{Learn, Create, Innovate};

Robot Operating System - ROS

Introduction

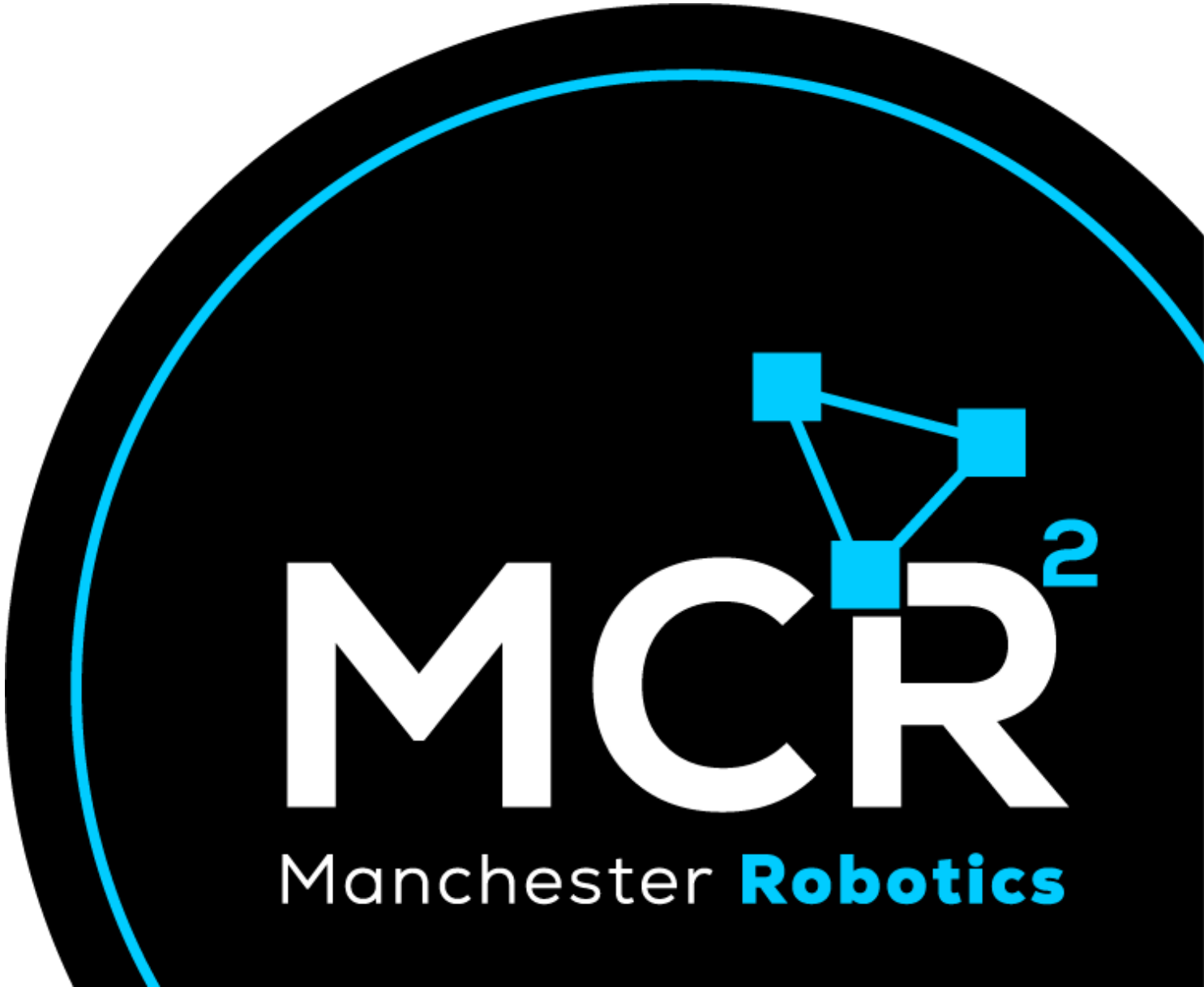
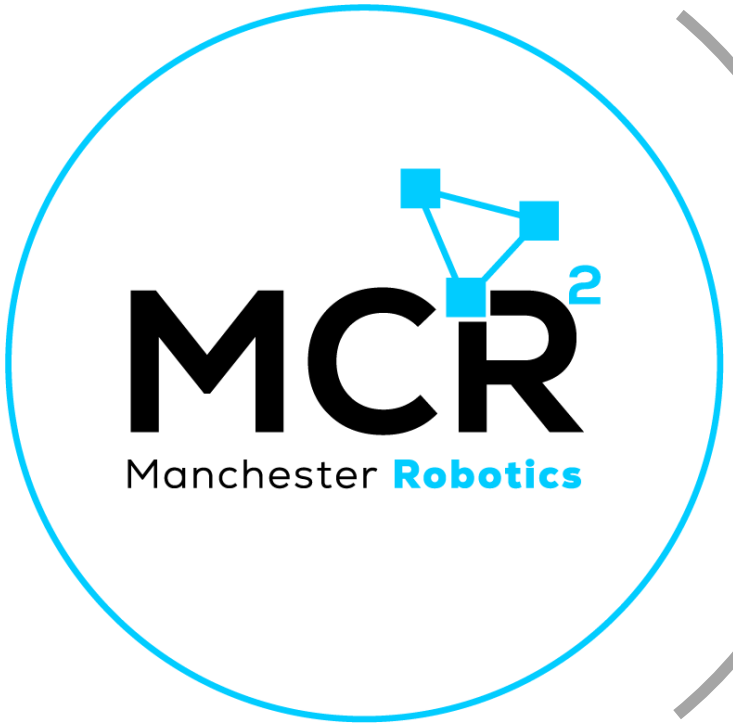


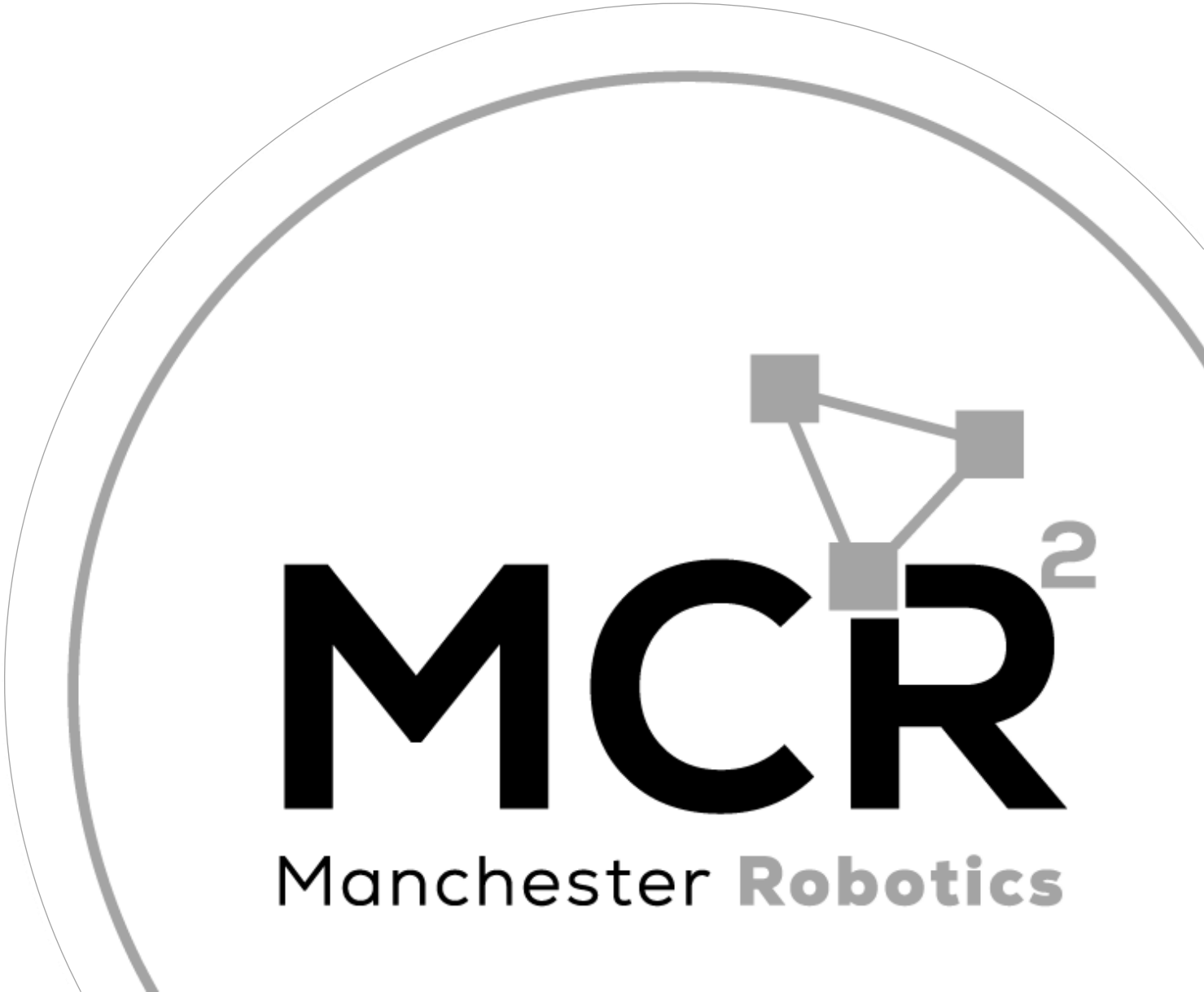
Table of contents



Robot Operating System - ROS

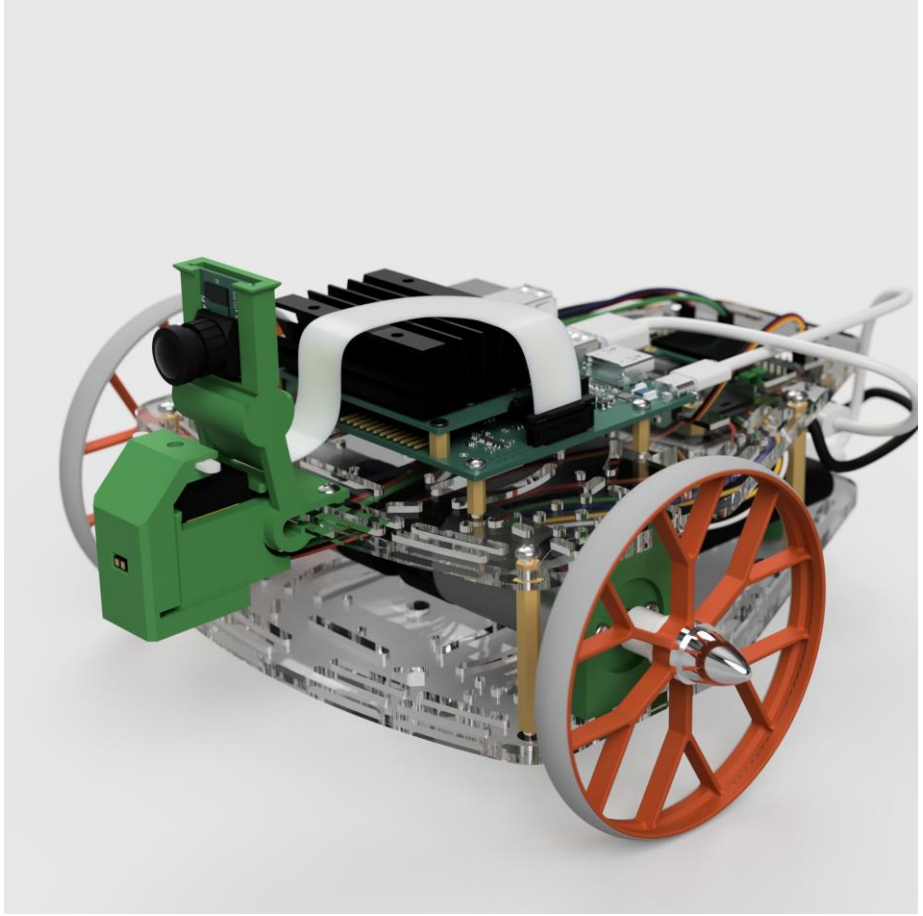
What is ROS?

{Learn, Create, Innovate};

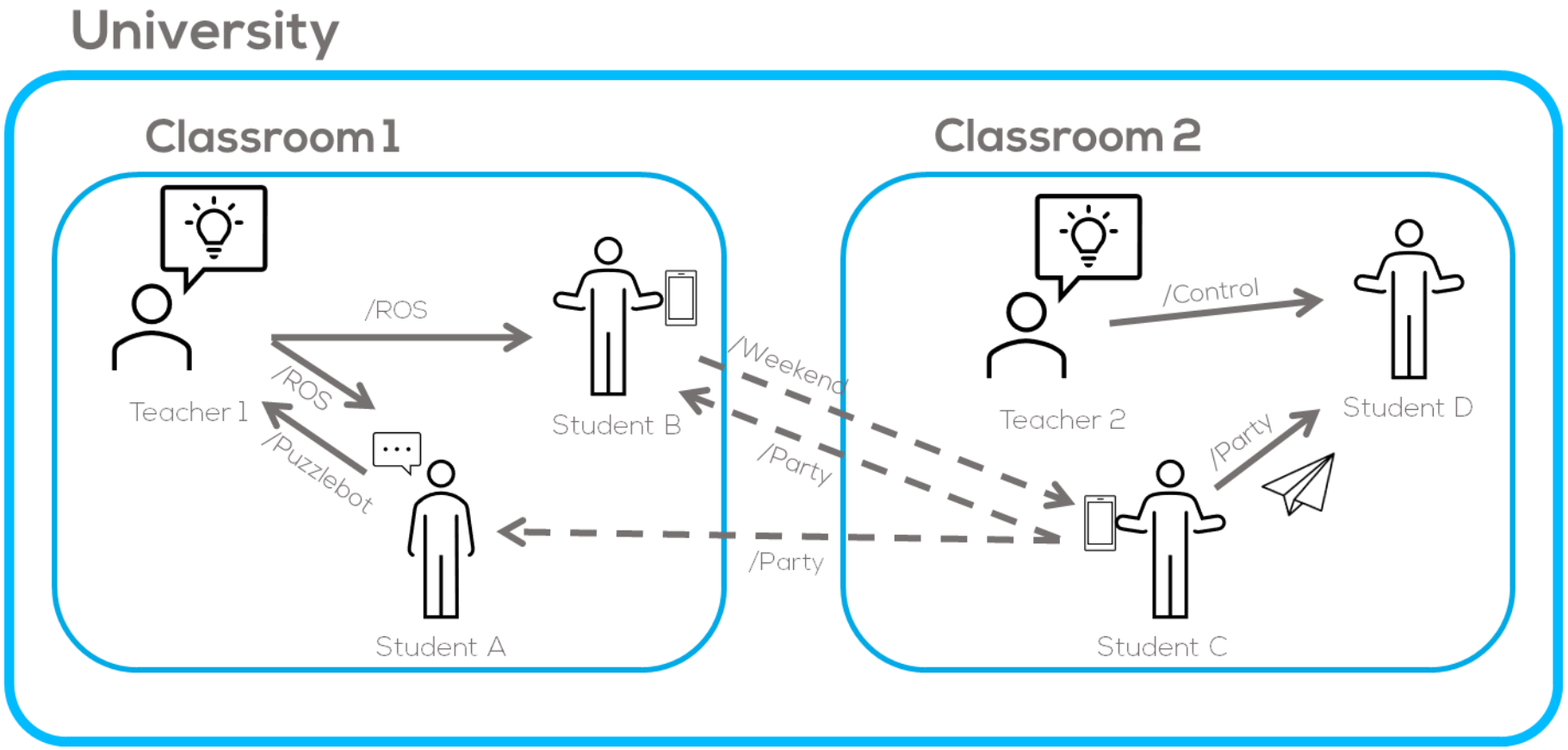


Introduction

- Robotics is becoming increasingly complex.
- The increasing number of sensors, actuators, and other hardware components used in robotics.
- Complex algorithms used in robotics nowadays, from robust control to AI.
- There is a need for a flexible, scalable, and reliable communication infrastructure.



Introduction

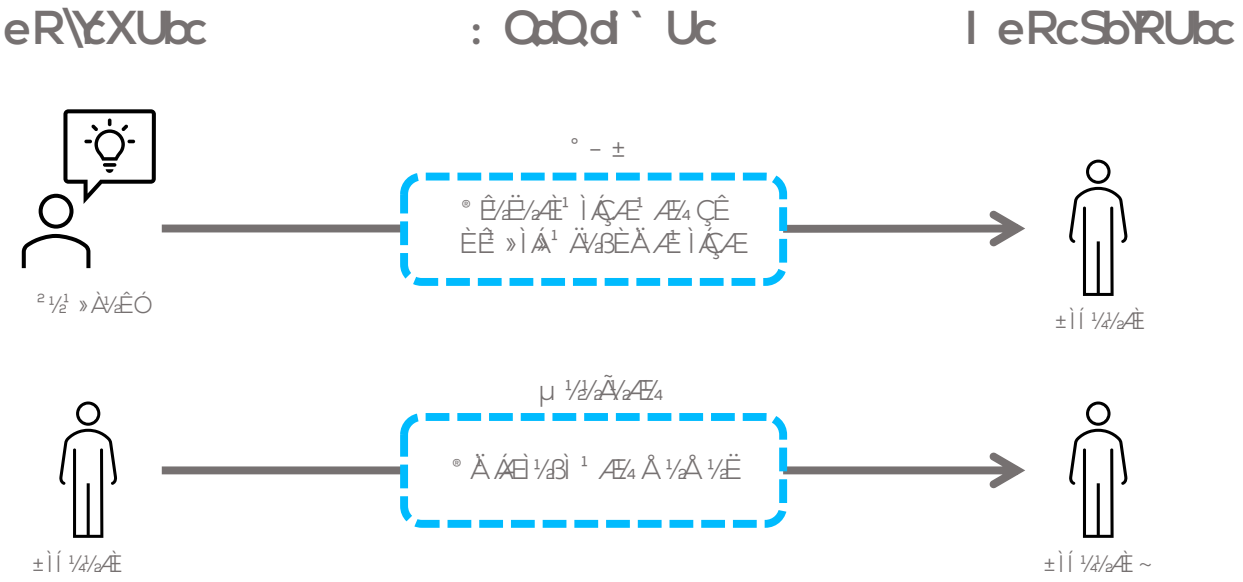


** Any resemblance to reality, is purely coincidental.

Introduction

How is the information delivered?

- The information is delivered through messages inside each topic.
- Any message or class has a certain format (is encoded), which both the teacher and the student know off and is expected.
- As an example: Between two students, it is expected some simple messages such as plain text, memes or figures.
- If the structure of the message is incorrect it would not be possible to understand it.

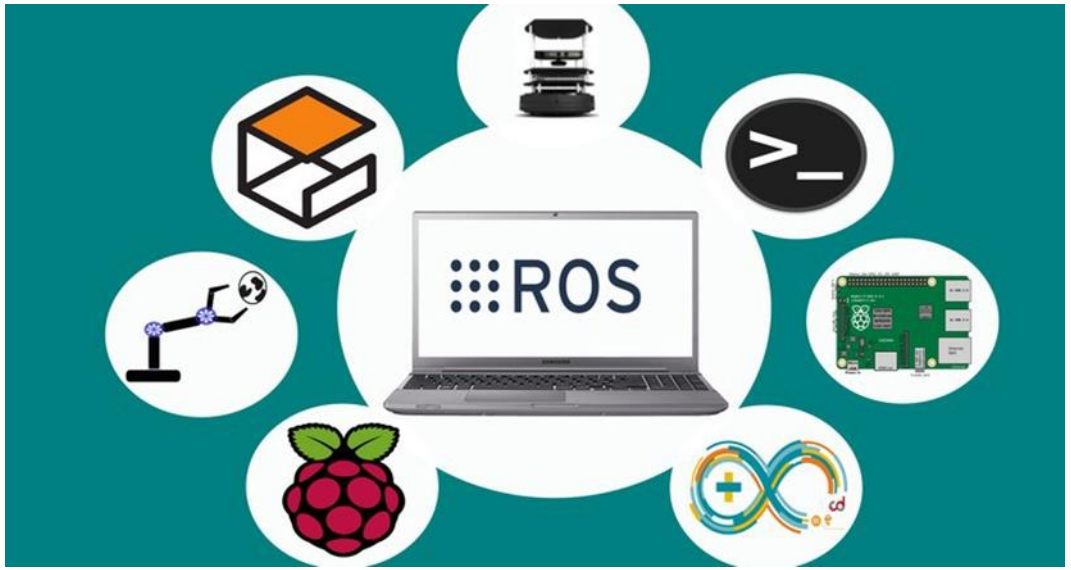


What is ROS?

What is ROS?

“ROS is a set of software libraries and tools for building robot applications. From drivers and state-of-the-art algorithms to powerful developer tools, ROS has the open-source tools you need for your next robotics project.”

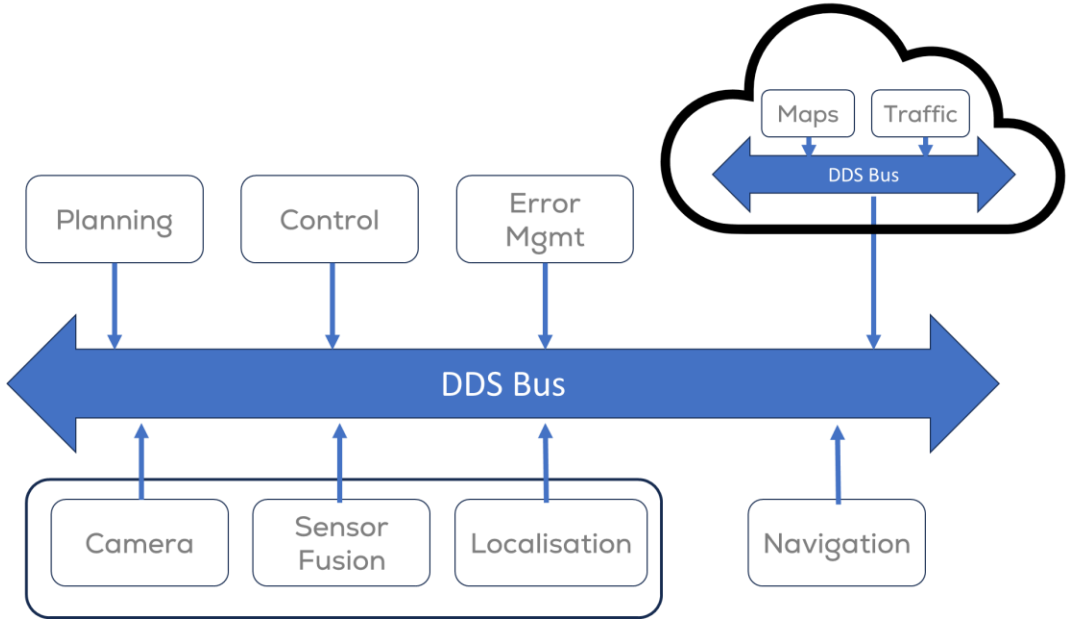
ROS is designed to be an **open, scalable, and Interoperable** framework for building robot applications.



What is ROS? Let's be more technical

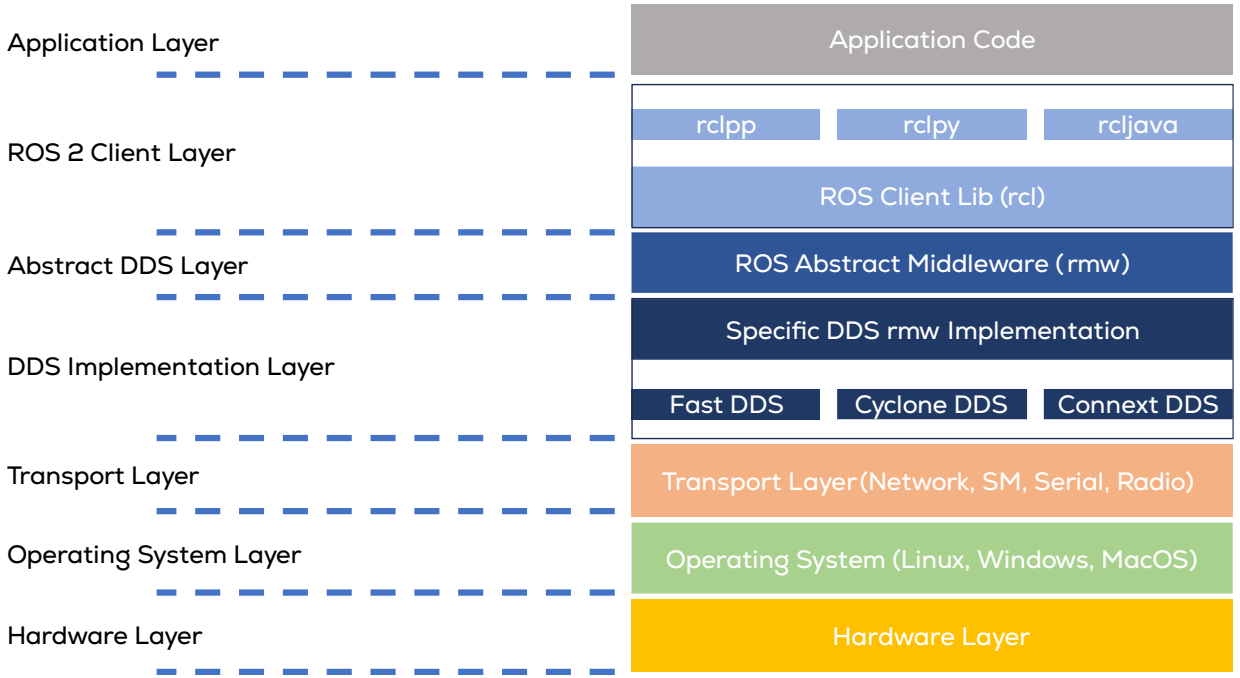
What is ROS?

- ROS 2 (Robot Operating System 2) is an open-source robotics middleware framework and tools.
  - A middleware is a software layer facilitating communication and data exchange between distributed systems.
- Utilises DDS (Data Distribution Service) for robust communication.
- Manages message passing and ensures efficient communication.
- Abstracts underlying complexities for developers.



ROS Architecture

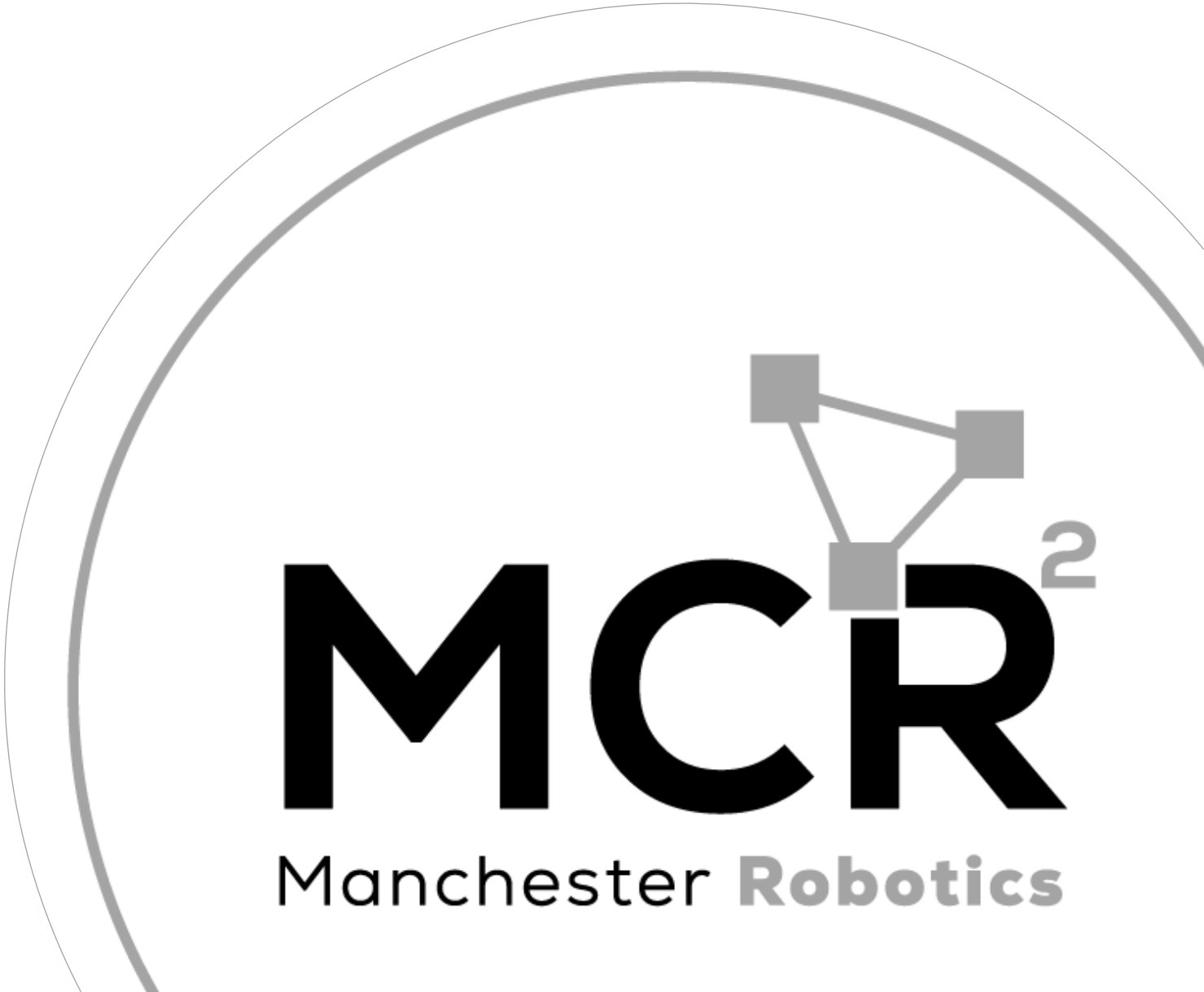
- ROS comprises of various software layers, that provide the infrastructure for building and running robot applications.
  - These layers work together to provide a high-level interface for building and running robot applications.
- DDS (Data Distribution Service) is a middleware that provides a high-performance, scalable, and secure way for nodes to exchange data and communicate with each other.
- rmw (ROS Middleware) is a middleware layer that abstracts the complexity of DDS.
- rcl (ROS Client Library) is a middleware layer that provides a high-level interface for building and running robot applications using rmw.
- Rclcpp/rclpy is a C++/Python implementation of rcl that provides a set classes and functions for building and running robot applications using rcl.



Robot Operating System - ROS

ROS Architecture

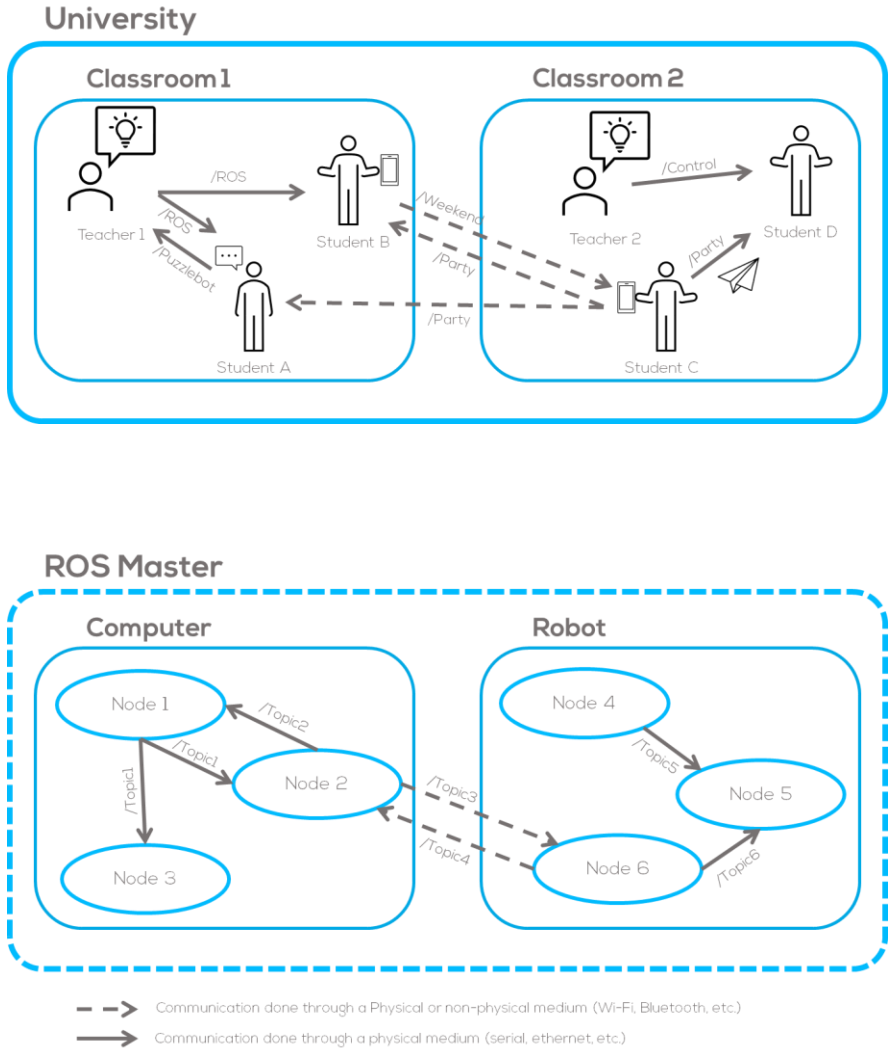
{Learn, Create, Innovate};



How ROS Works

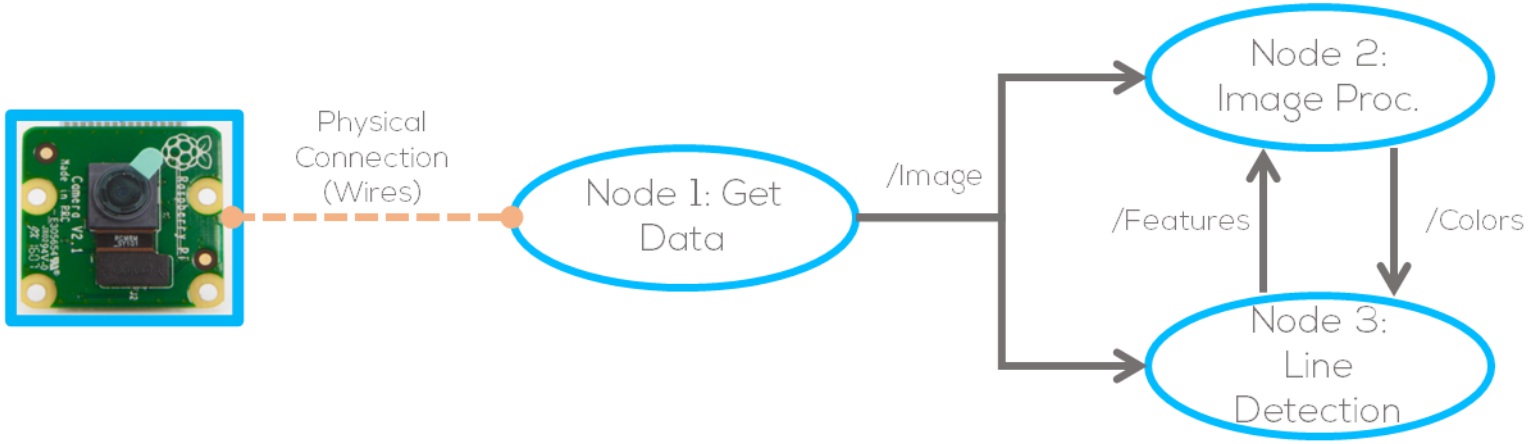
Communication Model

- ROS communication model is loosely based on the previous analogy.
- Where the persons may represent a piece of software (code) that acts as an element in the network called “Nodes”.
- Sharing messages (data) amongst these Nodes is done via topics and subscribers.
- ROS offers three different communication architectures Publisher-Subscriber, Services or Actions.



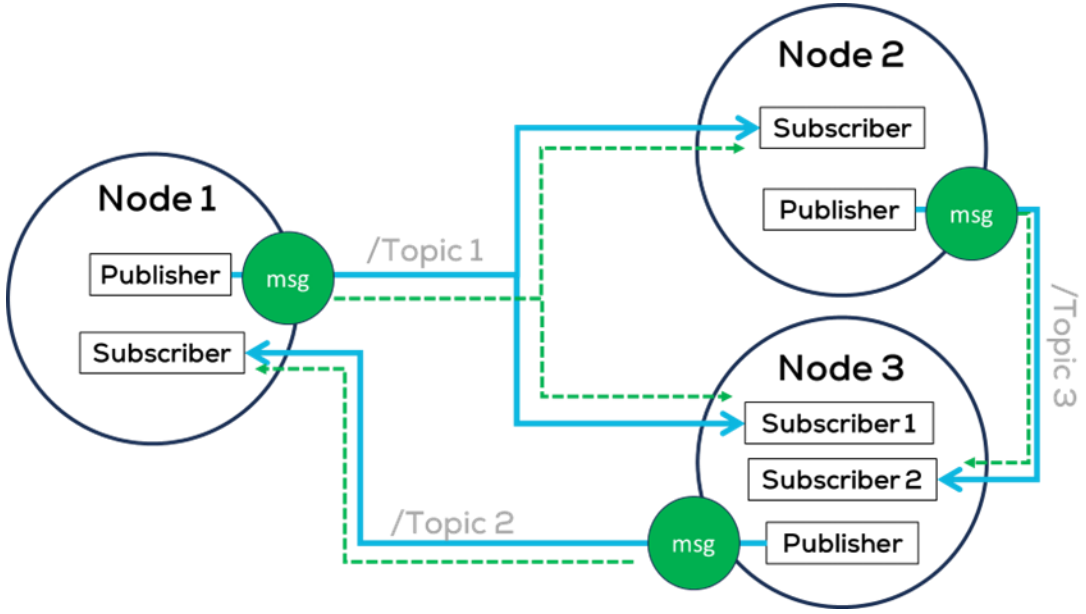
Nodes

- Fundamental ROS 2 element that serves a single, modular purpose in a robotics system
  - Wheel control
  - Camera
  - Processing
- It is a piece of software that acts as an element in the network.
- Executes part of a code and can be programmed in C++ or Python.



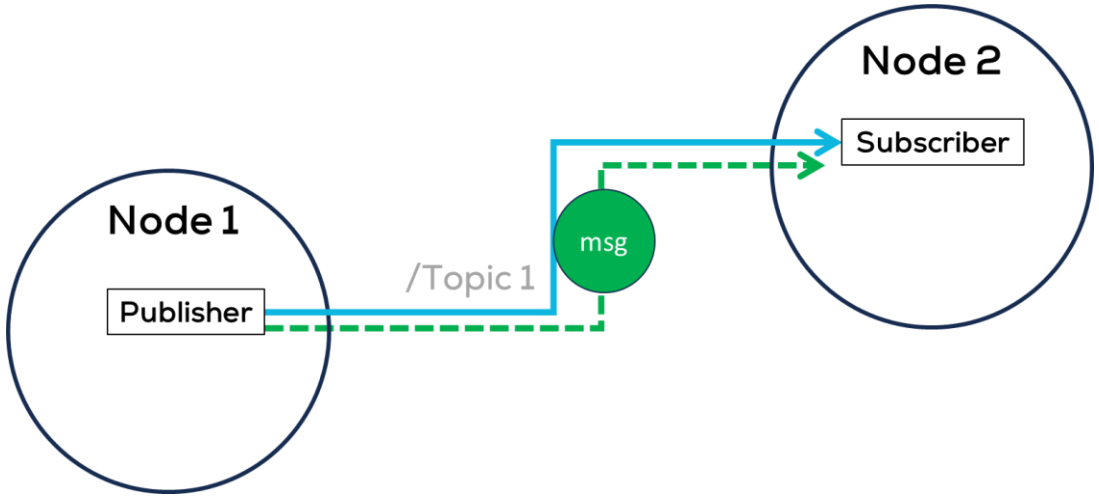
Topics

- Topics or Buses, are a vital element of the ROS graph that act as a bus for nodes to exchange messages.
- Topics are one of the main ways in which data is moved between nodes and therefore between different parts of the system.



Interface (Messages)

- Messages are the bits of data that are sent from one node to another over a topic.
- They provide a structure for the data being sent.
- Publishers and subscribers must send and receive the same type of message to communicate.
- Many types of data are supported such as integers, floating point, Boolean, etc.
- Messages can include nested structures and arrays.



Interface (Messages)

std_msgs/Float32

```
float32 data
```

geometry_msgs /Point

```
float64 x
float64 y
float64 z
```

geometry_msgs/Pose

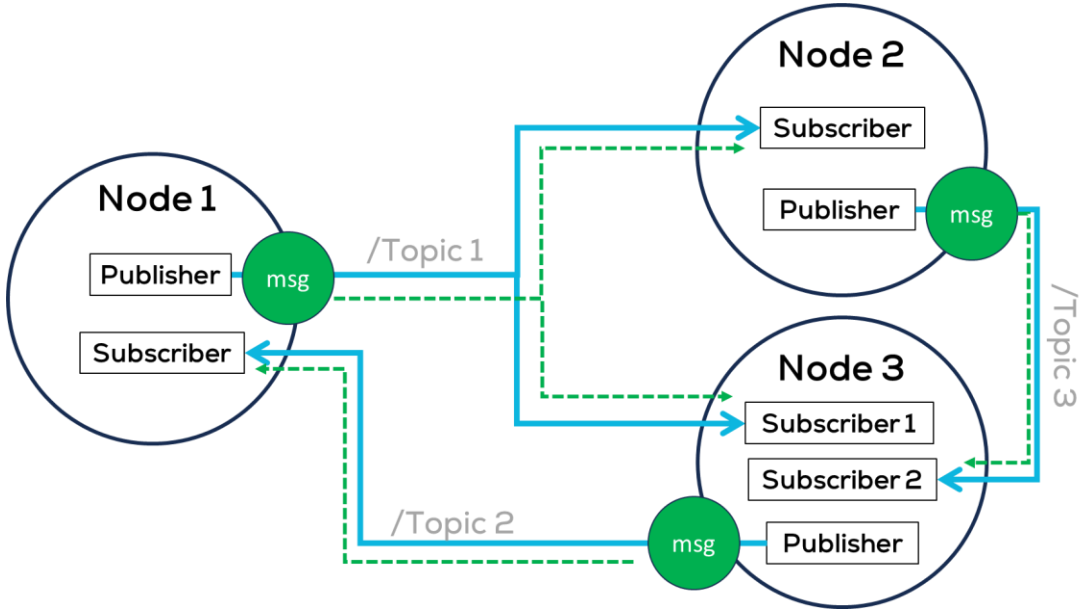
```
geometry_msgs/Point position
    float64 x
    float64 y
    float64 z
geometry_msgs/Quaternion orientation
    float64 x
    float64 y
    float64 z
    float64 w
```

Communication

Publisher and Subscriber:

- Sending a message in a certain topic is called a “publishing”.
- Receiving a message from a certain topic is called a “subscribing”.
- **Publishers** are objects created inside each node to send a message to a topic.
- **Subscribers** are objects created inside each node to “subscribe” to the topic so it can receive messages. The subscriber will then get any messages that are posted to that topic.

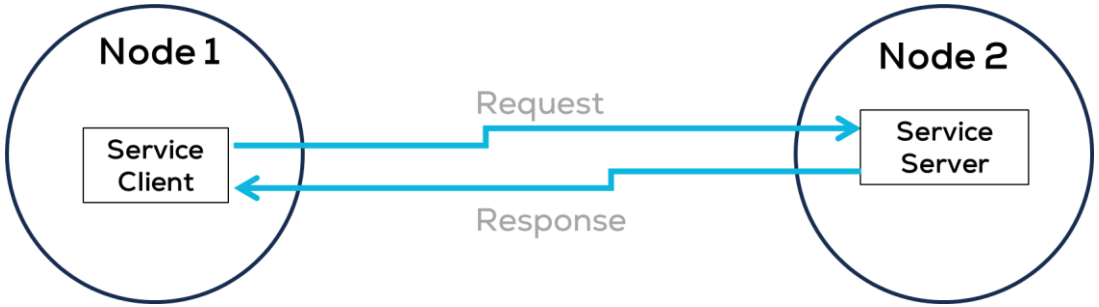
Publisher- Subscriber Architecture



How ROS Works

Services

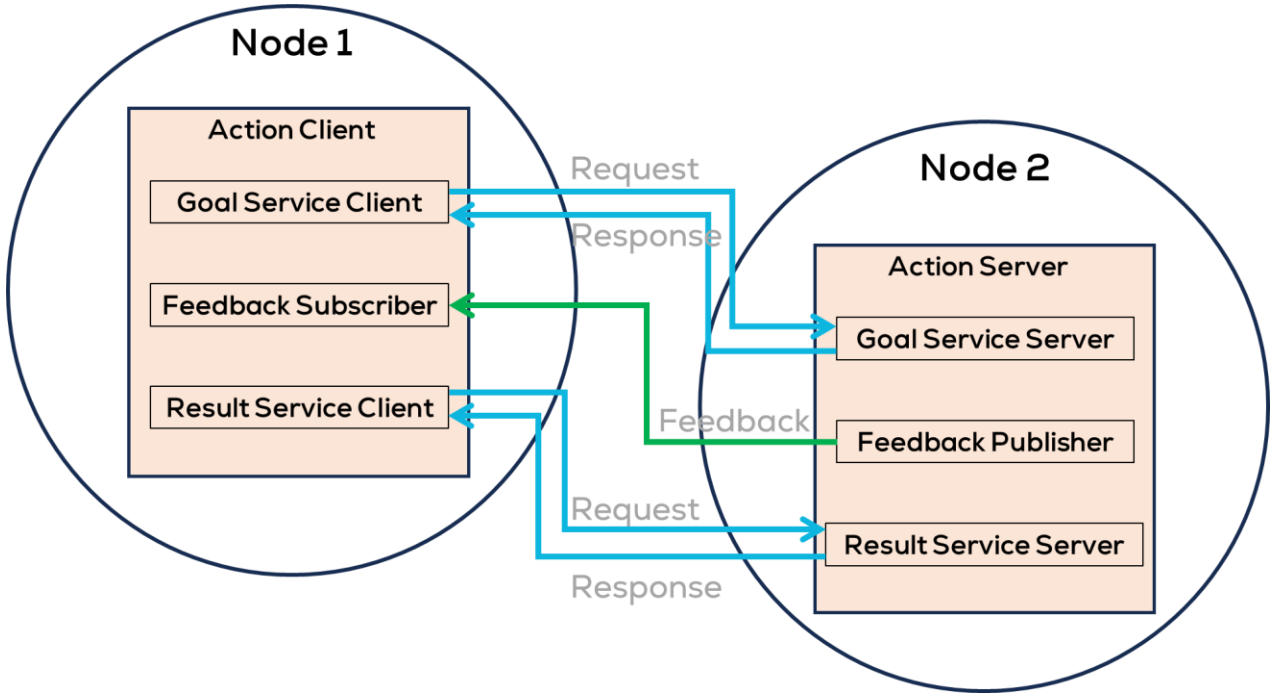
- A service is a way for nodes to request specific computation or data from other nodes and receive a response.
- Services are implemented using a request-response messaging pattern.
- A node (Node 1) request a computation or data (the client) by sending a request message to another node (the server), which performs the computation or retrieves the data and sends a response message back to the client.



How ROS Works

Actions

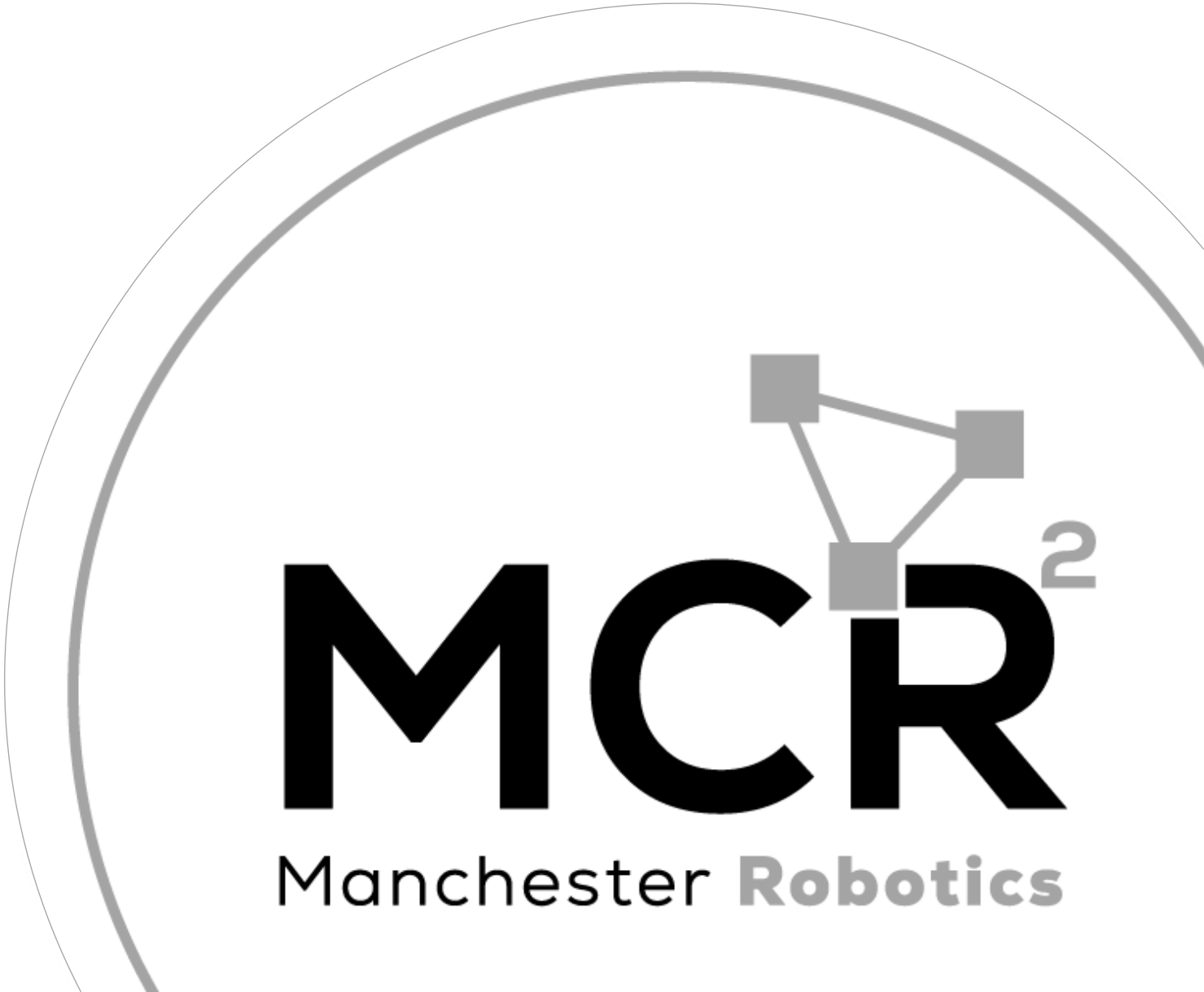
- Actions are intended for long running tasks.
- They consist of three parts: a goal, feedback, and a result.
- Actions are a combination of publishers/subscribers and services. They work as follows:
    1. **Node 1** "action client" **sends a Goal request** to Node 2 "action server".
    2. **Node 2** "action server" acknowledges the goal and **sends a response.**
    3. **Node 1 (client) sends a Result Request** to the Node 2 (server).
    4. **Node 2 returns a stream of feedback** using a publisher and Node 1 subscribes to the data feedback.
    5. **Node 2 (server) sends the response with the result** to the client.
- Actions, unlike services, can be cancelled while executing (preemtable).



Robot Operating System - ROS

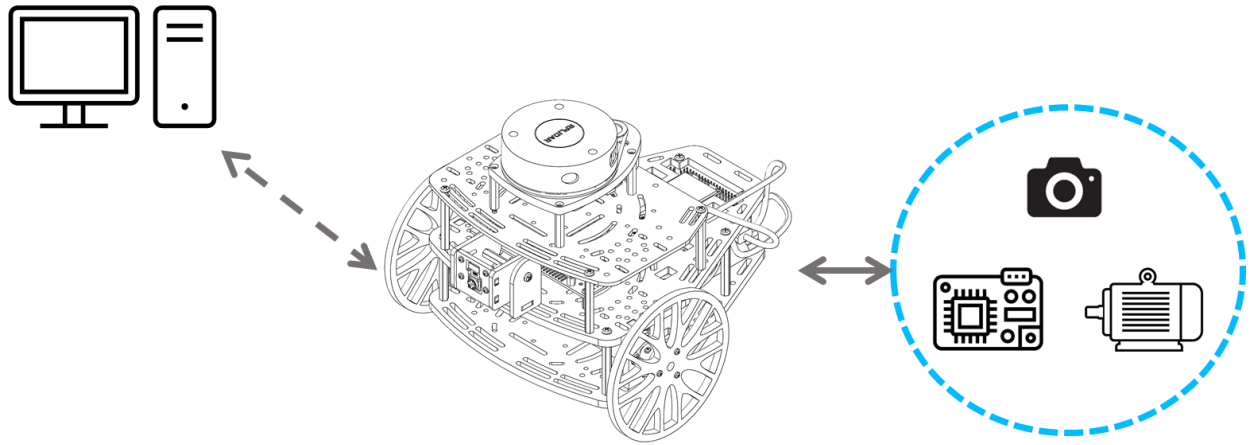
ROS Example

{Learn, Create, Innovate};



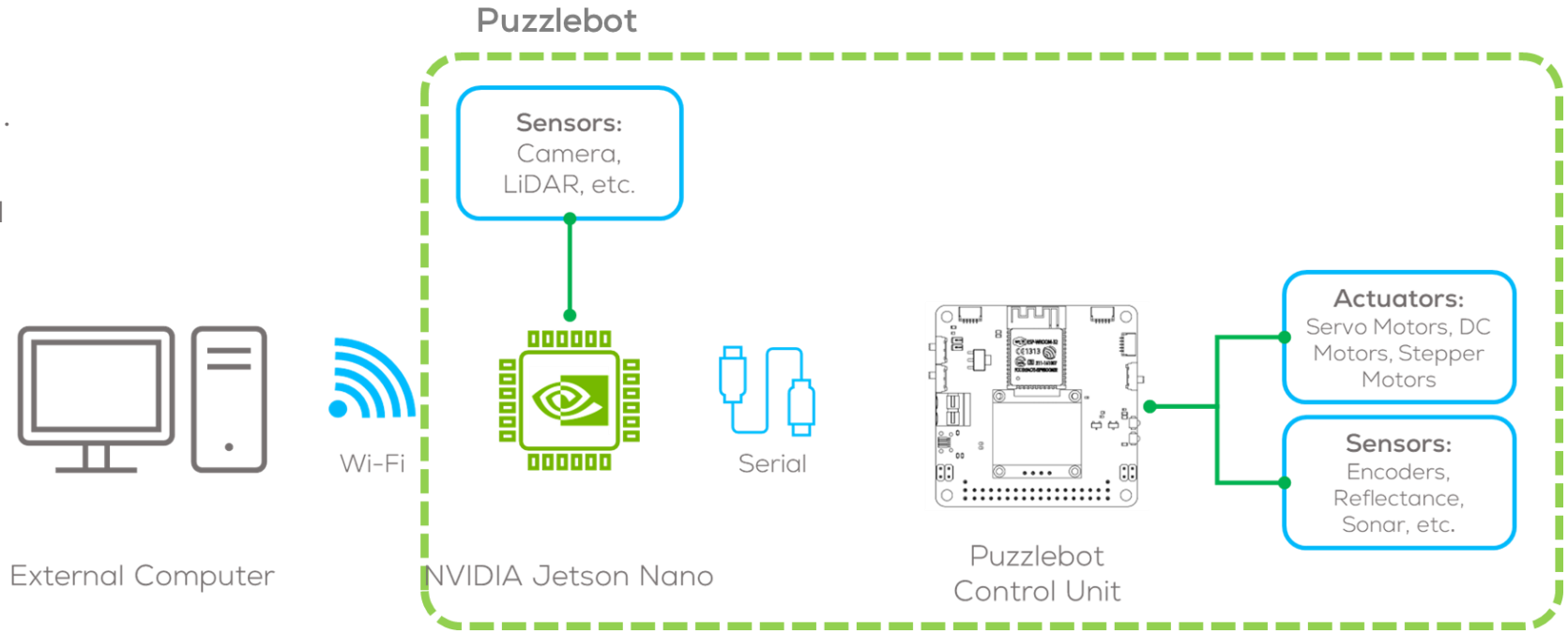
ROS Practical Example

- The system presented is a mobile robot called Puzzlebot® by Manchester Robotics.
- The robot is comprised of an on-board computing unit from NVIDIA the Jetson Nano used for high level control algorithms and a microcontroller for low level control tasks.
- The robot also contains different sensors and actuators such as motors, encoders, and cameras.

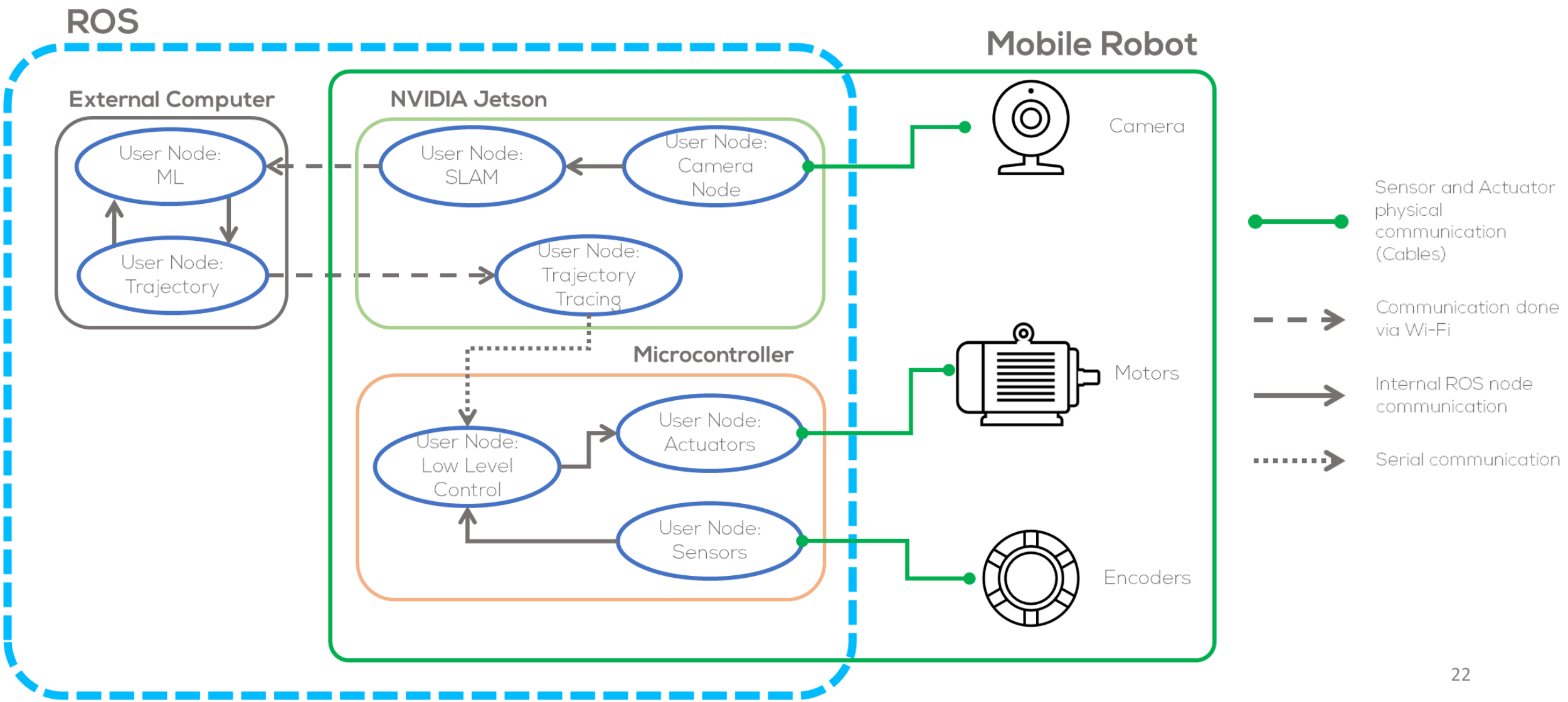


ROS Practical Example

- The robot is controlled by a computer running a ROS node and communicated to the robot via Wi-Fi.
- The on-board computer runs Ubuntu as OS and uses ROS nodes to perform different tasks and communicate with the external computer, microcontroller and the peripherals.
- The microcontroller contains several ROS nodes and provides access to the sensors and actuators.

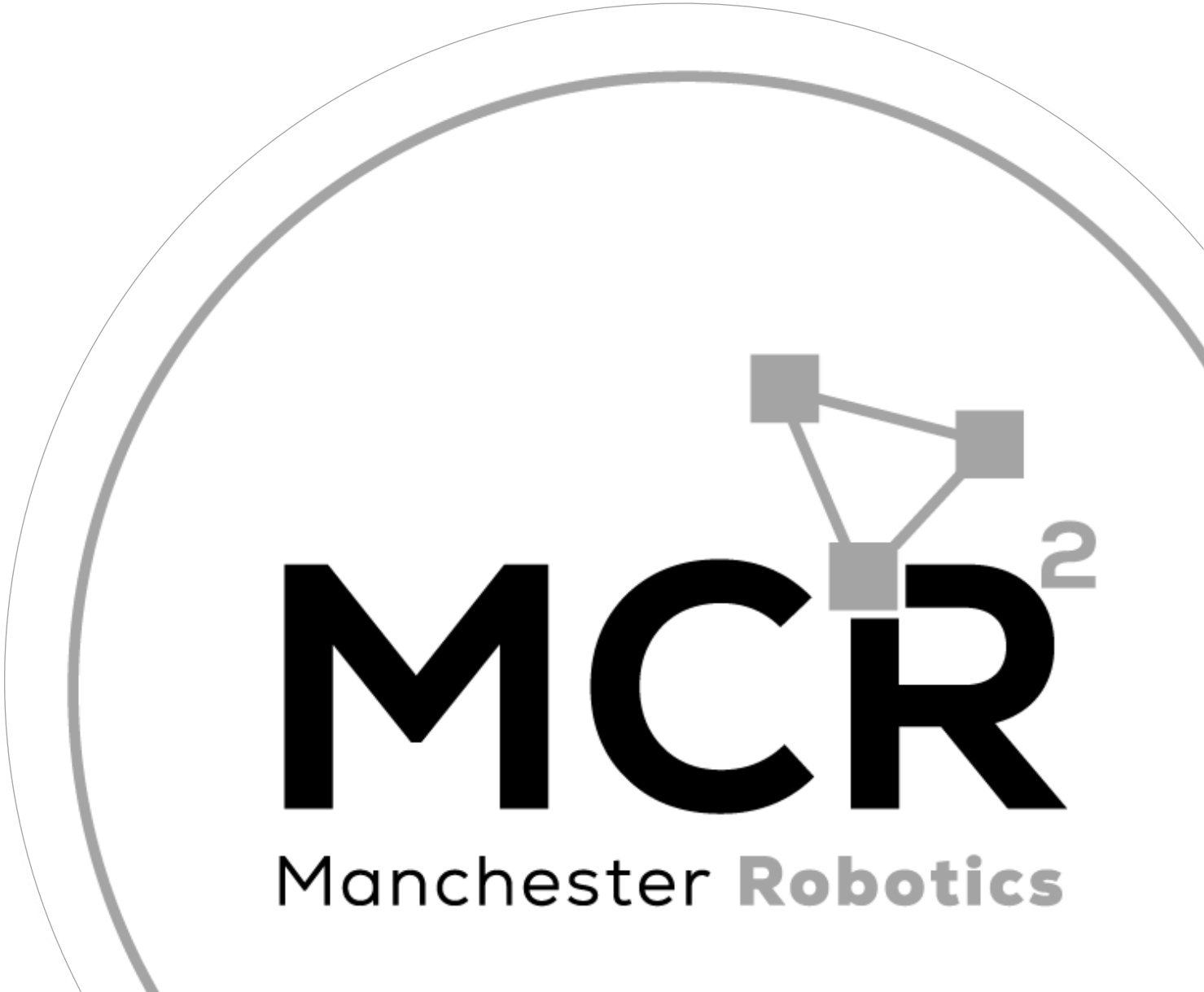


ROS Example


ROS
Mobile Robot
External Computer
User Node: ML
User Node: Trajectory
NVIDIA Jetson
User Node: SLAM
User Node: Camera Node
User Node: Trajectory Tracing
Microcontroller
User Node: Low Level Control
User Node: Actuators
User Node: Sensors
Camera
Motors
Encoders
Sensor and Actuator physical communication (Cables)
Communication done via Wi-Fi
Internal ROS node communication
Serial communication

Robot Operating System - ROS

ROS Organization

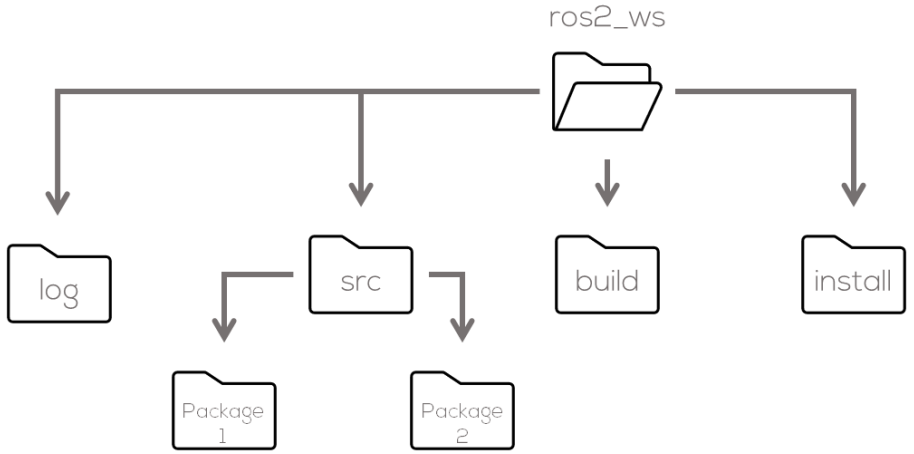


{Learn, Create, Innovate};

ROS Organization

ROS Workspace

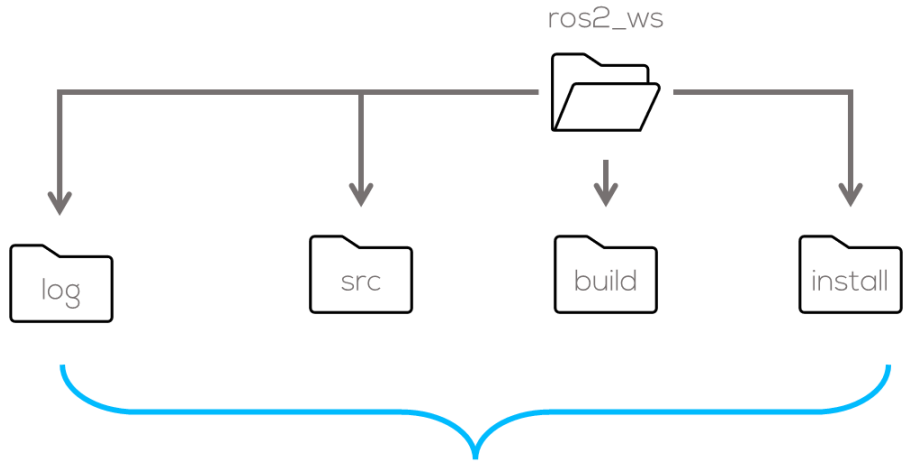
- We have seen some of the components of ROS. But how are they organized/represented in the computer?
- ROS projects are organized in workspaces, which are a collection of grouped codes called packages.
- A workspace is a set of directories (or folders) where you store related pieces of ROS code (ROS Packages).



catkin_workspace
- build
- install
- log
- src
  - Package 1
  - Package 2

ROS Organization: Workspace

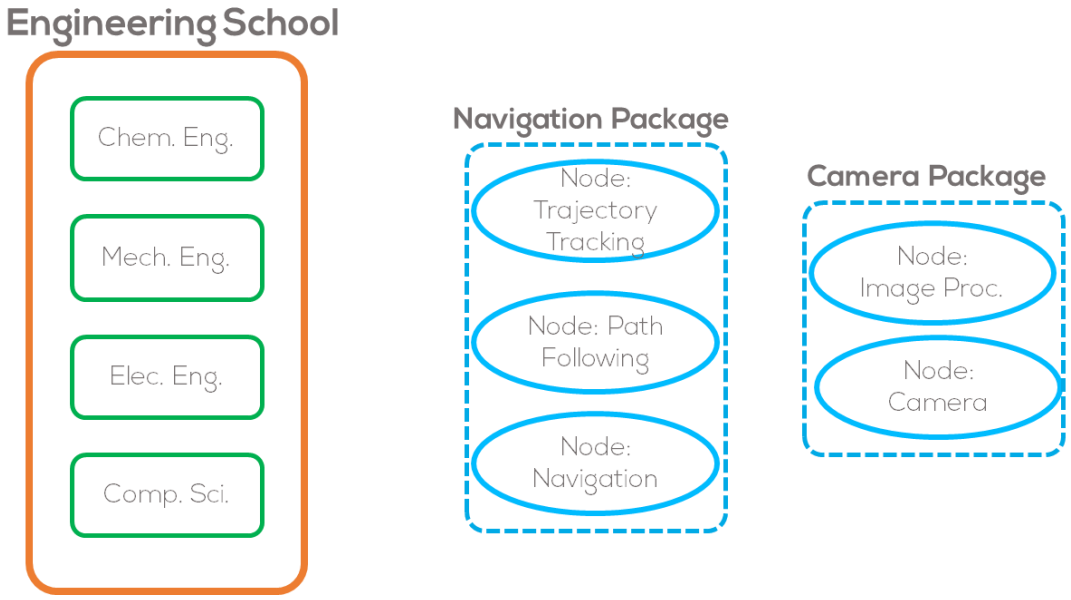
- The *build* directory will be where intermediate files are stored. For each package a subfolder will be created in which e.g. CMake is being invoked.
- The *install* directory is where each package will be installed to. By default, each package will be installed into a separate subdirectory.
- The *log* directory contains various logging information about each colcon invocation.
- The *src* subdirectory. Inside that subdirectory is where the source code of ROS packages will be located. Typically, the directory starts empty.



ROS Organization: Packages

ROS Packages

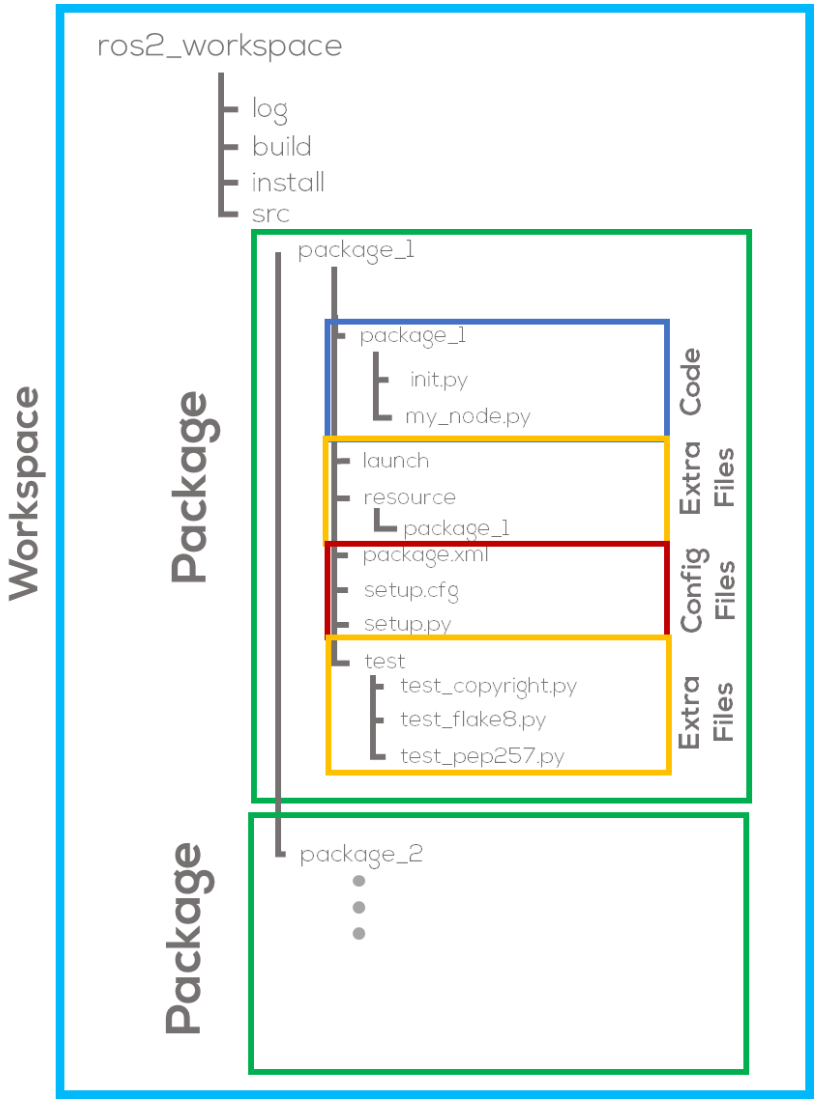
- To keep everything more organized, ROS uses the concept of packages.
- ROS Packages are a way of organizing code related between each other. They can be considered as a container for ROS code.
- The same way teachers are gathered within the engineering school nodes are gathered in ROS packages.
- Another analogy related with the university, could be the types of students. In this case, the students who pay attention can be one package and the ones who don't could be another package.



ROS Organization

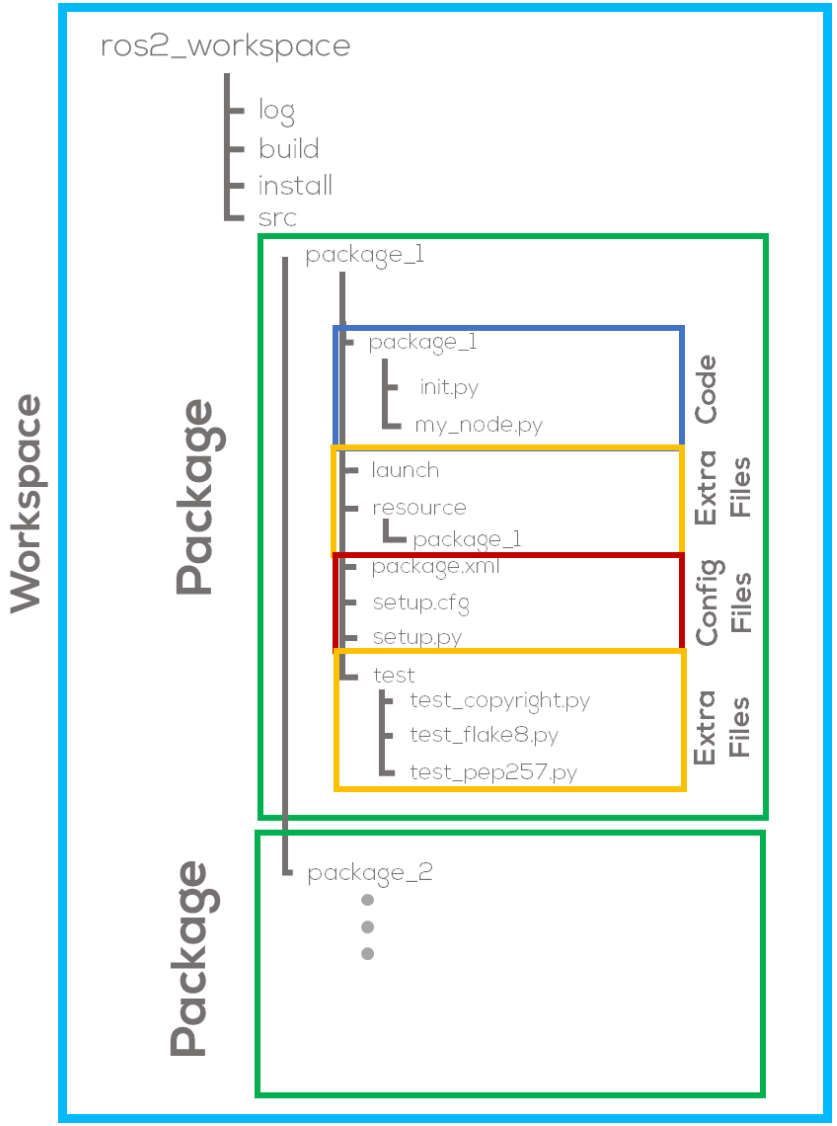
ROS Packages

- Each package in ROS requires different attributes to be compiled.
- These attributes are usually dependencies related with other packages, external libraries or custom messages, services and actions.
- ROS uses *ament* as its build system and *Colcon* as its build tool.
- CMake or Python can be used to create a package.
- Instructions for the compiler need to be allocated in CMake and Package files.



Packages in a workspace

- A single workspace can contain as many packages as wanted, each in their own folder.
- A workspace can also have packages of different build types in one workspace (CMake, Python, etc.). You cannot have nested packages.
- Best practice is to have a src folder within your workspace, and to create your packages in there. This keeps the top level of the workspace “clean”.



ROS2 CMake Package Command Line

- To create a package in C++

```
$ ros2 pkg create --build-type ament_cmake <package_name>
```

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

[WARNING]: Unknown license 'TODO: License declaration'.  This has been set in the package.xml, but no LICENS
E file has been created.
It is recommended to use one of the ament license identifiers:
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
```

- Minimum required content:
  - package.xml file containing meta information about the package
  - CMakeLists.txt file that describes how to build the code within the package

```
src/
└── test_package
    ├── CMakeLists.txt
    ├── include
    │   └── test_package
    ├── package.xml
    └── src
```

ROS2 Python Package Command Line

- To create a package with Python

```
$ ros2 pkg create --build-type ament_python <package_name>
```

```
mario@MarioPC:~/ros2_ws/src$ ros2 pkg create --build-type ament_python test_package
going to create a new package
package name: test_package
destination directory: /home/mario/ros2_ws/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['mario <mario@todo.todo>']
licenses: ['TODO: License declaration']
build type: ament_python
dependencies: []
creating folder ./test_package
creating ./test_package/package.xml
creating source folder
creating folder ./test_package/test_package
creating ./test_package/setup.py
creating ./test_package/setup.cfg
creating folder ./test_package/resource
creating ./test_package/resource/test_package
creating ./test_package/test_package/__init__.py
creating folder ./test_package/test
creating ./test_package/test/test_copyright.py
creating ./test_package/test/test_flake8.py
creating ./test_package/test/test_pep257.py

[WARNING]: Unknown license 'TODO: License declaration'.  This has been set in the package.xml, but no LICENS
E file has been created.
It is recommended to use one of the ament license identifiers:
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
```

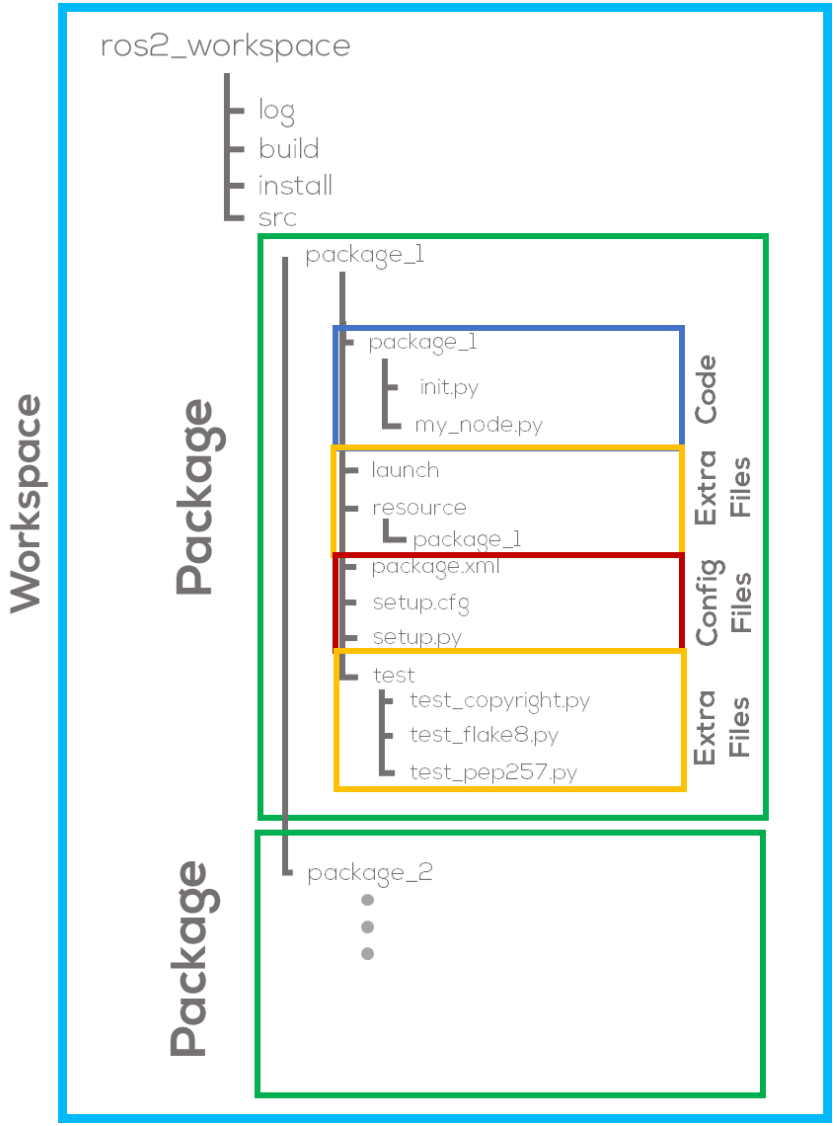
- package.xml file containing meta information about the package
- setup.py containing instructions for how to install the package
- setup.cfg is required when a package has executables, so ros2 run can find them
- /<package_name> - a directory with the same name as your package, used by ROS 2 tools to find your package, contains __init__.py

```
src/
└── test_package
    ├── package.xml
    ├── resource
    │   └── test_package
    ├── setup.cfg
    ├── setup.py
    ├── test
    │   ├── test_copyright.py
    │   ├── test_flake8.py
    │   └── test_pep257.py
    └── test_package
        └── __init__.py
```

ROS Organization

A Closer Look Into a ROS Package

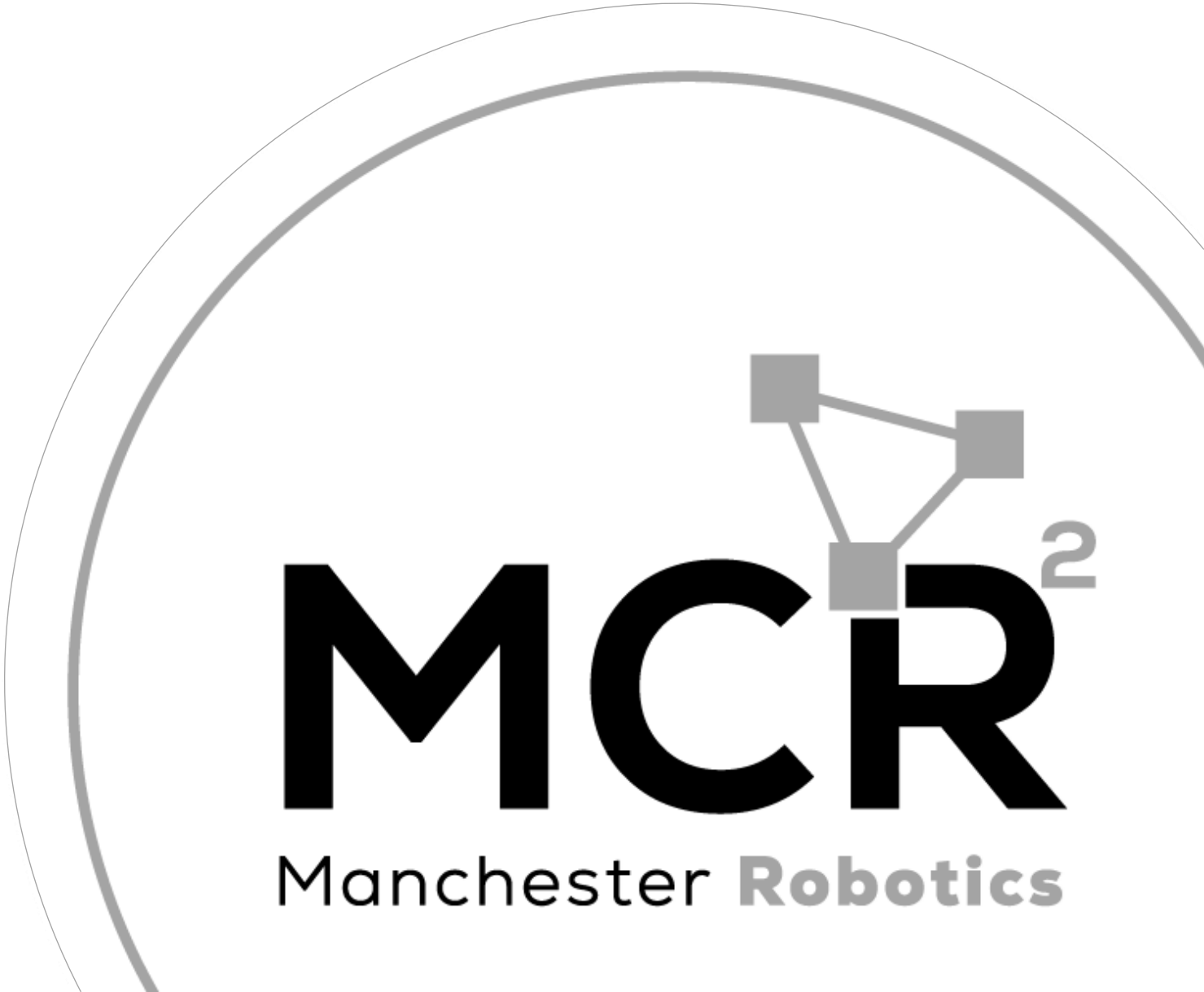
- Packages: Files that can be exported between projects
- Configuration files used to stablish code dependencies.
- Extra files and folders: They are files/folders dedicated for specific tasks. In this case the launch file allow us to execute several nodes at the same time
- Nodes : Code that we will execute inside each node.



Robot Operating System - ROS

Creating a Workspace

{Learn, Create, Innovate};



Create a workspace

Creating a workspace

- Open a terminal in Ubuntu
- Source the ROS2 underlay

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

replace $DISTRO$ for the name of your distribution e.g.:

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

- Create a directory to contain our workspace

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

- Access the created folder

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

- From the root of your workspace (ros2_ws), you can now build your packages using the command:

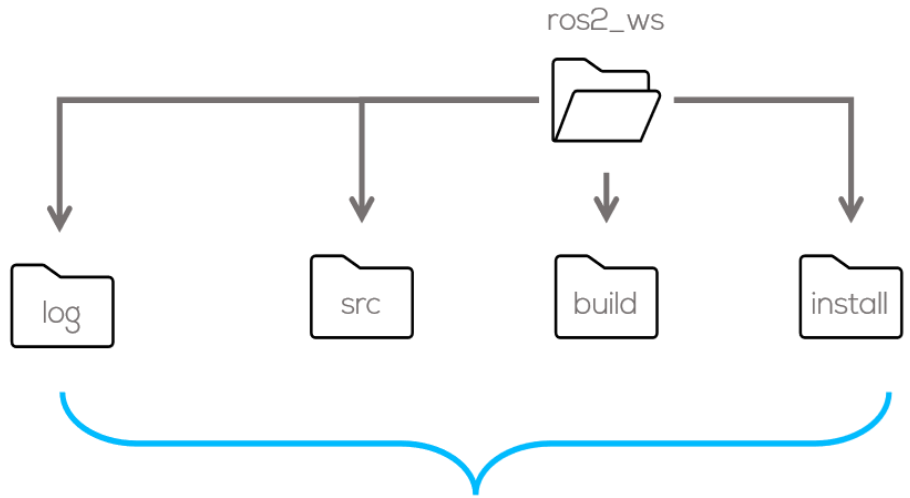
```
$ colcon build
```

- *colcon* does out of source builds. By default, it will create the *build*, *install* and *log* directories as peers of the *src* directory

Create a workspace

- Before you can use any of the installed executables or libraries, you will need to add them to your path and library paths.
- *Colcon* will have generated *bash/bat* files in the install directory to help setup the environment.
- These files will add all of the required elements to your path and library paths as well as provide any bash or shell commands exported by packages.

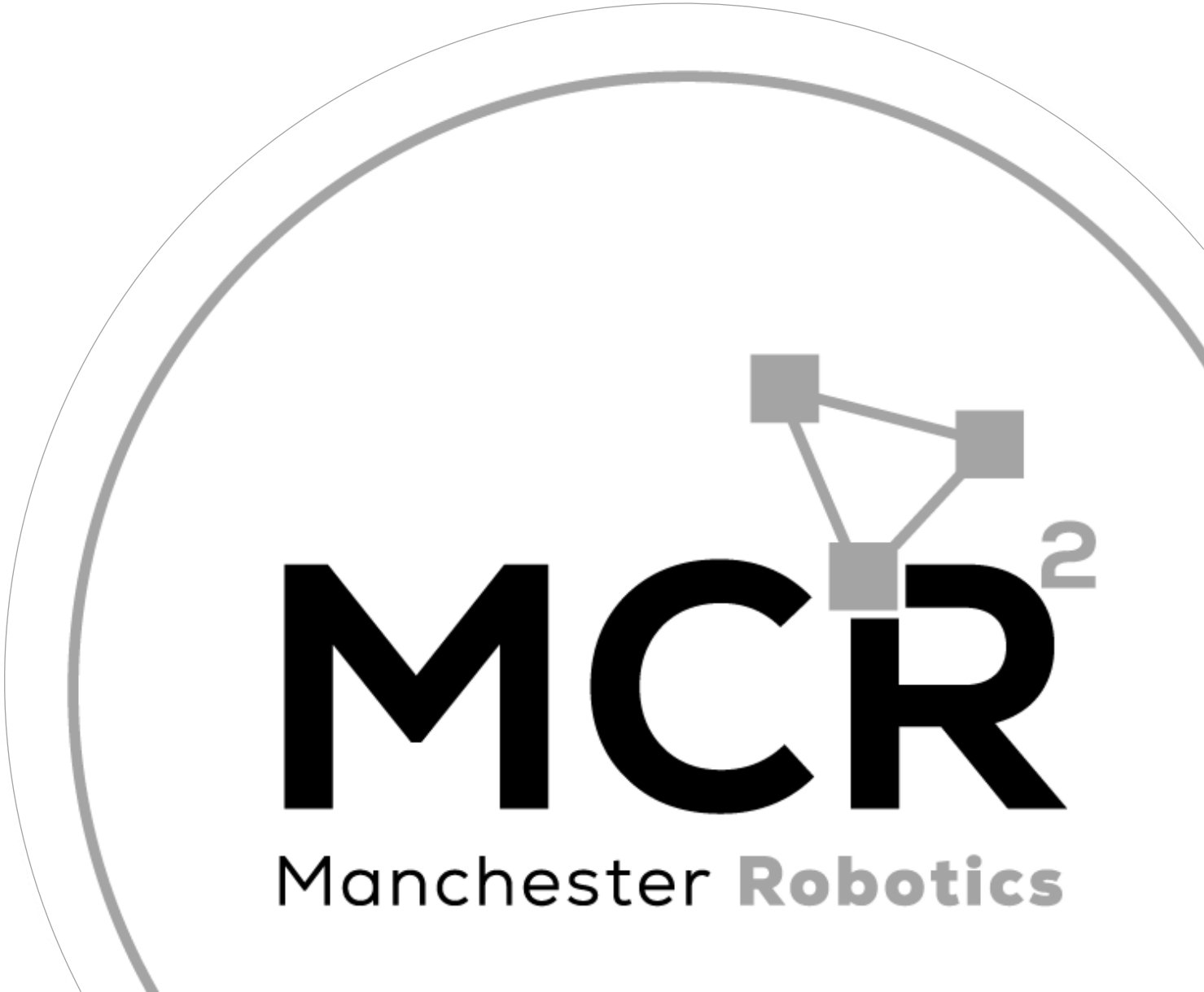
```
$ source install/setup.bash
```



ROS Activities

Activity #1: Talker – Listener

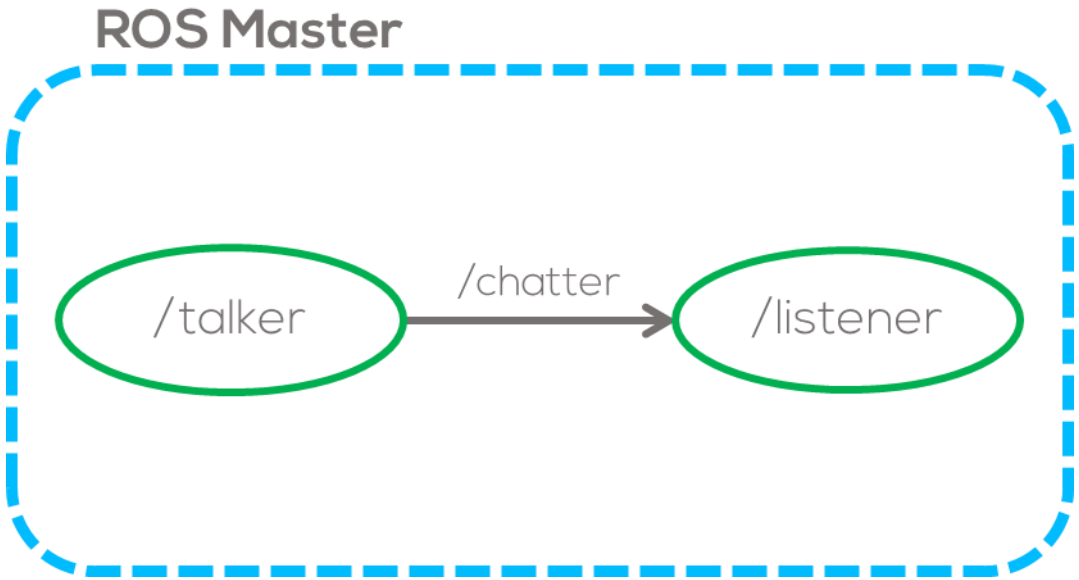
{Learn, Create, Innovate};



ROS Activity 1

Talker - Listener framework

- In this activity, the student will learn about nodes, topic and messages.
- The simplest task to perform in ROS is to communicate 2 nodes.
- The first node to be created will be the “talker” node. This node will send a simple message inside the topic “/chatter”.
- The second node to be programmed, will be the “listener” node. This node will subscribe to the topic “/chatter” of the “talker” node and print on the screen the message.



ROS Activity 1

Requirements

- Ubuntu in VM (MCR2 VM), dual booting or wsl.
- ROS installed (if not follow the steps in this link and select full installation)
- Workspace “ros2_ws” created following the previous steps.

Creating a package

- The easiest way to create a package is to use the ROS tool pkg create.

```
$ ros2 pkg create --build-type ament_python <package_name>
```

- For this exercise, create a package called basic_comms. The node name should be talker. Open a terminal and type the following

```
$ cd ~/catkin_ws/src
$ ros2 pkg create --build-type ament_python --node-name talker basic_comms
```

Beware that the command “`ros2 pkg create`” must be run inside the “src” folder.

- Once the package is created you will be able to see the package folder in ~/ros2_ws/src.
- Build the package you just created and add it to your environment

```
$ cd ~/ros2_ws
$ colcon build
$ source install/setup.bash
```

ROS Activity 1

- Check the executable that was recently created

```
$ ros2 run basic_comms talker
```

- The following information should be published on the screen

```
mario@MarioPC:~/ros2_ws$ ros2 run basic_comms talker
Hi from basic_comms.
```

Creating and Configuring the Talker Node

- If not created automatically, create the publisher node file in ~/ros2_ws/src/basic_comms/basic_comms

```
$ ~/ros2_ws/src/basic_comms/basic_comms
$ touch talker.py
```

- Since the "talker.py" is an executable script, you need give permission to ubuntu to run it.

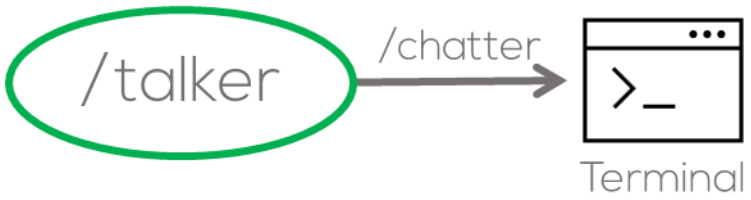
```
$ sudo chmod +x *
Use * to give permission to all files inside a folder
```

- Recall that this directory is a Python package with the same name as the ROS 2 package it's nested in.
- Open talker.py in a text editor (e.g. gedit, vscode, vim, …)

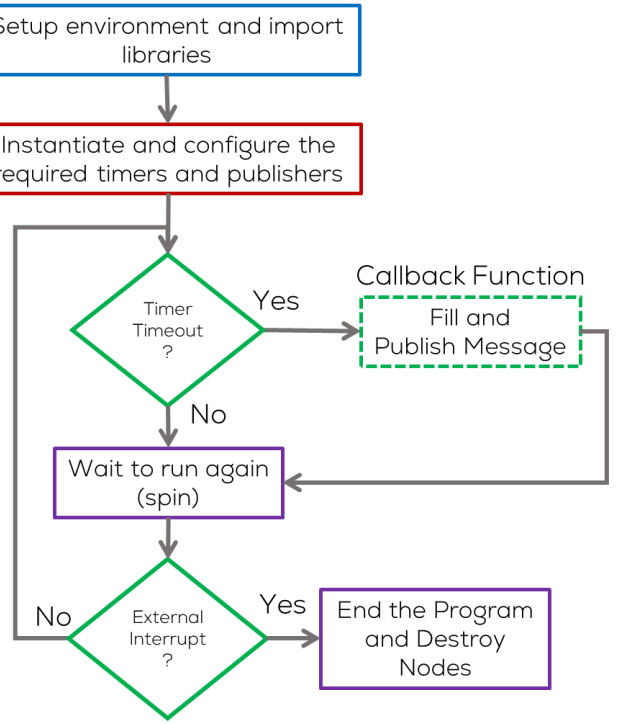
ROS Activity 1

Coding the Talker node

- This node will publish a message (string) into the "/chatter" topic.
- To start, the message will be seen in the terminal to verify that the node is working properly (debug).
- A graphical representation of this task will look as follows



- Nodes in ROS usually have a structure when programmed.



- Your code will be written in the file "talker.py".

Option 1: OOP – Talker node (Recommended)

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

#Class Definition
class MinimalPublisher(Node):
    def __init__(self):
        super().__init__('talker_node')
        self.publisher = self.create_publisher(String, 'chatter', 10)
        timer_period = 0.5 #seconds
        self.timer = self.create_timer(timer_period, self.timer_cb)
        self.i = 0

    #Timer Callback
    def timer_cb(self):
        msg = String()
        msg.data = 'Hello World: %d' %self.i
        self.publisher.publish(msg)
        self.get_logger().info('Publising: "%s"' % msg.data)
        self.i +=1
```

Imports

Create class

Callback Function

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

    minimal_publisher  = MinimalPublisher()

    try:
        rclpy.spin(minimal_publisher)
    except KeyboardInterrupt:
        pass
    finally:
        if rclpy.ok():
            rclpy.shutdown()
        minimal_publisher.destroy_node()


#Execute Node
if __name__ == '__main__':
    main()
```

Main

User Stop (Keyboard Interrupt)

Destroy Node

Execute code

Option 2: Old School– Talker node (Not Recommended)

```python
import rclpy
from rclpy.executors import ExternalShutdownException
from time import sleep
from std_msgs.msg import String


def main(args=None):
    try:
        rclpy.init(args=args)
        node = rclpy.create_node('minimal_publisher')
        publisher = node.create_publisher(String, 'topic', 10)
        msg = String()

        i = 0
        while rclpy.ok():
            msg.data = 'Hello World: %d' % i
            i += 1
            node.get_logger().info('Publishing: "%s"' % msg.data)
            publisher.publish(msg)
            sleep(0.5)  # seconds
    except (KeyboardInterrupt, ExternalShutdownException):
        pass

    finally:
        if rclpy.ok():  # Ensure shutdown is only called once
            rclpy.shutdown()
        node.destroy_node()


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

Imports

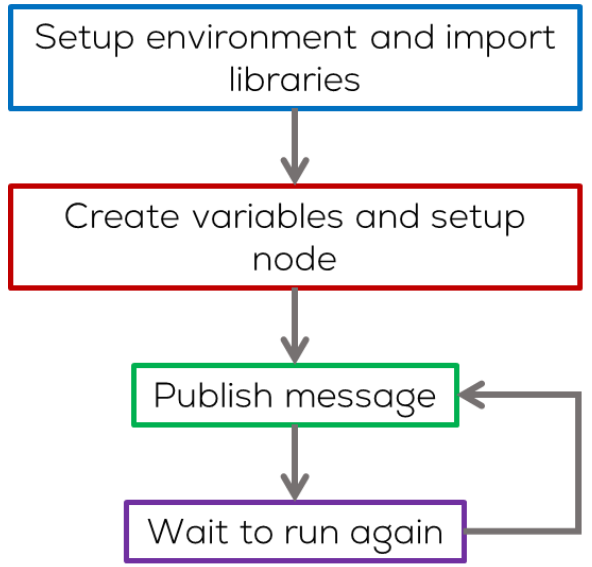
Main

Execute code

Fill message

Publish

Destroy Node



Add dependencies

- Open *package.xml* with your text editor.
- After the license, add the following dependencies corresponding to your node's import statements:

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

Exec dependencies are packages or libraries needed **when running the package**, not just for building it.

- Open the *setup.py* file
- Verify that the entry points are added correctly, if not add them yourself.

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

An entry point is the name of the python scripts that are available to run from with the package, and are shown in the list when you type

```
$ ros2 run <my_py_pkg>  <executable (entry_point)>
```

ROS Exercise 1 – Running the node

- **Running the node**

- Open a terminal and build your package

```
$ cd ~/ros2_ws
$ colcon build --symlink-install
$ source install/setup.bash
```

- Open a new terminal and run the node

```
$ ros2 run basic_comms talker
```

- To visualize the output of the node, open another terminal and use the command "echo" as follows

```
$ ros2 topic echo /chatter
```

Results

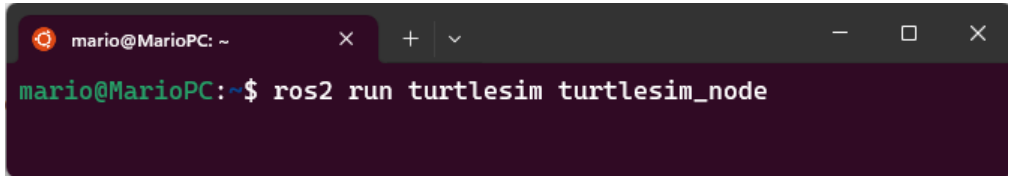

```
mario@MarioPC: ~/ros2_ws
mario@MarioPC:~/ros2_ws$ ros2 run basic_comms talker
[INFO] [1737107350.071949959] [talker_node]: Publising: "Hello World: 0"
[INFO] [1737107350.570230870] [talker_node]: Publising: "Hello World: 1"
[INFO] [1737107351.062139555] [talker_node]: Publising: "Hello World: 2"
[INFO] [1737107351.573222669] [talker_node]: Publising: "Hello World: 3"
[INFO] [1737107352.061774640] [talker_node]: Publising: "Hello World: 4"
[INFO] [1737107352.552430577] [talker_node]: Publising: "Hello World: 5"
[INFO] [1737107353.060345187] [talker_node]: Publising: "Hello World: 6"
^Cmario@MarioPC:~/ros2_ws$
```

Press "Ctrl+c" at each open terminal to stop the nodes and ROS

Nodes Command Line

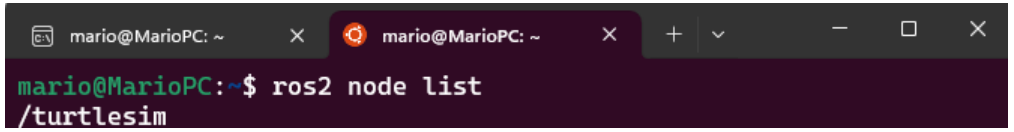
- Launching an executable form a package

```
ros2 run <package_name> <executable_name>
```



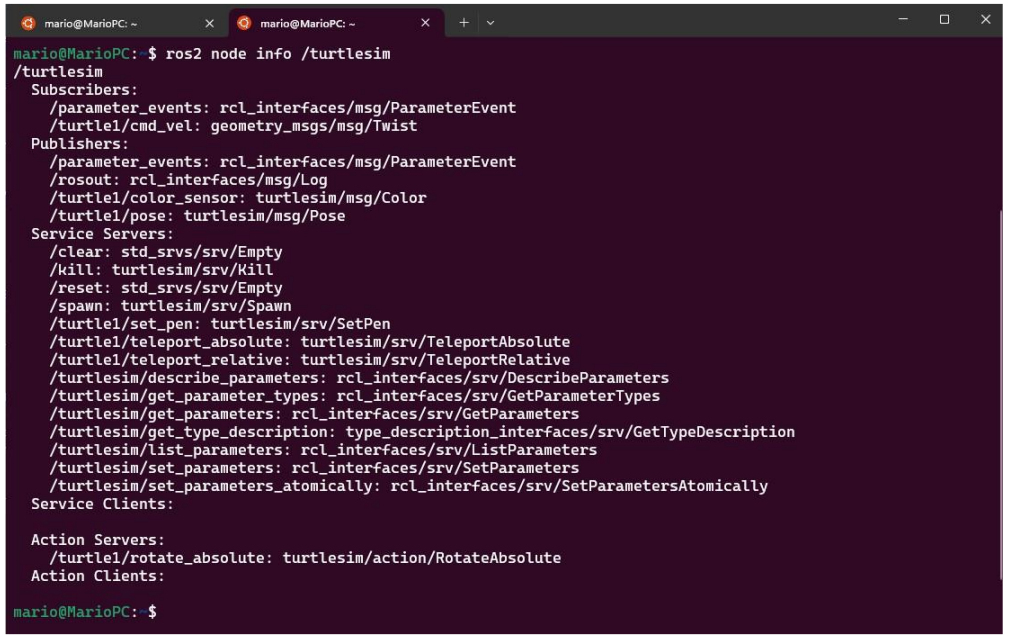
- Show all the running nodes

```
ros2 node list
```



- Detailed information of the available nodes

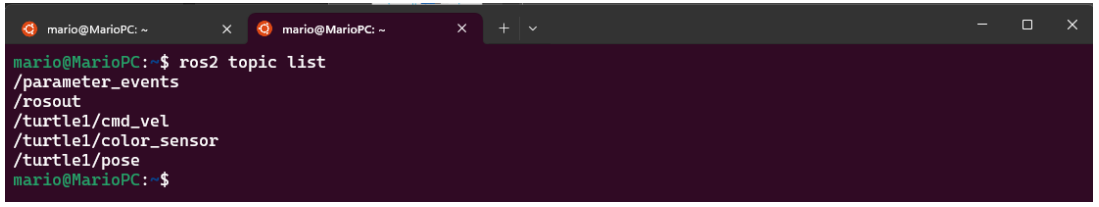
```
ros2 node info <node_name>
```



Topics Command Line

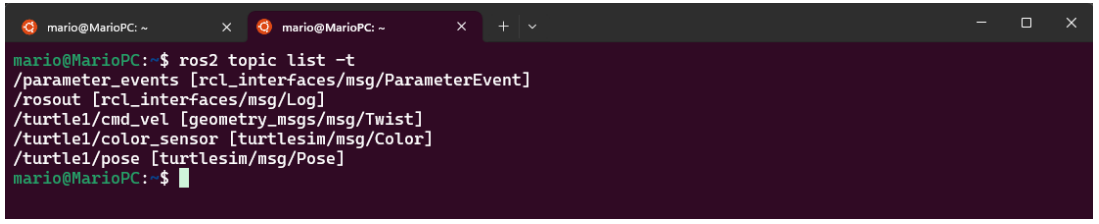
- List of all the topics currently active in the system

```
ros2 topic list
```



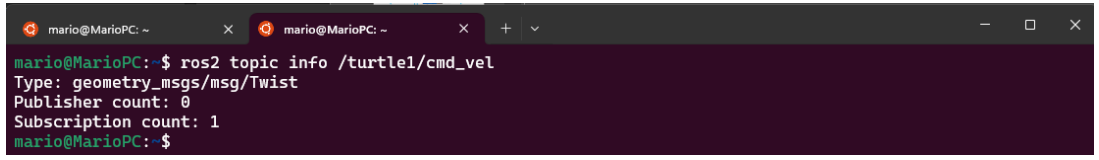
- List of all the topics currently active in the system with message information

```
ros2 topic list -t
```



- See the data being published on a topic

```
ros2 topic info <topic_name>
```

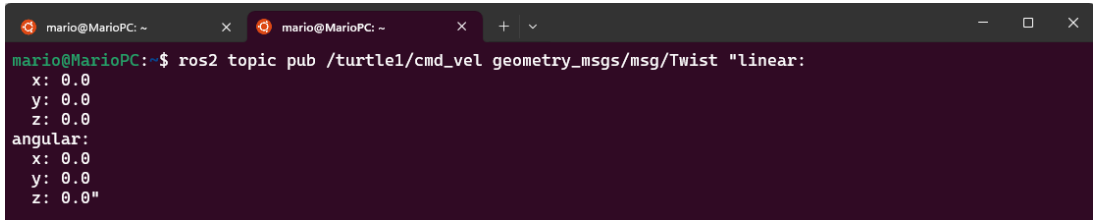


Topics Command Line

- Publish data onto a topic directly from the command line

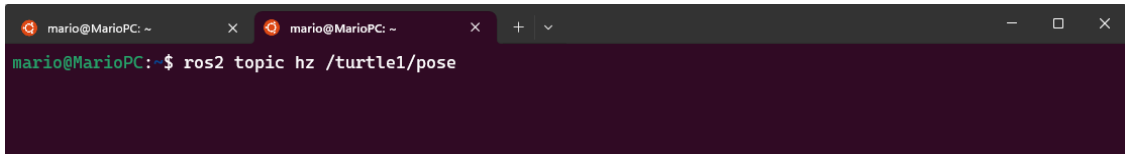
```
ros2 topic pub <topic_name> <msg_type> '<args>’

'<args>’ - data to pass to the topic in YAML syntax
```



- View the rate at which data is published

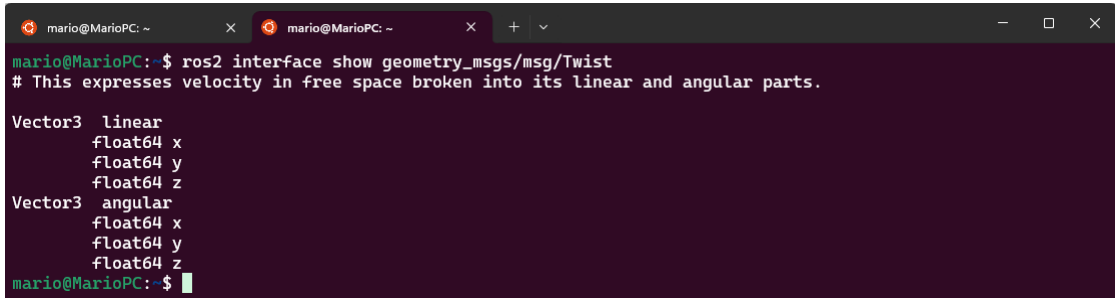
```
ros2 topic hz <topic_name>
```



Interface (Messages) Command Line

- Publish data onto a topic directly from the command line

```
ros2 interface show <msg_type>
```



ROS Activity 1

Listener Node

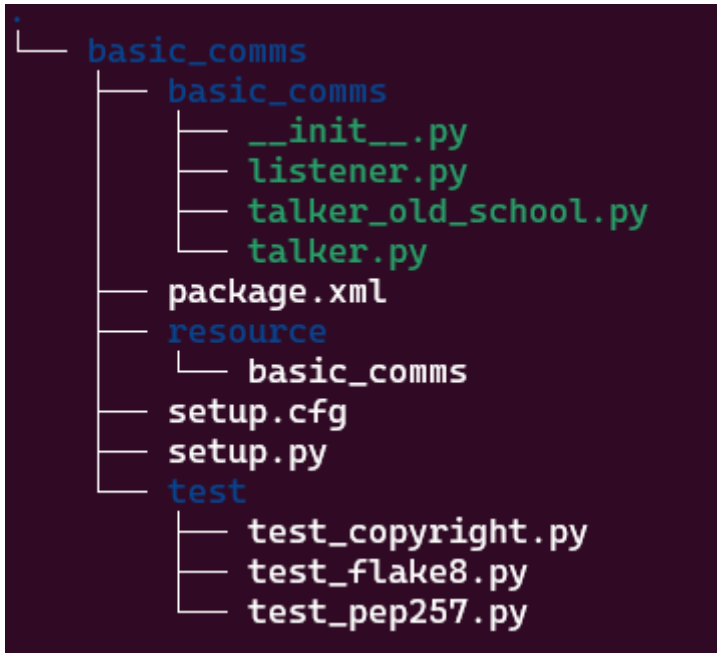
- In as in the previous node, create a script called "listener.py" inside the basic_comms package.

```
$ ~/ros2_ws/src/basic_comms/basic_comms/src
$ touch listener.py
```

- Give permission for the node to be executed

```
$ sudo chmod +x listener.py
```

- Recall that this directory is a Python package with the same name as the ROS 2 package it's nested in.
- Open listener.py in a text editor (e.g. gedit, vscode, vim, …)

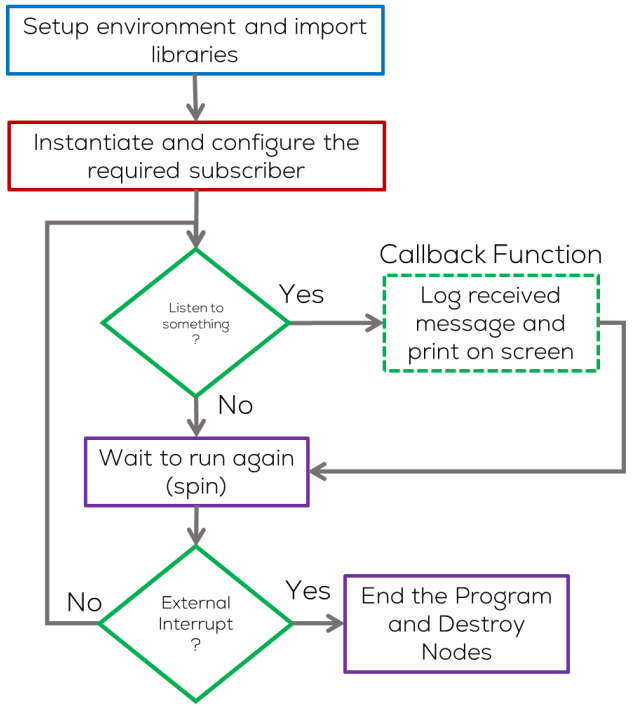


ROS Activity 1

Coding the Listener node

- This node will subscribe to the "/chatter" topic and display on terminal the message (String) it has received.
- To start, the message will be sent form the terminal (manually) to verify that the node is working properly (debug).
- A graphical representation of this task will look as follows

- The structure to be used for this node is the following.



- Your code will be written in the file "listener.py".

Option 1: OOP – Listener node (Recommended)

```python
import rclpy
from rclpy.node import Node

from std_msgs.msg import String

class MinimalSubscriber(Node):

    def __init__(self):
        super().__init__('listener_node')
        self.subscription = self.create_subscription(
            String, 'chatter', self.listener_callback,10)
        self.subscription #No unusedvariable warning


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

Imports

Constructor

Subscriber callback

```python
#Main
def main(args=None):

    rclpy.init(args=args)
    minimal_subscriber  = MinimalSubscriber()

    try:
        rclpy.spin(minimal_subscriber)
    except KeyboardInterrupt:
        pass
    finally:
        if rclpy.ok():  # Ensure shutdown is only called once
            rclpy.shutdown()
        minimal_subscriber.destroy_node()

#Execute Node
if __name__ == '__main__':
    main()
```

Main

User Stop (Keyboard Interrupt)

Destroy Node

Execute code

Option 2: Old School– Listener node (Not Recommended)

```python
import rclpy
from rclpy.executors import ExternalShutdownException
from std_msgs.msg import String

g_node = None

def chatter_callback(msg):
    global g_node
    g_node.get_logger().info(
        'I heard: "%s"' % msg.data)

def main(args=None):
    global g_node
    rclpy.init(args=args)
    g_node = rclpy.create_node('minimal_subscriber')
    subscription = g_node.create_subscription(String, 'chatter',
chatter_callback, 10)
    subscription  # prevent unused variable warning

    try:
        while rclpy.ok():
            rclpy.spin_once(g_node)
    except (KeyboardInterrupt, ExternalShutdownException):
        pass
    finally:
        if rclpy.ok():  # Ensure shutdown is only called once
            rclpy.shutdown()
        g_node.destroy_node()

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

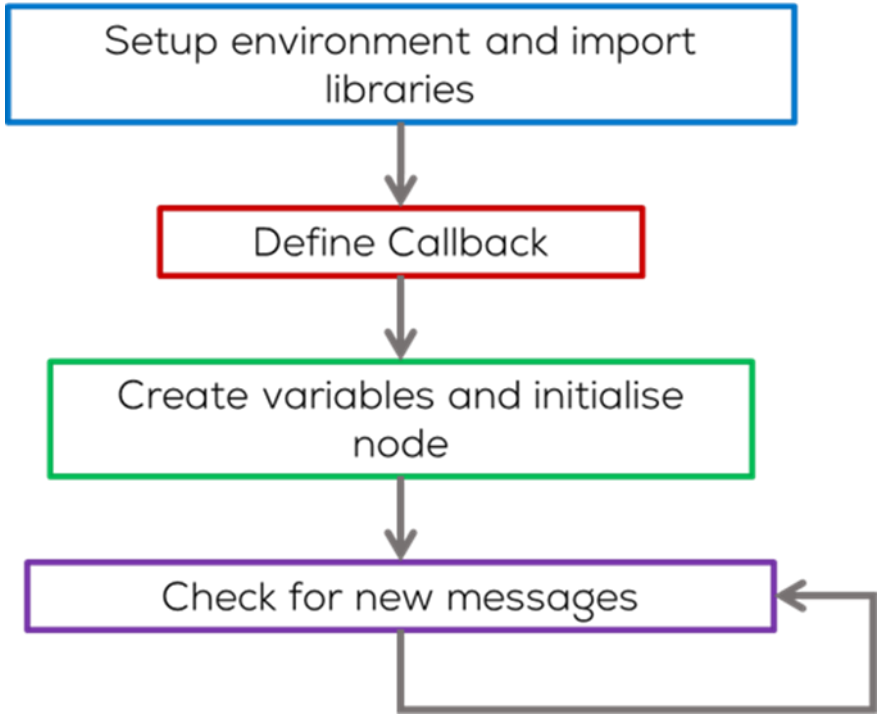
Imports

Subscriber Callback

Configure the subscriber

Spin

Destroy Node



ROS Exercise 1 – Running the node

- Add the entry point to the *setup.py* file

```
entry_points={
    'console_scripts': [
        'talker = basic_comms.talker:main',
        'listener = basic_comms.listener:main',
    ],
},
```

- Open a terminal and build your package

```
$ cd ~/ros2_ws
$ colcon build --symlink-install
$ source install/setup.bash
```

- Open a new terminal and run the node

```
$ ros2 run basic_comms listener
```

Results

- To visualize the output of the node, open another terminal and use the command “echo” as follows

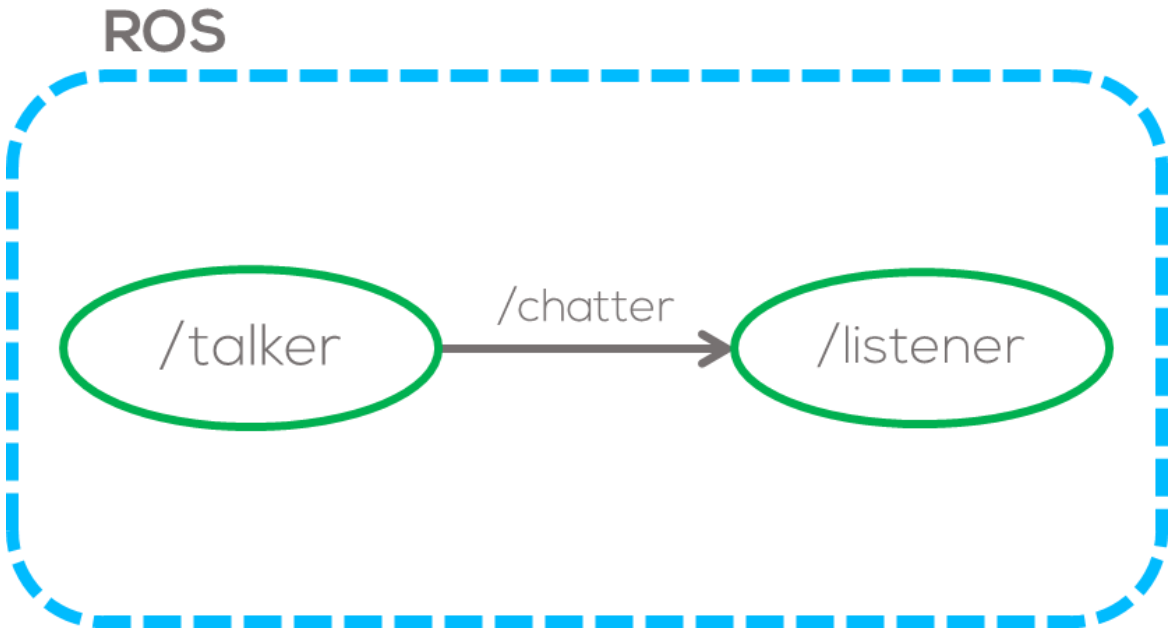
```
$ ros2 topic pub /chatter std_msgs/msg/String "data: Hello"
```

Press “Ctrl+c” at each open terminal to stop the nodes and ROS

ROS Activity 1

Talker – Listener Nodes

- Having developed both nodes, now is time to put everything together and let the nodes to communicate.
- This can be achieved in two different ways, manually and via a ROS tool called launch file.



ROS Activity 1 – Running the nodes

- Open a new terminal and run the talker node you just made using the following command

```
$ source install/setup.bash
$ ros2 run basic_comms talker
```

- Open a new terminal and run the listener node you just made using the following command

```
$ source install/setup.bash
$ ros2 run basic_comms listener
```

Results


```
mario@MarioPC:~/ros2_ws$ source install/setup.bash
mario@MarioPC:~/ros2_ws$ ros2 run basic_comms talker
[INFO] [1737110747.804010629] [talker_node]: Publising: "Hello World: 0"
[INFO] [1737110748.284448398] [talker_node]: Publising: "Hello World: 1"
[INFO] [1737110748.782552292] [talker_node]: Publising: "Hello World: 2"
[INFO] [1737110749.281078642] [talker_node]: Publising: "Hello World: 3"
[INFO] [1737110749.785573422] [talker_node]: Publising: "Hello World: 4"
[INFO] [1737110750.306490880] [talker_node]: Publising: "Hello World: 5"
[INFO] [1737110750.784166559] [talker_node]: Publising: "Hello World: 6"
```

```
mario@MarioPC:~/ros2_ws$ source install/setup.bash
mario@MarioPC:~/ros2_ws$ ros2 run basic_comms listener
[INFO] [1737110747.794048055] [listener_node]: I heard "Hello World: 0"
[INFO] [1737110748.284812173] [listener_node]: I heard "Hello World: 1"
[INFO] [1737110748.782959386] [listener_node]: I heard "Hello World: 2"
[INFO] [1737110749.281279208] [listener_node]: I heard "Hello World: 3"
[INFO] [1737110749.786445283] [listener_node]: I heard "Hello World: 4"
[INFO] [1737110750.307141764] [listener_node]: I heard "Hello World: 5"
[INFO] [1737110750.784555696] [listener_node]: I heard "Hello World: 6"
```

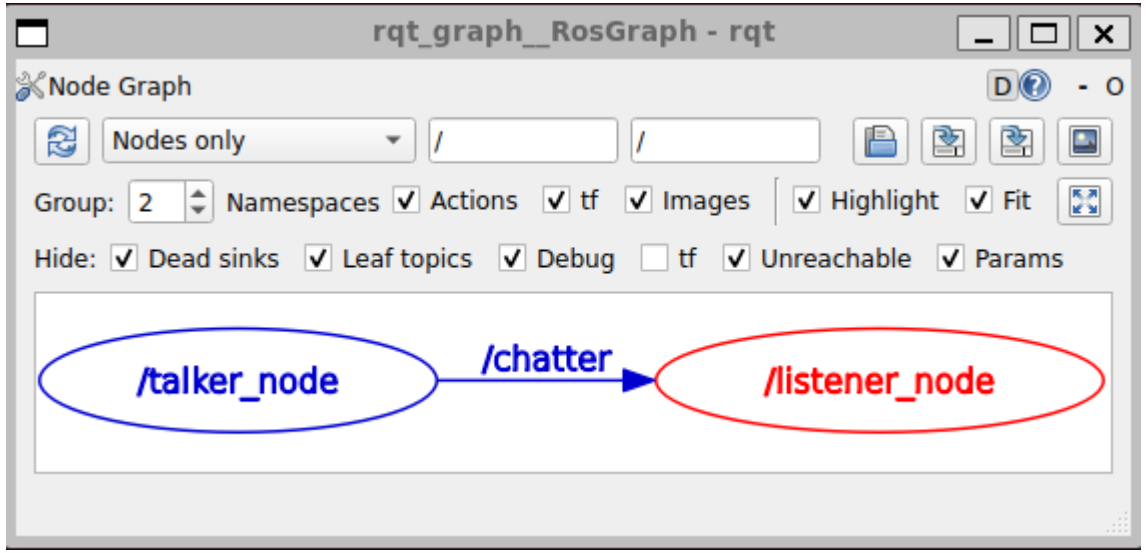
Press “Ctrl+c” at each open terminal to stop the nodes and ROS

ROS Tools for Debugging

- rqt
  - Visualization tool that provides graphical information of the nodes and system status.
    - *Ros2 run rqt_plot*: Displays scalar data published to ROS topics.
    - *ros2 run rqt_graph*: Displays a visual graph of the processes running in ROS and their connections

- Using the previous example, open another terminal and type the following

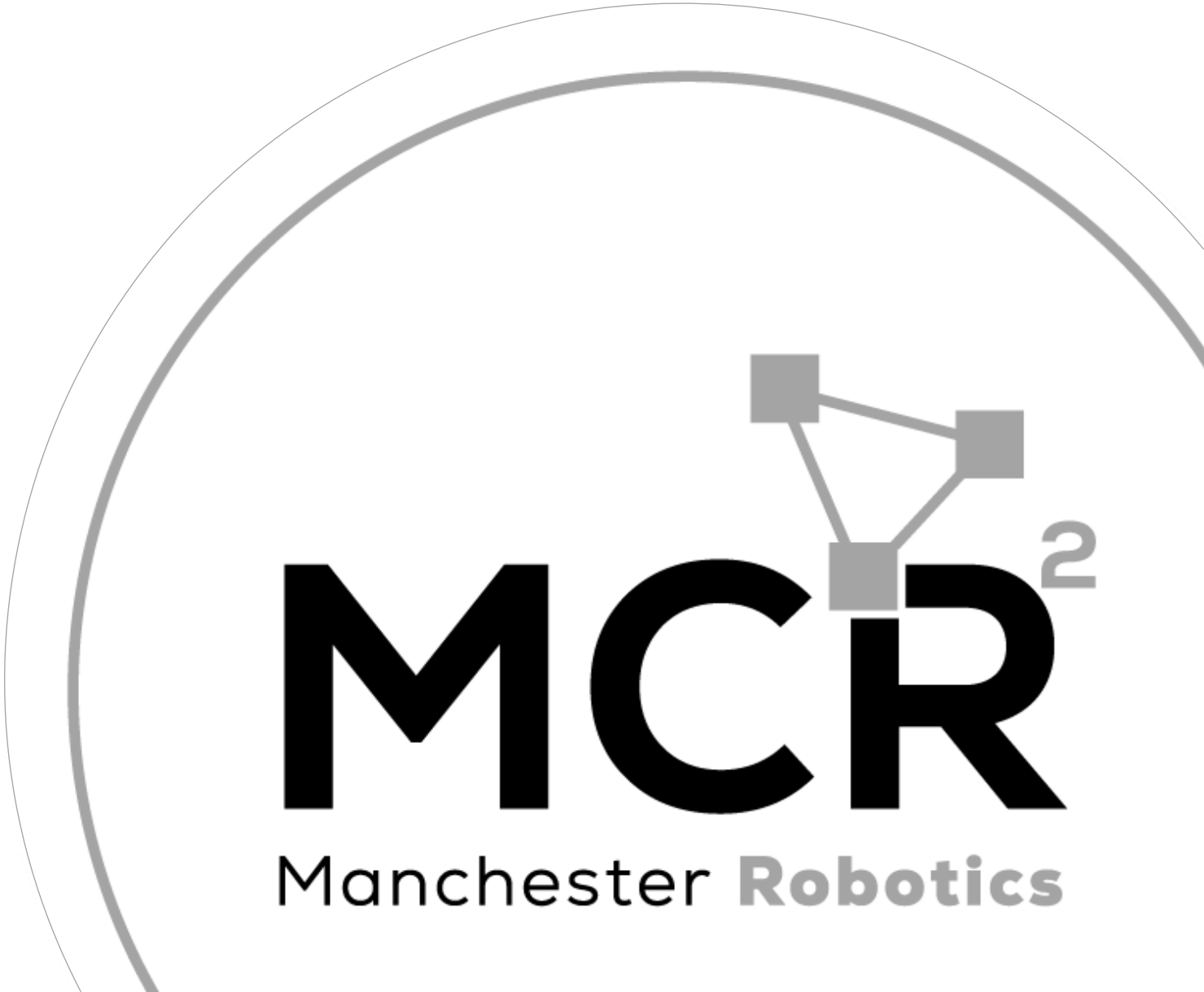
```
$ ros2 run rqt_graph rqt_graph
```



Robot Operating System - ROS

ROS Launch Files

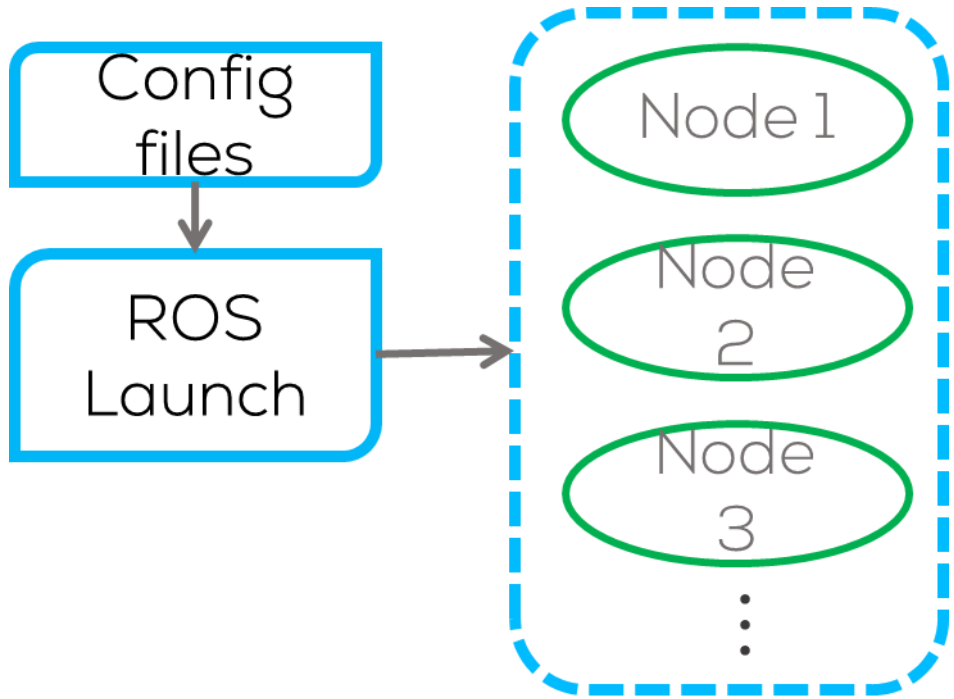
{Learn, Create, Innovate};



ROS2 Launch file

Launch Files

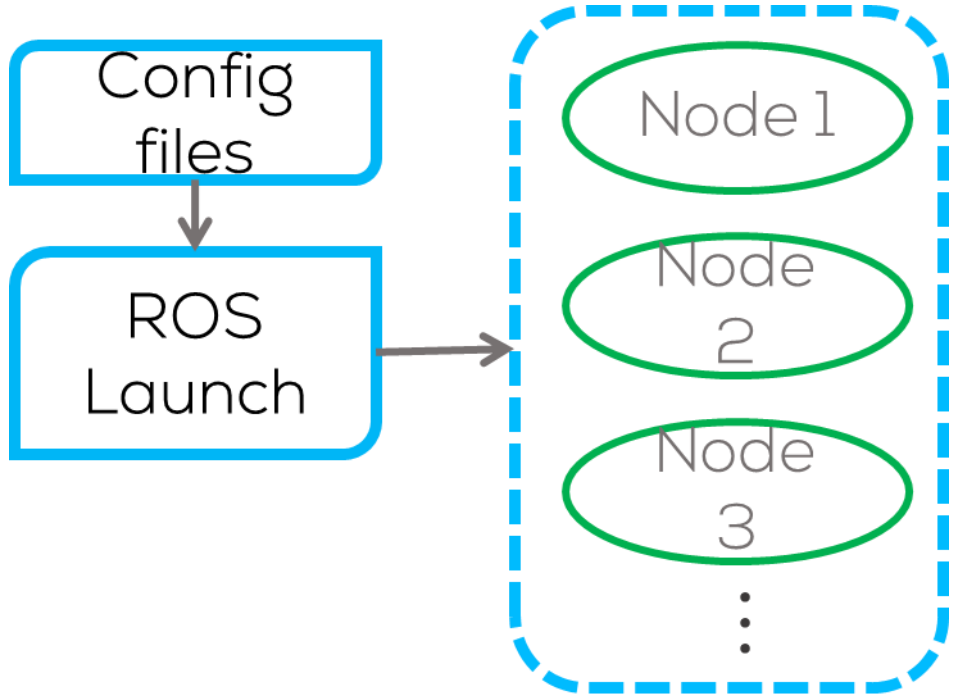
- Launch files are sets of commands written in Python, .xml, and .yaml that allow the simultaneous execution of various scripts.
- Launch files allow you to run any object used within the ROS2 architecture.
- A wide variety of tools allow you to parametrise the launch file and adapt it to a project's requirements.



Launch files

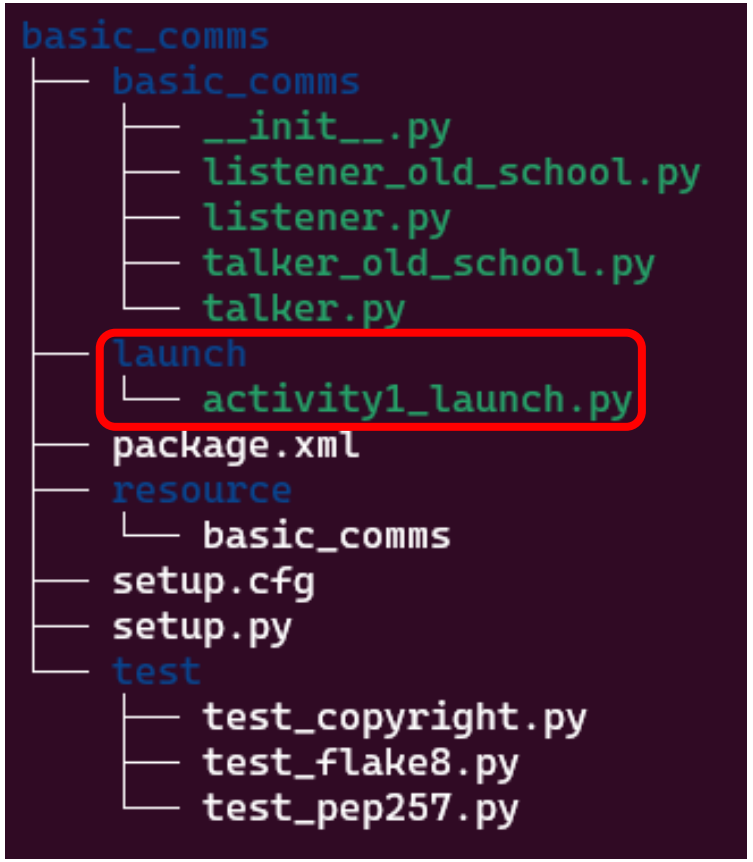
- Launch files allow you to start up and configure several executables containing ROS 2 nodes simultaneously.
- Running a single launch file with the ros2 launch command will start up your entire system - all nodes and their configurations - at once.
- To use a launch file

```
$ ros2 launch <package_name> <file_name>
```



Launch files

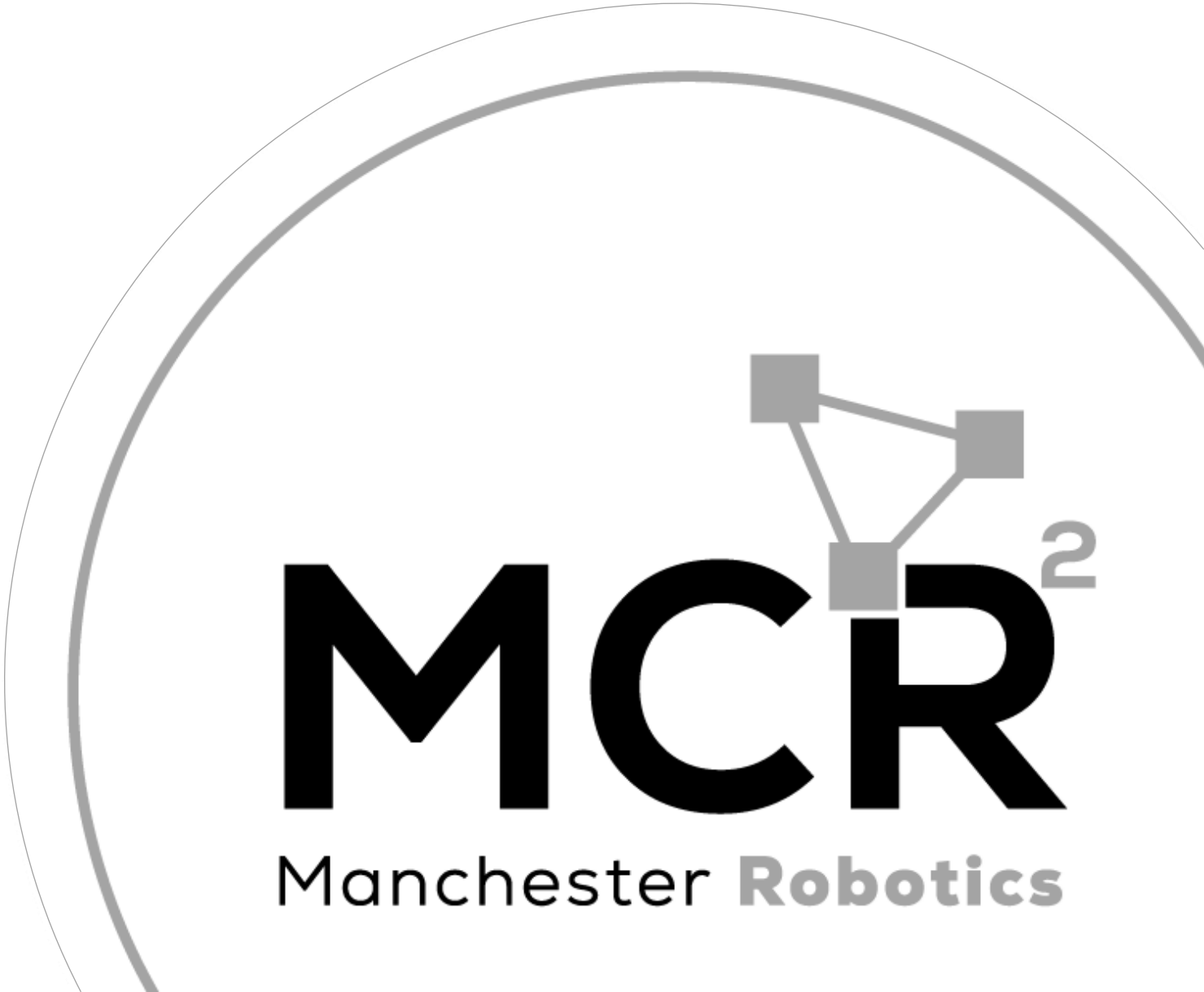
- The user can **set parameters** used by the nodes in ROS.
- The user can also **set arguments** used by the ROS launch files and nodes in ROS.
- **All the processes** will shut down when the ROS launch process **is killed (ctrl+C)**.
- **ROS Launch files** are **usually located** inside a **folder called "launch"** in each package.



Robot Operating System - ROS

Activity 1.2- Making a Launch File

{Learn, Create, Innovate};



ROS Activity 1(Launch file)

- For exercise 1, a ROS Launch file will be developed as follows.
- Make the launch file directory inside the "basic_comms" package.

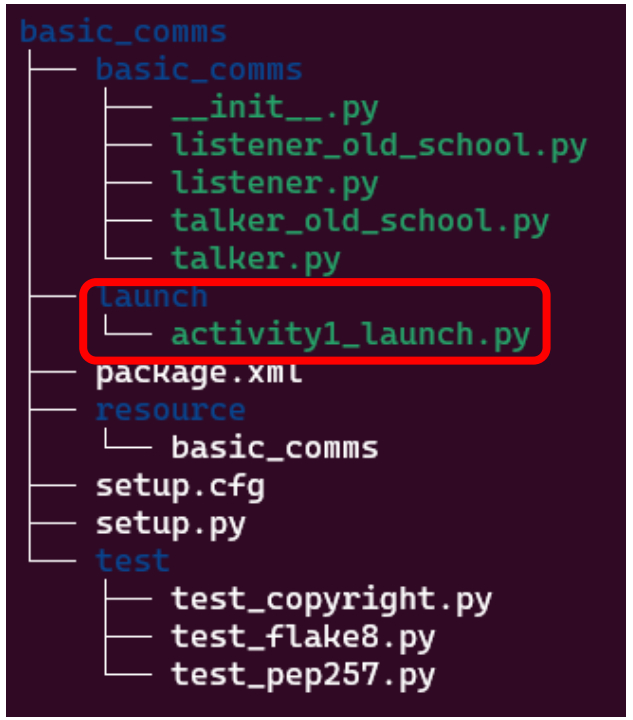
```
$ cd ~/ros2_ws/src/basic_comms
$ mkdir launch
```

- Make a launch file and give exedcutable permissions inside the previous folder

```
$ cd launch
$ touch activity1_launch.py
$ chmod +x activity1_launch.py
```

IMPORTANT NOTE
Launch files must be named as follows:
<NAME>_launch.py #Replace <NAME> with a name set by the user

- For this exercise, the talker and listener nodes (previously done) will be launched.



ROS2 Launch file

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

def generate_launch_description():
    talker_node = Node(package='basic_comms',           Node to
                       executable='talker',             be
                       output='screen'                  Launched
                       )
    listener_node = Node(package='basic_comms',         Node to
                         executable='listener',         be
                         output='screen'                Launched
                         )

    l_d = LaunchDescription([talker_node,listener_node])
    return l_d                                          Set
                                                        launch
                                                        content
```

- Open activity1_launch.py in a text editor (e.g. gedit, vscode, vim, …), write and save the launch code given in the snippet.
- Open the *setup.py* file and add the following lines at the top.

```
import os
from glob import glob
```

- *Add the following inside "data_files"*

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

ROS Activity 1 (Launch file)

- To run the roslaunch file, open a terminal and type

```
$cd ~/ros2_ws
$ colcon build
$ source install/setup.bash
```

- Run the launch file using the following command

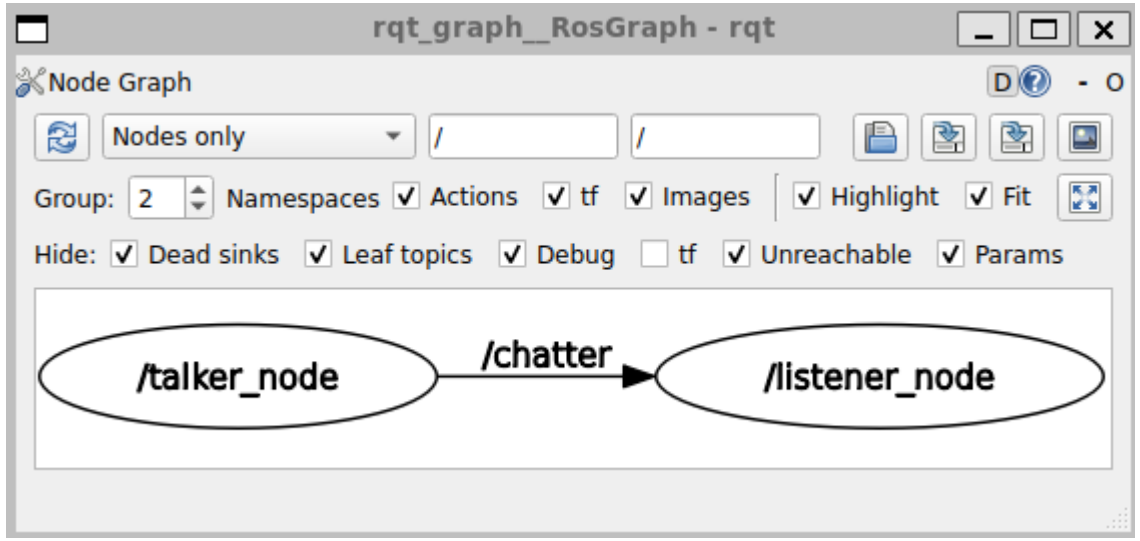
```
$ ros2 launch basic_comms activity1_launch.py
```

```
mario@MarioPC:~/ros2_ws/src$ ros2 launch basic_comms activ
ity1_launch.py
[INFO] [launch]: All log files can be found below /home/ma
rio/.ros/log/2025-01-17-13-13-13-808729-MarioPC-44977
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [talker-1]: process started with pid [44980]
[INFO] [listener-2]: process started with pid [44981]
[talker-1] [INFO] [1737115994.860153944] [talker_node]: Pu
blising: "Hello World: 0"
[listener-2] [INFO] [1737115994.860698030] [listener_node]
: I heard "Hello World: 0"
[talker-1] [INFO] [1737115995.334933573] [talker_node]: Pu
blising: "Hello World: 1"
[listener-2] [INFO] [1737115995.335287790] [listener_node]
: I heard "Hello World: 1"
[talker-1] [INFO] [1737115995.843716088] [talker_node]: Pu
blising: "Hello World: 2"
[listener-2] [INFO] [1737115995.844653703] [listener_node]
: I heard "Hello World: 2"
[talker-1] [INFO] [1737115996.342499708] [talker_node]: Pu
blising: "Hello World: 3"
```

Results

- Running the ROS tool “rqt_graph” it is possible to observe the nodes currently active

```
$ ros2 run rqt_graph rqt_graph
```



Q&A

Questions?

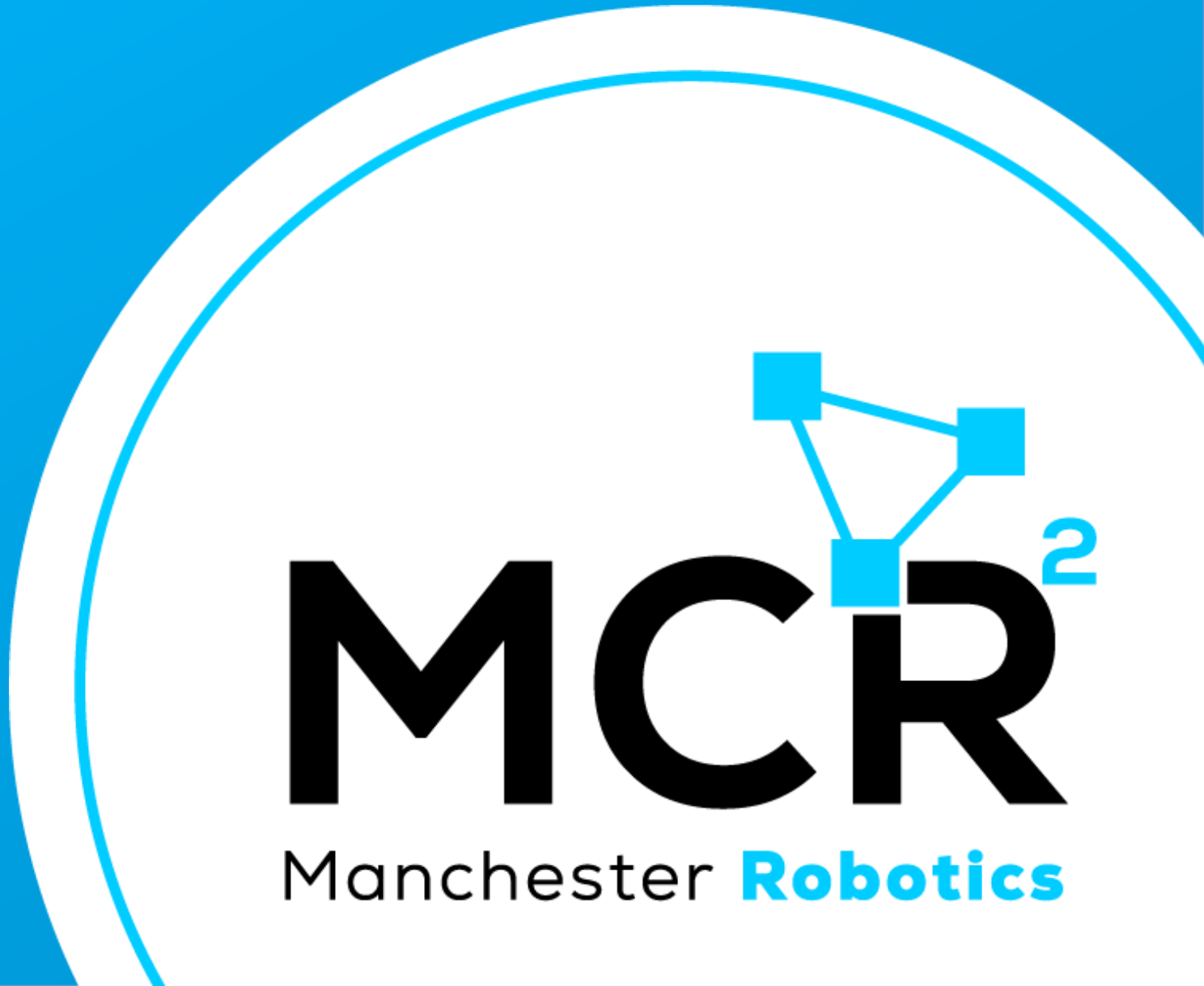
{Learn, Create, Innovate};

MCR² Manchester Robotics

Thank You

Robotics For Everyone

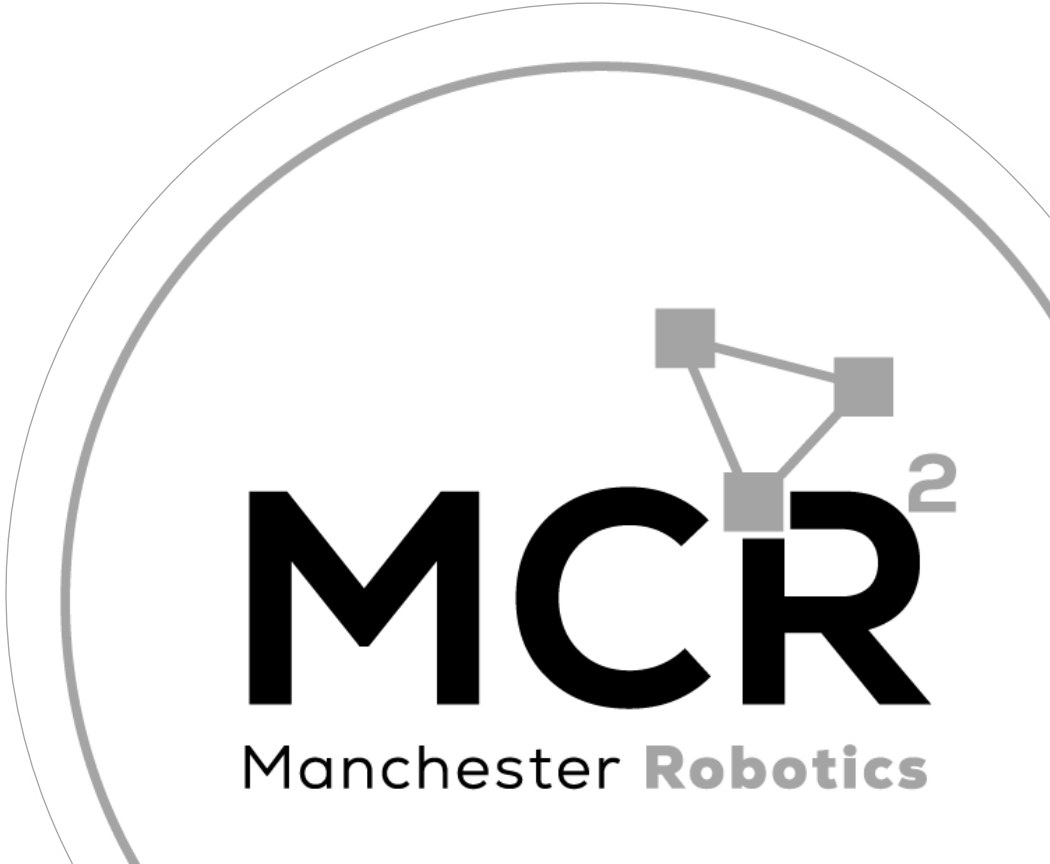
{Learn, Create, Innovate};



T&C

Terms and conditions

{Learn, Create, Innovate};



Terms and conditions

- *THE PIECES, IMAGES, VIDEOS, DOCUMENTATION, ETC. SHOWN HERE ARE FOR INFORMATIVE PURPOSES ONLY. THE DESIGN IS PROPRIETARY AND CONFIDENTIAL TO MANCHESTER ROBOTICS LTD. (MCR2). THE INFORMATION, CODE, SIMULATORS, DRAWINGS, VIDEOS PRESENTATIONS ETC. CONTAINED IN THIS PRESENTATION IS THE SOLE PROPERTY OF MANCHESTER ROBOTICS LTD. ANY REPRODUCTION, RESELL, REDISTRIBUTION OR USAGE IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF MANCHESTER ROBOTICS LTD. IS STRICTLY PROHIBITED.*
- *THIS PRESENTATION MAY CONTAIN LINKS TO OTHER WEBSITES OR CONTENT BELONGING TO OR ORIGINATING FROM THIRD PARTIES OR LINKS TO WEBSITES AND FEATURES IN BANNERS OR OTHER ADVERTISING. SUCH EXTERNAL LINKS ARE NOT INVESTIGATED, MONITORED, OR CHECKED FOR ACCURACY, ADEQUACY, VALIDITY, RELIABILITY, AVAILABILITY OR COMPLETENESS BY US.*
- *WE DO NOT WARRANT, ENDORSE, GUARANTEE, OR ASSUME RESPONSIBILITY FOR THE ACCURACY OR RELIABILITY OF ANY INFORMATION OFFERED BY THIRD-PARTY WEBSITES LINKED THROUGH THE SITE OR ANY WEBSITE OR FEATURE LINKED IN ANY BANNER OR OTHER ADVERTISING.*