4 Versions of ROS

Starting from May 2010, one iteration of ROS has been released every year till 2020. The last ROS version was ROS Noetic Ninjemys. The first version of ROS-2- ROS-2 Ardent Apalone was released in 2017. There is a new ROS 2 distribution released yearly on May 23rd (World Turtle Day). As of making this document, the latest version of ROS-2 is Jazzy Jalisco released in 2023.

However, the version widely used is ROS-2 Humble Hawksbill released in 2022 and this is the version we shall use here on a Ubuntu 22.04 system. Henceforth in this document, "ROS" by default refers to "ROS-2".

It should be possible to run ROS from the macOS Terminal.

Each version of ROS has its own favourite Ubuntu distribution that it works best with. The details are in ROS's website. Note that although ROS can be installed on a Windows system, it is suggested you dual boot your system to Ubuntu 22.04 LTS to make things smoother.

NOTE

For this Summer School, use **ROS-2 Humble Hawksbill** on a **Dual-Booted** Ubuntu **22.04 LTS** installation

5 ROS Installation

1. Dual boot your laptop with Ubuntu 22.04 LTS (infinite videos on YouTube)
2. Ensure Step 1 is followed. Virtual Machines do not deliver nearly the same performance as dual booting and running on bare metal
3. You now need to install ROS2 on your system using the **deb packages** methods as given here. A brief walkthrough is provided here.
4. Save this script as `install_ROS2.sh` in your `/home` directory.

```
#!/bin/bash

set -e

echo "=== Setting locale ==="
locale

sudo apt update
sudo apt install -y 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

echo "=== Setting up ROS 2 sources ==="
sudo apt install -y software-properties-common
sudo add-apt-repository universe

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

echo "=== Adding ROS 2 repository ==="
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

echo "=== Updating package index and upgrading system ==="
sudo apt update

echo "=== Installing ROS 2 Humble Desktop ==="
sudo apt install -y ros-humble-desktop

echo "=== ROS 2 installation completed successfully ==="
```

5. Now, run these commands

```
Terminal
$ chmod +x install_ros2.sh
$ ./install_ros2.sh
```

Enter your password if prompted. The ROS installation should not take more than 15 minutes with a stable internet connection.

6. Every time you open a new terminal, the ROS2 files need to sourced in order to use them. This needs to be done using the `source` command. However, it becomes a cumbersome job while dealing with a number of terminals, so we automate it by running it, everytime a terminal is opened, by adding the path to `.bashrc` file.

```
Terminal
$ echo "source /opt/ros/humble/setup.bash" »  /.bashrc
```

Now open a new terminal. If you get any error message, then follow these steps:

(a) In the same terminal,

```
Terminal
$ sudo apt install gedit -y $ gedit .bashrc
```

(b) Now, a text editor should appear. Go to the last line of the file and add this:

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

(c) Hit `Ctrl+S` and close the window.

7. Your ROS2 installation is now complete. To check if it is installed properly, follow these steps

(a) Open a terminal and type this command

```
Terminal 1
$ ros2 run demo_nodes_cpp talker
```

(b) Open a second terminal and type this command

```
Terminal 2
$ ros2 run demo_node_pylistener
```

You should see the talker saying that it's `Publishing` messages and the listener saying `I heard` those messages. This verifies both the C++ and Python APIs are working properly. Hooray!

6 The World of ROS

6.1 The Bigger Picture

ROS 2 is a communication framework for building robot software. Instead of writing one giant program, ROS encourages modular design: small programs (called *nodes*) interact with each other. ROS 2 is built for reliability and flexibility, with support for real-time systems, multiple programming languages, and distributed computing.

6.2 Nodes

A *node* is a single-purpose program in a ROS system. Each node should do one thing well—like controlling a motor or reading a sensor. Nodes are the basic building blocks of any ROS-based robot. ROS 2 automatically handles communication between nodes so they can work together like a team.

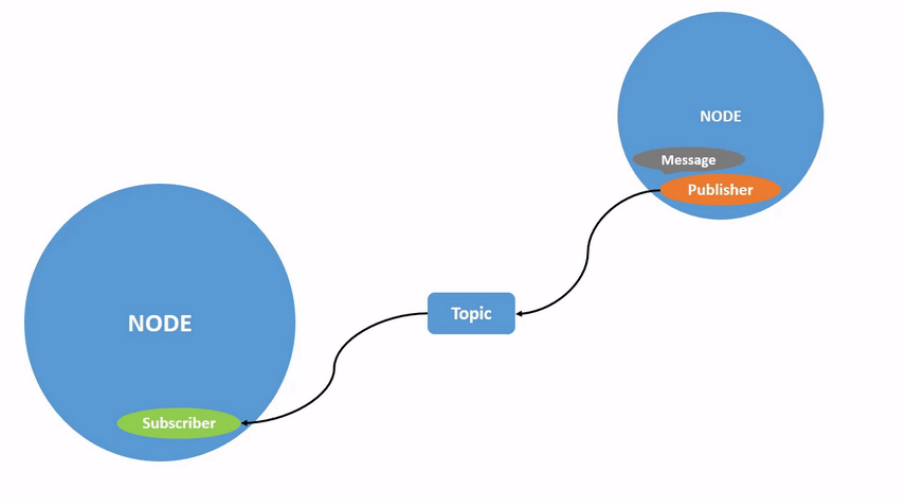

Figure 3: Message at Publisher Node

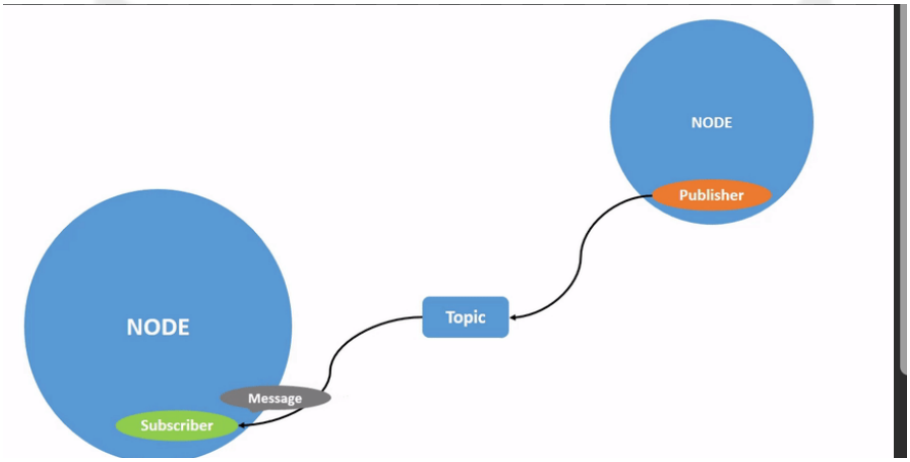

Figure 4: Messasge at Subscriber Node

Lifecycle of a Node

1. **Unconfigured**
   The node has been created but has not yet been initialized. Parameters and resources are not yet set up.

2. **Inactive**
   The node has been configured, but it is not currently performing its main functionality. Publishers, timers, and subscriptions are inactive. This state is used for safe initialization before activation.

3. **Active**
   The node is fully operational. It can communicate over topics, services, and actions. This is the main working state of a node.

4. **Finalized (Shutdown)**
   The node has been cleaned up and is no longer in use. All resources have been released.

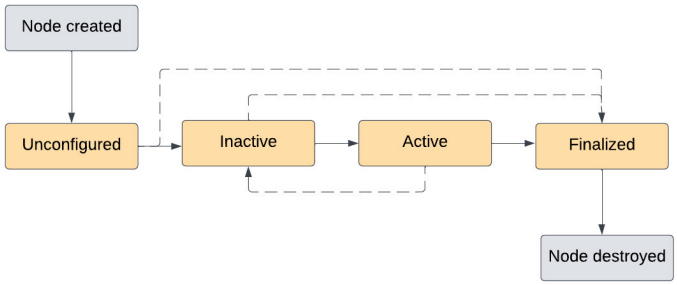

Figure 5: Lifecycle of a Node

6.3 Topics

Topics are a core concept in ROS 2 used for communication between nodes using a **publish/subscribe** model.

- A **publisher** is a node that sends messages on a topic.
- A **subscriber** is a node that receives messages from that topic.

This model allows for **loose coupling** between nodes — the publisher doesn't need to know which nodes (if any) are listening, and subscribers don't need to know who is publishing.

Topics are ideal for **continuous or streaming data**, such as sensor readings, camera feeds, or real-time control commands.

Simple Publisher and Subscriber in Python

Let's walk through a basic example where one terminal runs a publisher and another runs a subscriber, both using the `/chatter` topic.

1. **Terminal 1 – Run a talker (publisher):**

```
Publisher                                                    - □ X
ros2 run demo_nodes_py talker
```

This command starts a node that publishes a string message (e.g., "Hello World") repeatedly on the `/chatter` topic.

2. **Terminal 2 – Run a listener (subscriber):**

```
Subscriber                                                   - □ X
ros2 run demo_nodes_py listener
```

This subscriber node listens to the `/chatter` topic and prints any messages it receives.

You'll see the publisher printing messages like:

```
[INFO] [1747661792.696455426] [talker]: Publishing: "Hello World: 0"
[INFO] [1747661793.677762270] [talker]: Publishing: "Hello World: 1"
[INFO] [1747661794.677970611] [talker]: Publishing: "Hello World: 2"
[INFO] [1747661795.677502009] [talker]: Publishing: "Hello World: 3"
[INFO] [1747661796.677839779] [talker]: Publishing: "Hello World: 4"
[INFO] [1747661797.678026972] [talker]: Publishing: "Hello World: 5"
[INFO] [1747661798.677851620] [talker]: Publishing: "Hello World: 6"
[INFO] [1747661799.677968696] [talker]: Publishing: "Hello World: 7"
```

You'll see the listener printing messages like:

```
[INFO] [1747661792.709891028] [listener]: I heard: [Hello World: 0]
[INFO] [1747661793.679151489] [listener]: I heard: [Hello World: 1]
[INFO] [1747661794.679471036] [listener]: I heard: [Hello World: 2]
[INFO] [1747661795.678168212] [listener]: I heard: [Hello World: 3]
[INFO] [1747661796.679254730] [listener]: I heard: [Hello World: 4]
[INFO] [1747661797.679515283] [listener]: I heard: [Hello World: 5]
[INFO] [1747661798.679535750] [listener]: I heard: [Hello World: 6]
[INFO] [1747661799.679482361] [listener]: I heard: [Hello World: 7]
```

Keep in mind

You need to run these commands in two separate terminals for the communication to work.

Run the command below to get a list of all the topics you can run:

```
List Topics                                                  - □ X
ros2 topic list
```

6.4 Services

Services enable two-way communication between nodes using a **request-response** model. A client node sends a request, and a server node replies. This is useful for quick, one-off

tasks like querying a sensor status or setting a parameter.

Unlike topics, services are **synchronous** — the client waits for the server's reply before moving on.

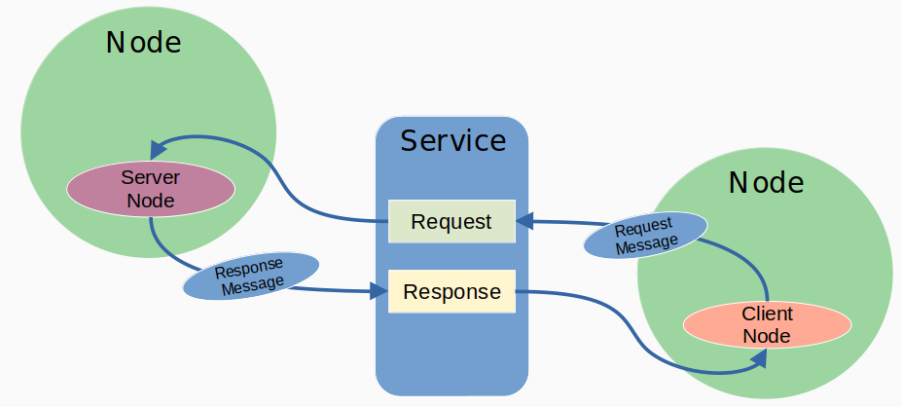

Figure 6: ROS Architecture

6.5 Parameters

Parameters are used to store configurable values in a node, such as maximum speed, sensor thresholds, or mode settings. They allow you to adjust behavior without modifying the code, and can be read or updated during runtime for flexibility.

By default, a node must declare all the parameters it intends to use during its lifetime. This ensures that the type and name of each parameter are well-defined at node startup, reducing the chances of misconfiguration. However, for scenarios where not all parameters are known ahead of time, a node can be instantiated with `allow_undeclared_parameters` set to `true`, permitting the use of parameters that haven't been explicitly declared.

Types of Parameters

Each parameter consists of a key (a string) and a value, which can be one of the following types:

- `bool`
- `int64`
- `float64`
- `string`
- `byte[]`
- `bool[]`
- `int64[]`

- `float64[]`
- `string[]`

Interacting with Parameters Using the Terminal

ROS 2 provides built-in command-line tools to manage and inspect parameters at runtime. These commands allow users to list, get, set, and persist parameter values.

List Parameters
```
ros2 param list
```

Get Parameter Value
```
ros2 param get <node_name> <parameter_name>
```

Set Parameter Value
```
ros2 param set <node_name> <parameter_name> <value>
```

Dump Parameters to YAML File
```
ros2 param dump <node_name>
```

Load Parameters from YAML File
```
ros2 param load <node_name> <file.yaml>
```

6.6 Actions

Actions are designed for long-running tasks that require progress updates or the ability to cancel, like moving a robot arm or navigating to a goal. Actions split the process into three stages: goal request, feedback updates, and final result.

They are implemented using a client-server architecture, where the action client sends a goal to the server, the server processes the request over time, and periodically sends feedback to the client. Once the task is completed (or canceled), the server sends a final result message. This pattern is useful when tasks may take seconds or minutes to complete, as it allows the client to monitor the task and respond to changes in real-time, such as issuing a cancel request or reacting to progress feedback. Actions thus provide a robust asynchronous interface that combines the best aspects of services (goal-result interaction) and topics (continuous updates).

6.7 Comparing Topics, Services, and Actions

Feature	Topics	Services	Actions
Communication Type	One-way (publish → subscribe)	Two-way (request ↔ response)	Goal-oriented with feedback and result
Timing	Asynchronous, continuous	Synchronous, blocking	Asynchronous, supports long-running tasks
Use Case	Continuous data streams	Quick, one-time tasks	Tasks needing feedback, cancellation, or delay
Pattern	One-to-many	One-to-one	Client-server with progress updates
Examples	Sensor data, control commands	Set configuration, query state	Navigate to point, pick-and-place operation

So basically:

- Use **topics** when you want to broadcast data constantly.
- Use **services** for quick, on-demand interactions.
- Use **actions** when you need long-running behavior with updates.

7 Hello, World! - ROS Edition

7.1 Workspace

A workspace is a directory containing ROS 2 packages. It is always a standard practice to have designate a specifc folder as a *workspace* and store all the projects in that folder only. This is a guideline and it is suggested that everyone follows this to keep files organised.

Before using ROS 2, it's necessary to source your ROS 2 installation workspace in the terminal you plan to work in. This makes ROS 2's packages available for you to use in that terminal.

Colcon is the tool used to build ROS packages. First step is to install the colcon extensions

```
Terminal                                                    - □ X
$ sudo apt install python3-colcon-common-extensions
```

Source

In case you have not installed ROS via the tutorial given in this book, then depending on how you installed ROS 2 (from source or binaries), and which platform you're on, your exact source command will vary. Consult the installation guide you followed if these commands don't work for you.

```
Terminal                                                    - □ X
$ source /opt/ros/humble/setup.bash
```

As this is a redundant process, we have set up our terminal such that ROS files are loaded everytime a new terminal is pulled up as we have added a line to the `.bashrc` file in the installation guide in this book.

PRO TIP

Do not forget to source the ROS directory to the Terminal everytime you open one. To automate this for you, add the line 'source /opt/ros/humble/setup.bash' to your '.bashrc' configuration file, as given in the installation tutorial.

Build the Workspace:

First create a folder. The name doesn't matter, but it is helpful to have it indicate the purpose of the workspace. Let's choose the directory name ros2_ws, for "development workspace":

```
Terminal
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws/src
colcon build
```

The `colcon build` command builds all the ROS packages in a workspace.

Workspace Structure:

A ROS workspace is a directory with a particular structure. Commonly there is a `src` subdirectory. This folder structure appears automatically once you build the workspace. Inside that subdirectory is where the source code of ROS packages will be located.

- The `build` directory will be where intermediate files are stored. For each package (package = project) a subfolder will be created
- The `install` directory is where each package will be installed to. By default each package will be installed into a separate subdirectory.
- The `log` directory contains various logging information about each colcon invocation.

Sourcing the install directory:

We sourced the ROS directory into the `.bashrc` file. And yes, you guessed it right - we also need to source the `install` directory of the workspace into the terminal everytime we need to run a package. We can do this using this command:

```
echo "<PATH_TO_WORKSPACE>/ros2\_ws/source install/setup.bash" >> ~/.bashrc
```

If that doesn't work (you know how to check!) follow the instructions in the installation section.

NOTE

For every workspace you create, you will have to source its `install` directory in the `.bashrc` file.

TROUBLESHOOTING

Don't worry if you cant see the `install` directory in the workspace. Just build the workspace again and it should appear. Everytime you build the workspace, your `install` directory gets updated.

7.2 Package Creation

Package is the unit that is responsible for organizing your ROS code. It allows you to share your code with others. So for the creation of a package ROS uses "ament" and "colcon". In ROS 2, ament is the build system, while colcon is the build tool. In essence, ament provides the rules and structure for building ROS 2 packages, while colcon is the tool that runs those rules to build the software. Now let's create our first package:

Create a Package:

The syntax to create a package is

```
Terminal                                                    - □ X
ros2 pkg create --build-type ament_python <PACKAGE_NAME>
```

Make sure you're in the SRC folder of the workspace whenever you create a package

NOTE

If you want to create a python package replace ament_cmake with ament_python.

For this tutorial, you will use the optional argument –node-name which creates a simple Hello World type executable in the package. Enter the following command in your terminal:

```
Terminal                                                    - □ X
ros2 pkg create --build-type ament_cmake --license Apache-2.0 --node-name my_node
my_package
```

You will now have a new folder within your workspace's SRC directory called my_package.

Build the Package:

Now to build the package return to the root of your workspace, and run the following command:

```
Terminal                                                    - □ X
colcon build
```

This builds all the packages that you have in the workspace which is fine with few packages but to build a particular package (here my_package) use:

```
Terminal                                                    - □ X
colcon build --packages-select my_package_1 my_package_2
```

Note that whenever you make any change to the package files, you need to rebuild that package in order to see the change. In a way, building is similar to compiling the code.

LOOK BEFORE YOU BUILD!

Make sure you are in the **root** of your workspace directory when running the `colcon build` command. Otherwise, you might end up messing your workspace folder structure.

Another useful argument to `colcon` is the symlink-install. It is particularly useful when nodes are written in Python. As python code doesn't require compilation of code, the ROS package also need not be built again and again.

```
Terminal                                                    - □ X
colcon build -symlink-install
```

Source the Setup File:

To use your new package and executable, first open a new terminal and source your main ROS 2 installation. Then, from inside the ros2_ws directory, run the following command to source your workspace:

```
Terminal                                                    - □ X
source install/local_setup.bash
```

Use the Package: To run the executable you created using the –node-name argument during package creation, enter the command and you will get the following output

```
Terminal                                                    - □ X
$ ros2 run my_package my_node
hello world my_package package
```

7.3 A Publisher

Nodes are executable processes that communicate over the ROS graph. In this tutorial, the nodes will pass information in the form of string messages to each other over a topic. The example used here is a simple "talker" and "listener" system; one node publishes data and the other subscribes to the topic so it can receive that data.

Create Package

Navigate into the ros2_ws directory created in a previous tutorial. Recall that packages should be created in the src directory, not the root of the workspace. So, navigate into ros2_ws/src, and run the package creation command:

```
Terminal                                                    - □ X
$ ros2 pkg create --build-type ament_python --license Apache-2.0 py_pubsub
```

WATCH OUT!

Make sure you are in the `src` folder only when you create a package. Otherwise -as you might have guessed- you'll end up messing the workspace folder structure

Navigate into ros2_ws/src/py_pubsub/py_pubsub. Recall that this directory is a Python package with the same name as the ROS 2 package it's nested in. Once you do this, you should be having the following file structure:

```
Terminal                                                    - □ X
~ros2_ws/src$ cd py_pubsub
~ros2_ws/src/py_pubsub$ ls -1
LICENSE
package.xml
py_pubsub
resource
setup.cfg
setup.py
test
~ros2_ws/src/py_pubsub$ cd py_pubsub
~ros2_ws/src/py_pubsub/py_pubsub$ code .
```

The `py_pubsub/py_pubsub` folder is where you put all your python files. The subfolder will have `__init__.py` by default. Open that subfolder in VS Code using the command `code .`. Make sure you have Visual Studio Code installed using `apt` or the snap store.

Write the Code

Now create a python file named publisher_member_function.py adjacent to __init__.py (it can be found inside py_pubsub. Then enter the following code in it.

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

class MinimalPublisher(Node):
    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, '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 World: %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)
    minimal_publisher = MinimalPublisher()
    rclpy.spin(minimal_publisher)

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


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

Add Dependencies

Navigate one level back to the ros2_ws/src/py_pubsub directory, where the setup.py, setup.cfg, and package.xml files have been created for you. Open package.xml with your text editor. As mentioned in the previous tutorial, make sure to fill in the <description>, <maintainer> and <license> tags:

```
<description>Examples of minimal publisher/subscriber using rclpy</
    description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache License 2.0</license>
```

After the lines above, add the following dependencies corresponding to your node's import statements:

```
<exec_depend>rclpy</exec_depend>
<exec_depend>std_msgs</exec_depend>
```

This declares the package needs rclpy and std_msgs when its code is executed. Make sure to save the file.

Add to an Entry Point:

Open the setup.py file. Again, match the maintainer, maintainer_email, description and license fields to your package.xml:

```
maintainer='YourName',
maintainer_email='you@email.com',
description='Examples of minimal publisher/subscriber using rclpy',
license='Apache License 2.0',
```

Add the following line within the console_scripts brackets of the entry_points field:

```
entry_points={
    'console_scripts': [
        'talker = py_pubsub.publisher_member_function:main'
        ],
},
```

ATTENTION!

Always double check your `entry_points`. Notice that it is a `str:list` dictionary- the list should not have any trailing or stray commas. If something goes wrong, this should be the first point to check

Check `setup.cfg`

The contents of the setup.cfg file should be correctly populated automatically, like so:

```
[develop]
script_dir=$base/lib/py_pubsub
[install]
install_scripts=$base/lib/py_pubsub
```

This is simply telling setuptools to put your executables in lib, because ros2 run will look for them there.

7.4 A Subscriber

The Subscriber Code

Return to ros2_ws/src/py_pubsub/py_pubsub to create the next node. Create a subscriber node named subscriber_member_function.py Then enter the following code in it.

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

class MinimalSubscriber(Node):
    def __init__(self):
        super().__init__('minimal_subscriber')
        self.subscription = self.create_subscription(
            String,
            'topic',
            self.listener_callback,
            10)
        self.subscription # prevent unused variable warning

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


def main(args=None):
    rclpy.init(args=args)
    minimal_subscriber = MinimalSubscriber()
    rclpy.spin(minimal_subscriber)
    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_subscriber.destroy_node()
    rclpy.shutdown()

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

Add an Entry point

Reopen setup.py and add the entry point for the subscriber node below the publisher's entry point. The entry_points field should now look like this:

```
         entry_points={
        'console_scripts': [
                'talker = py_pubsub.publisher_member_function:main',
                'listener = py_pubsub.subscriber_member_function:main'
        ],
```

Again, be careful of the python syntax.

BUILD & RUN

In the root of your workspace, ros2_ws, build your new package:

```
Terminal                                                    - □ X
colcon build --packages-select py_pubsub
```

Open a new terminal, navigate to ros2_ws, and source the setup files (if not sourced in the `.bashrc` file:

```
Terminal                                                    - □ X
source install/setup.bash
```

Now run the talker node. The terminal should start publishing info messages every 0.5 seconds, like so:

```
Terminal                                                    - □ X
ros2 run py_pubsub talker
[INFO] [minimal_publisher]: Publishing: "Hello World: 0"
[INFO] [minimal_publisher]: Publishing: "Hello World: 1"
```

Open another terminal, source the setup files from inside ros2_ws again, and then start the listener node. The listener will start printing messages to the console, starting at whatever message count the publisher is on at that time:

```
Terminal                                                    - □ X
ros2 run py_pubsub listener
[INFO] [minimal_subscriber]: I heard: "Hello World: 10"
```

Enter Ctrl+C in each terminal to stop the nodes from spinning.

Code Explanation:

- Coming to the code the first lines of code after the comments import rclpy so its Node class can be used. The next statement imports the built-in string message type that the node uses to structure the data that it passes on the topic.
- These lines represent the node's dependencies. Recall that dependencies have to be added to package.xml, which you'll do in the next section. Next, the MinimalPublisher class is created, which inherits from (or is a subclass of) Node.
- Following is the definition of the class's constructor. super().__init__ calls the Node class's constructor and gives it your node name, in this case minimal_publisher. create_publisher declares that the node publishes messages of type String (imported from the std_msgs.msg module), over a topic named topic, and that the "queue size" is 10. Queue size is a required QoS (quality of service) setting that limits the amount of queued messages if a subscriber is not receiving them fast enough.
- Next, a timer is created with a callback to execute every 0.5 seconds. self.i is a counter used in the callback. timer_callback creates a message with the counter value appended, and publishes it to the console with get_logger().info.

- The subscriber node's code is nearly identical to the publisher's. The constructor creates a subscriber with the same arguments as the publisher. Recall from the topics tutorial that the topic name and message type used by the publisher and subscriber must match to allow them to communicate. The subscriber's constructor and callback don't include any timer definition, because it doesn't need one. Its callback gets called as soon as it receives a message.

- The callback definition simply prints an info message to the console, along with the data it received. Recall that the publisher defines msg.data = 'Hello World: %d' % self.i. The main definition is almost exactly the same, replacing the creation and spinning of the publisher with the subscriber.Since this node has the same dependencies as the publisher, there's nothing new to add to package.xml. The setup.cfg file can also remain untouched.

7.5 Launch Files

As you have seen, every node is its own Python file and to launch that node, you need to summon it from the terminal. When you have to launch several nodes at once, it becomes cumbersome to open different terminals and launch one node every terminal. That's where Launch files come into picture. You can launch all the nodes in one go by running a single launch file.
Launch files must be kept in a folder called `launch` in the toplevel folder of the package, i.e, `<WORKSPACE>/src/<PACKAGE>/launch`. Create a launch directory and create a python file `py_pubsub.launch.py` (The template to follow is `<PACKAGE>.launch.py`).
Launch files my be a xml, or yaml (ya-mel) or a python file. A Python launch files gives much more flexibility and ease compared to the other two and therefore we shall be using python files only.

```
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    ld = LaunchDescription()

    talker = Node(
        package='py_pubsub',
        executable='talker'
    )

    listener = Node(
        package='py_pubsub',
        executable='listener'
    )

    ld.add_action(talker)
    ld.add_action(listener)
    return ld
```

As you might have guessed it again, we need to make sure this folder also gets reflected in the install directory of the workspace, so that the launch files also get sourced. For this, update the `data_files` argument in the `setup.py`. The last line is the newly added.

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

All that is left now is to build the package. As we are using python files everywhere, it is always a good idea to pass the —-symlink–install so that new changes made can be reflected directly without needing to re-build the package.

ATTENTION!

It won't work if you don't save it (sigh). A common source of confusion is not saving the files after changes. In the VS Code settings, make sure the files are auto-saved when you switch tabs. This setting is off by default and causes confusion to many beginners.

Note that we are launching two nodes in the same terminal, so we will see everything being dumped on the same screen. Now, you figure out how to launch the two nodes in two separate tabs.

```
Terminal                                                    - □ X
~ros2_ws/ $ colcon build symlink-install --packages-select py_pubsub Starting >>
py_pubsub
Finished << py_pubsub [0.67s]
~ros2_ws/ $ source install/setup.bash
~ros2_ws/ $ ros2 launch py_pubsub py_pubsub.launch.py
talker-1
[INFO] [minimal_publisher]: Publishing: "Hello World : 0"
listener-2
[INFO] [minimal_subscriber]: I Heard: "Hello World : 0"
talker-1
[INFO] [minimal_publisher]: Publishing: "Hello World : 1"
listener-2
[INFO] [minimal_subscriber]: I Heard: "Hello World : 1"
talker-1
[INFO] [minimal_publisher]: Publishing: "Hello World : 2"
listener-2
[INFO] [minimal_subscriber]: I Heard: "Hello World : 2"
```

8 Some Useful Tools

8.1 rqt

`rqt` is a Qt-based GUI framework in ROS 2 that hosts various plugins for visualizing and interacting with the ROS 2 system. It is useful for debugging, monitoring, and introspection of topics, nodes, parameters, and more.

Commonly Used `rqt` Plugins in ROS 2

Plugin Name	Description
`rqt_graph`	Visualizes the computation graph (nodes and topic connections).
`rqt_console`	Displays log messages aggregated from nodes.
`rqt_logger_level`	Adjusts logger verbosity levels at runtime.
`rqt_plot`	Plots numeric topic data over time.
`rqt_image_view`	Visualizes image data from topics (e.g., camera feed).
`rqt_reconfigure`	For dynamic parameter tuning (less commonly used in ROS 2).
`rqt_tf_tree`	Displays the TF2 frame hierarchy.

Installing `rqt` in ROS 2

Use the following command to install `rqt`:

```
Terminal - □ X
sudo apt install ros-humble-rqt
```

Running `rqt`

Launch the GUI with:

```
Terminal - □ X
rqt
```

You can also directly run specific plugins:

```
Terminal - □ X
rqt_graph
rqt_plot
```

8.2 ros2doctor

When your ROS 2 setup is not working as expected, you can check it with the `ros2 doctor` tool.

`ros2 doctor` checks many parts of your ROS 2 system, such as:

- Platform and OS
- ROS 2 version
- Network settings
- Environment variables
- Running nodes and topics

It gives warnings and explanations if something might be wrong.

To check your general ROS 2 setup run the command:

```
Terminal                                                    - □ X
   ros2 doctor
```

This will check all parts of your setup and show any warnings or errors.

If everything is fine, you'll see:

```
Terminal                                                    - □ X
      All <n> checks passed
```

It's common to get a few warnings. Warnings do not mean your system is broken, they just suggest that something may not be ideal.

Example warning:

```
Terminal                                                    - □ X
UserWarning: Distribution <distro> is not fully supported or tested.
```

Even if there are warnings, the message `All <n> checks passed` will appear unless there's an actual error.

If there is an error, the output might look like:

```
Terminal                                                    - □ X
      1/3 checks failed
      Failed modules: network
```

Errors usually mean something important is missing and should be fixed.

You can also check a live ROS 2 system to find communication problems.

Example:

1. In one terminal:

```
Terminal                                                    - □ X
ros2 run turtlesim turtlesim_node
```

2. In a second terminal:

```
Terminal                                                    - □ X
ros2 run turtlesim turtle_teleop_key
```

3. In a third terminal, run:

```
Terminal                                                    - □ X
ros2 doctor
```

You may see new warnings like:

```
Terminal                                                    - □ X
UserWarning: Publisher without subscriber detected on /turtle1/color_sensor.
UserWarning: Publisher without subscriber detected on /turtle1/pose.
```

This means those topics are being published but no one is subscribed to them.
To remove these warnings, run:

```
Terminal                                                    - □ X
    ros2 topic echo /turtle1/color_sensor
    ros2 topic echo /turtle1/pose
```

Now if you run `ros2 doctor` again, those warnings will be gone.
If you close the turtlesim window or stop the teleop node and then run `ros2 doctor`, you'll see more warnings about missing publishers or subscribers.
In large systems, `ros2 doctor` is very helpful for finding why nodes aren't talking to each other.

You can get more details by using:

```
Terminal                                                    - □ X
ros2 doctor --report
```

This gives you detailed information in five categories:

- Network Configuration
- Platform Information
- RMW Middleware
- ROS 2 Information

- Topic List

Keep in mind: `ros2 doctor` is not a code debugger. It only checks the system setup and communication state.

8.3 Commands to Remember

Here's a cheat sheet of important ros2 commands.

Command	Function
`ros2 doctor`	Diagnoses the ROS 2 environment setup.
`ros2 node list`	Lists all currently running nodes.
`ros2 node info <node>`	Displays information about a specific node.
`ros2 topic list`	Lists all available topics.
`ros2 topic echo <topic>`	Displays messages being published on a topic.
`ros2 topic info <topic>`	Shows information about a topic.
`ros2 topic pub <topic> <type>`	Publishes a message to a topic.
`ros2 topic hz <topic>`	Measures the publishing rate (Hz) of a topic.
`ros2 topic type <topic>`	Shows the message type of a topic.
`ros2 service list`	Lists all available services.
`ros2 service type <service>`	Displays the type of a service.
`ros2 service call <service> <type>`	Calls a service with a request.
`ros2 param list`	Lists all parameters of all nodes.
`ros2 param get <node> <param>`	Retrieves the value of a parameter.
`ros2 param set <node> <param> <value>`	Sets the value of a parameter.
`ros2 action list`	Lists all available actions.
`ros2 action send_goal <action> <type>`	Sends a goal to an action server.
`ros2 launch <package> <launch_file.py>`	Launches a specified launch file.
`ros2 run <package> <executable>`	Runs an executable from a package.
`ros2 pkg list`	Lists all installed packages.
`ros2 pkg info <package>`	Shows information about a specific package.
`ros2 interface show <type>`	Displays the definition of a message or service type.
`ros2 msg list`	Lists all available message types.
`ros2 srv list`	Lists all available service types.
`rqt`	Starts the RQT graphical interface.
`rqt_graph`	Visualizes the computation graph (nodes and topic connections).
`rqt_console`	Shows log messages being published by nodes.
`rqt_plot`	Plots numeric data published on topics.
`ros2 -help`	Lists all ROS 2 subcommands.

NOTE

You can always use –help or -h to see the list of all available commands.

9 DIY Tasks

Here is the list of tasks that will enhance and test your understanding of the learnings in this module

1. **Emulating a Temperature Sensor**
   You need to create one node simulating a temperature sensor and two nodes which serve as the temperature monitors.

   - The Sensor gives readings in Kelvin Scale
   - Monitor-1 needs to display reading in Celsius Scale
   - Monitor-2 needs to display reading in Farenheit Scale

   Create a launch file to launch all three nodes. The Sensor generates temperature readings sampled out of a gaussian distribution whose mean and variance are specified at the time of launch through command line arguments.

2. **Calculator**
   Develop a ROS 2 service server node that serves as a calculator (involving addition, multiplication, subtraction and division) with integers or float numbers as input parameters received from a client. Implement a client node that sends requests to the service server node and displays the result returned by the server.
   The input will be a string where the numbers and symbols are separated by a space. This expression needs to be evaluated keeping in mind the BODMAS rules.
   The client needs to process this input and send discrete requests **only** to the server. The server should not do any computation apart from addition, subtraction, multiplication and division.
   Hint: Generate the equivalent postfix expressions and evaluate using a sutiable data structure like a stack
   Sample Input-Outputs:

```
Terminal                                                    - □ X
$ ros2 run calculator client
Enter Expression :
1+6
Result: 7.000
$ ros2 run calculator client
Enter Expression :
1-2/3
Result: 0.333
$ ros2 run calculator client
Enter Expression :
2+2-2/2
Result: 3.000
```

Deliverables: A (preferably LaTeX) document containing

1. The ROS Architecture in the tasks, with the rqt_graph
2. Your approach of solving the expression in the second task
3. The screenshots of your outputs two tasks
4. Code of all your nodes - in proper formatting

Please submit a single pdf file only.