{Learn, Create, Innovate};

Robot Operating System - ROS

Introduction

Manchester Robotics

Table of contents

MCR^2

Manchester **Robotics**

1 What is ROS

2 ROS Basics

3 ROS Architecture

4 ROS Example

5 ROS Organization

6 ROS Activity

7 ROS Launch Files

8 Questions

Robot Operating System - ROS

What is ROS?

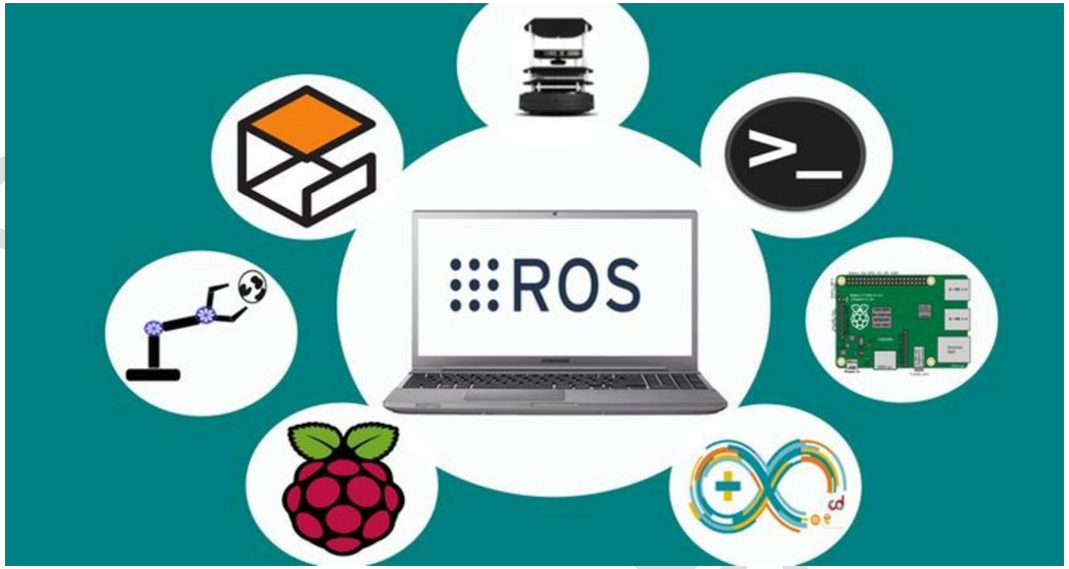
{Learn, Create, Innovate};

MCR²
Manchester Robotics

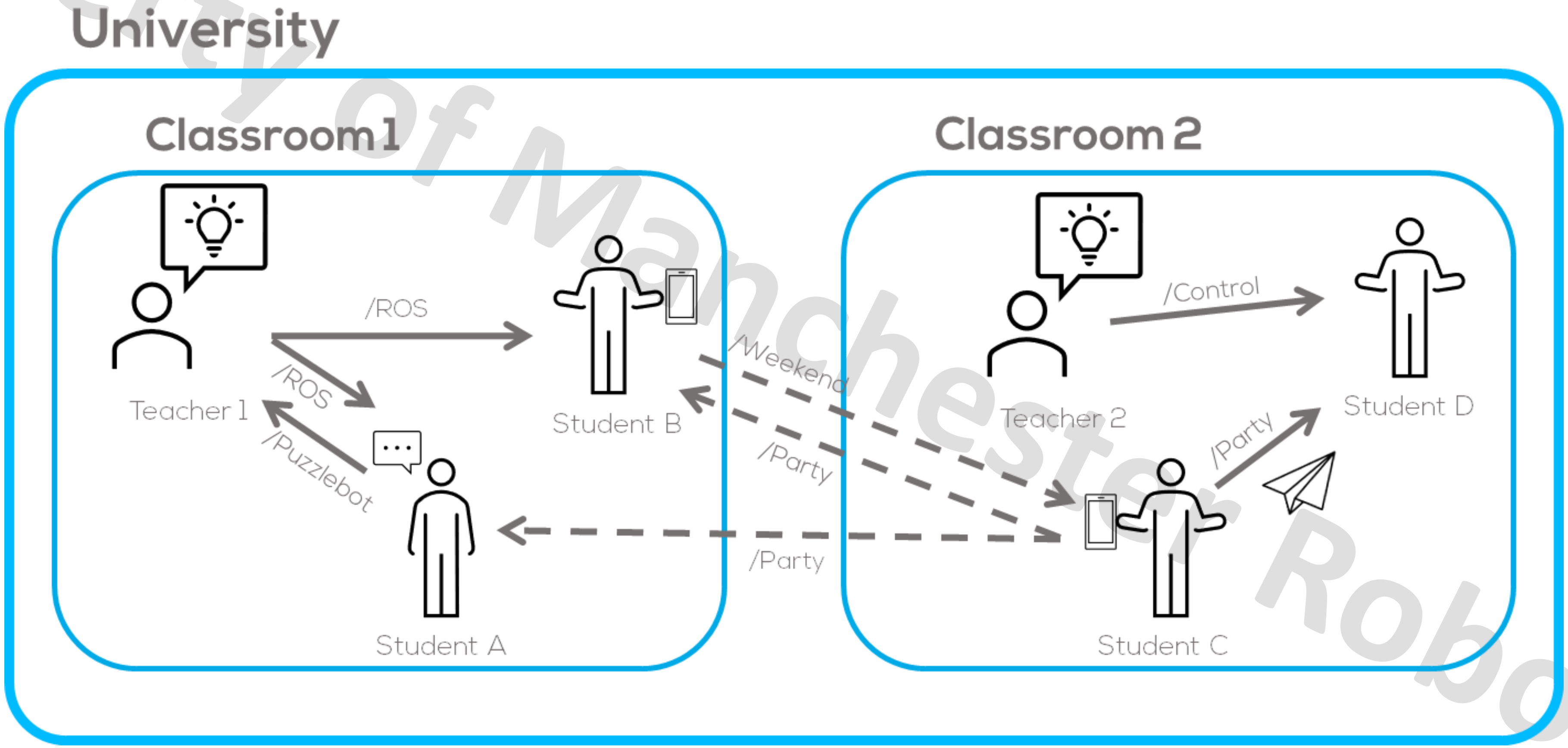
What is ROS?

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 Basics

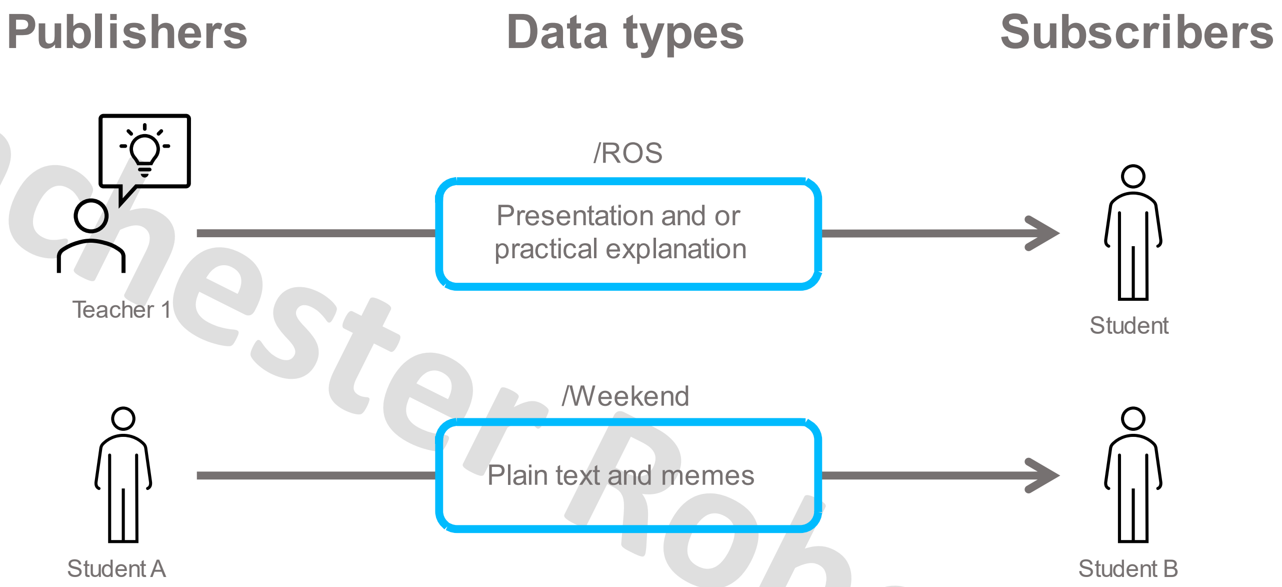


** Any resemblance to reality, is purely coincidental.

ROS Basics

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.



Robot Operating System - ROS

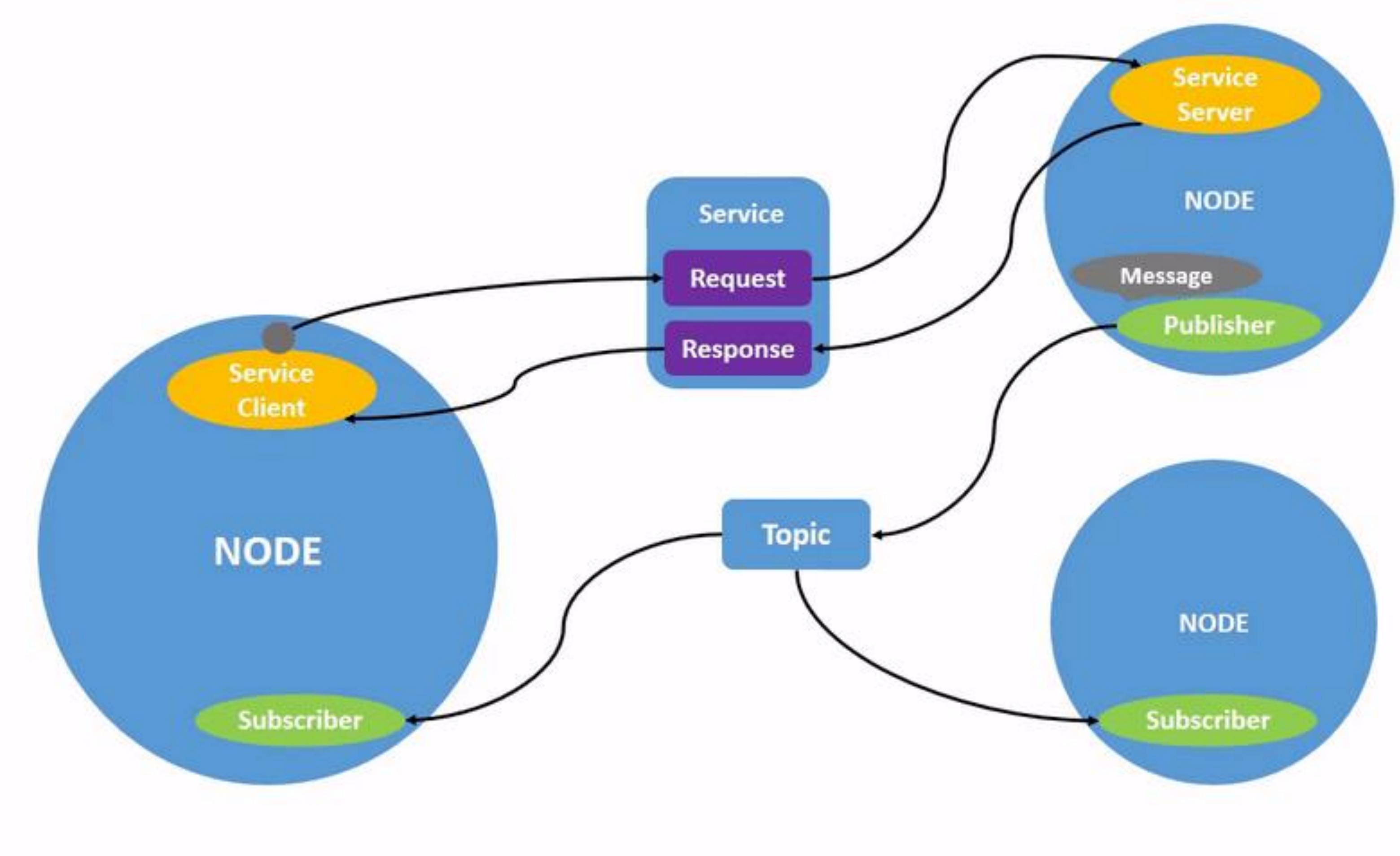
ROS Architecture

{Learn, Create, Innovate};

MCR2

Manchester **Robotics**

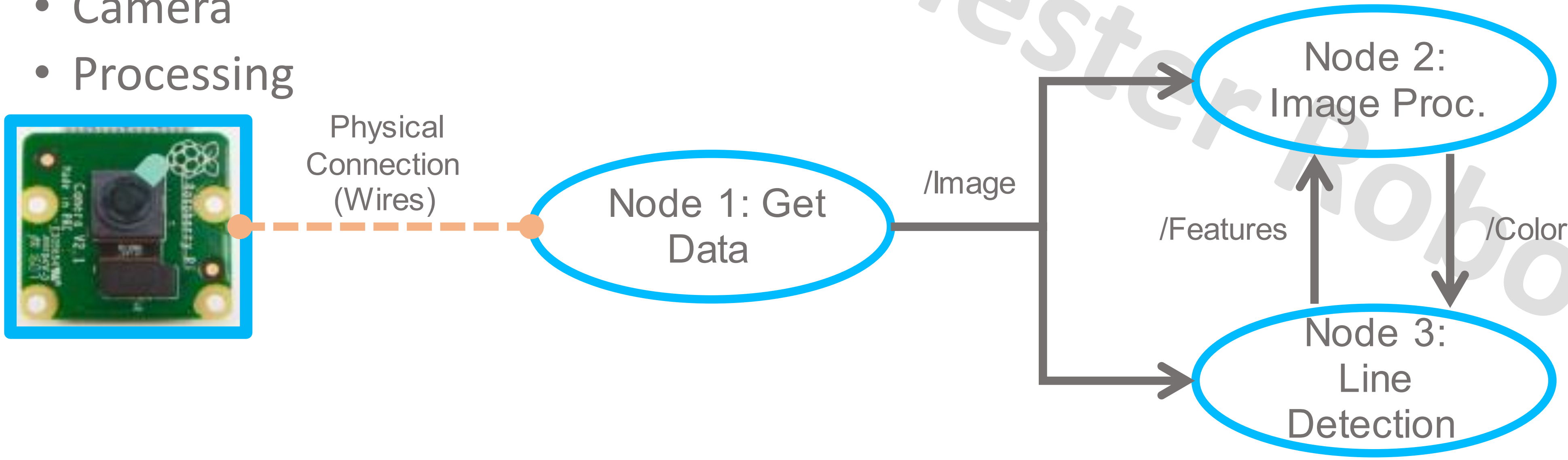
ROS2 graph


Service
Server
NODE
Service
Request
Response
Message
Publisher
Service
Client
NODE
Topic
NODE
Subscriber
Subscriber

Nodes

- Fundamental ROS 2 element that serves a single, modular purpose in a robotics system
  - LRF
  - 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.

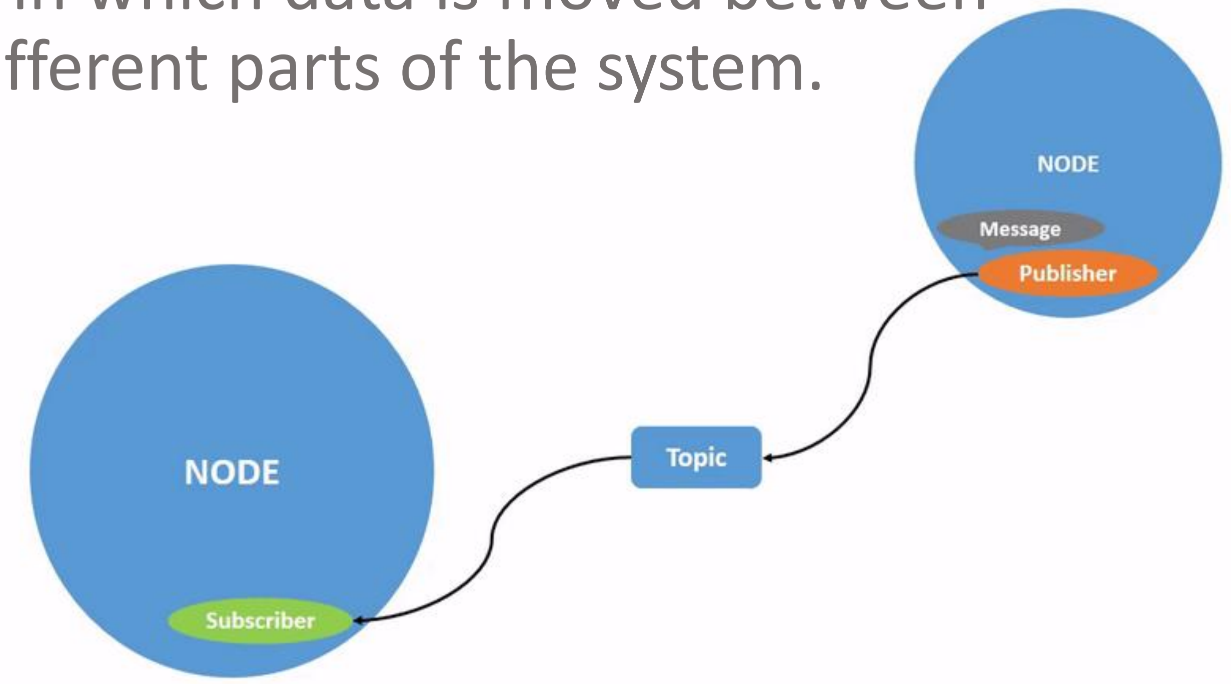


Nodes

- Launching a node
  - 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>

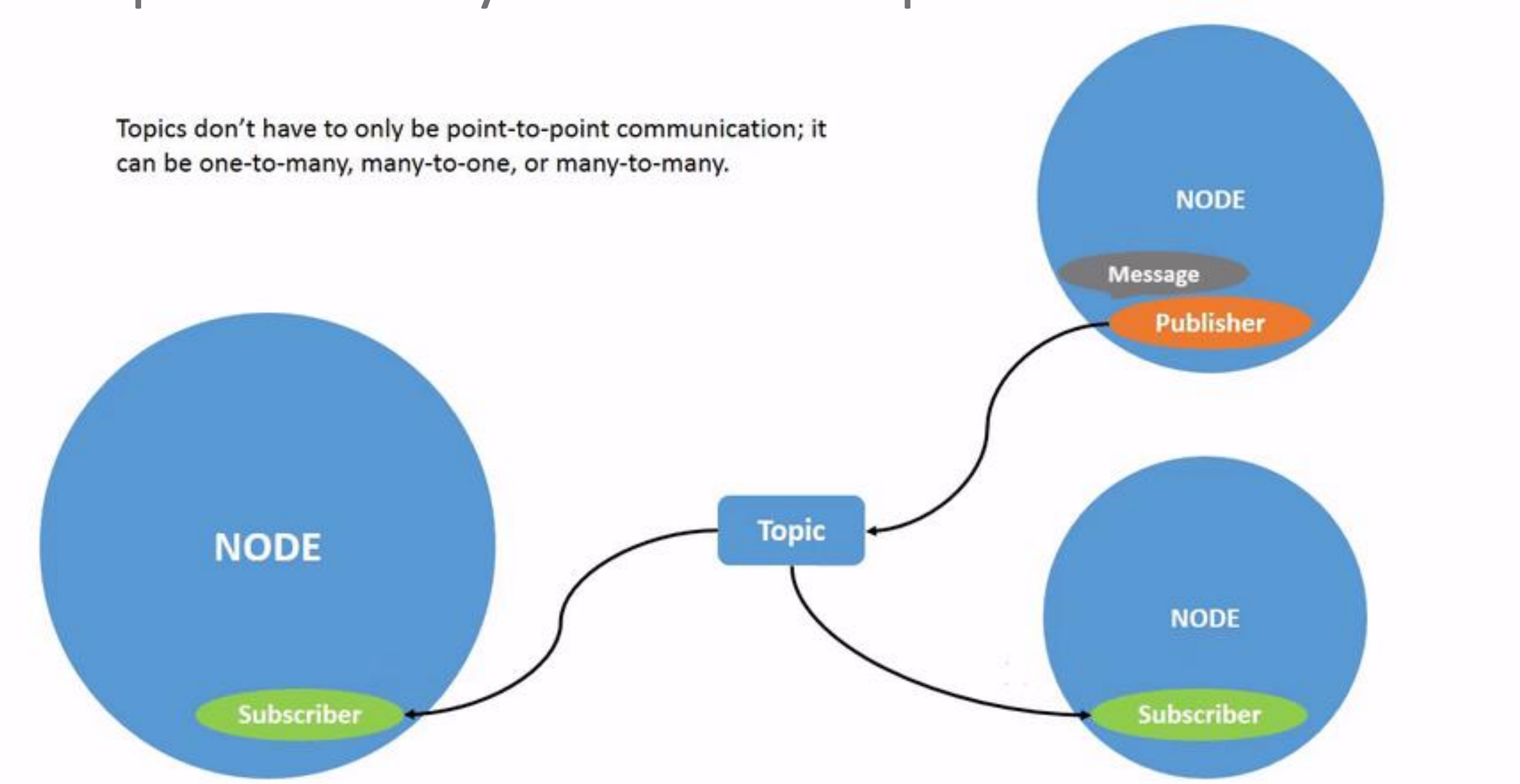
Topics

- Topics 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.



Topics

- A node may publish data to any number of topics and simultaneously have subscriptions to any number of topics.



Topics don't have to only be point-to-point communication; it can be one-to-many, many-to-one, or many-to-many.

Topics

- 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>

    - Type: geometry_msgs/msg/Twist
    - Publisher count: 1
    - Subscription count: 2

Interface (Messages)

- Nodes send data over topics using messages.
- 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.

- ros2 topic list –t
- /turtle1/cmd_vel [geometry_msgs/msg/Twist

Interface (Messages)

- ros2 interface show geometry_msgs/msg/Twist
- # This expresses velocity in free space broken into its linear and angular parts.

- Vector3 linear
  - float64 x
  - float64 y
  - float64 z
- Vector3 angular
  - float64 x
  - float64 y
  - float64 z

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
```

Topics

- 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>

Robot Operating System - ROS

ROS Example

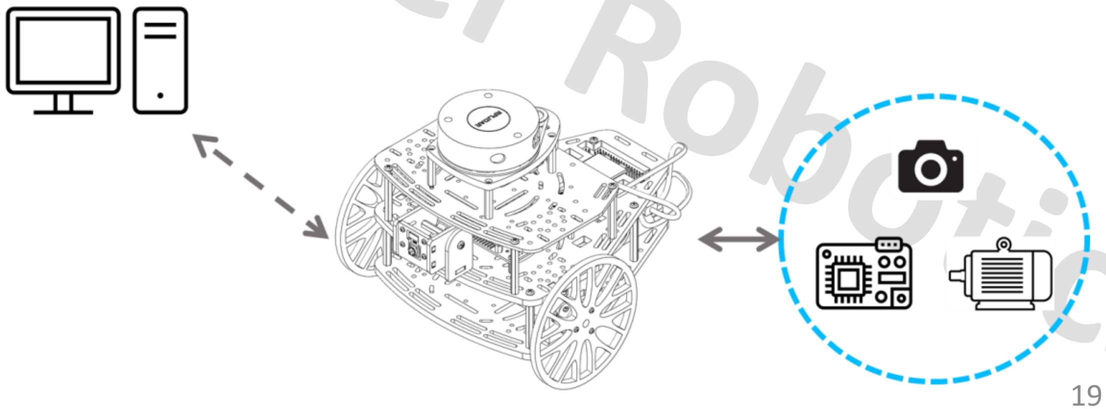
{Learn, Create, Innovate};

MCR^2

Manchester Robotics

ROS Practical Example

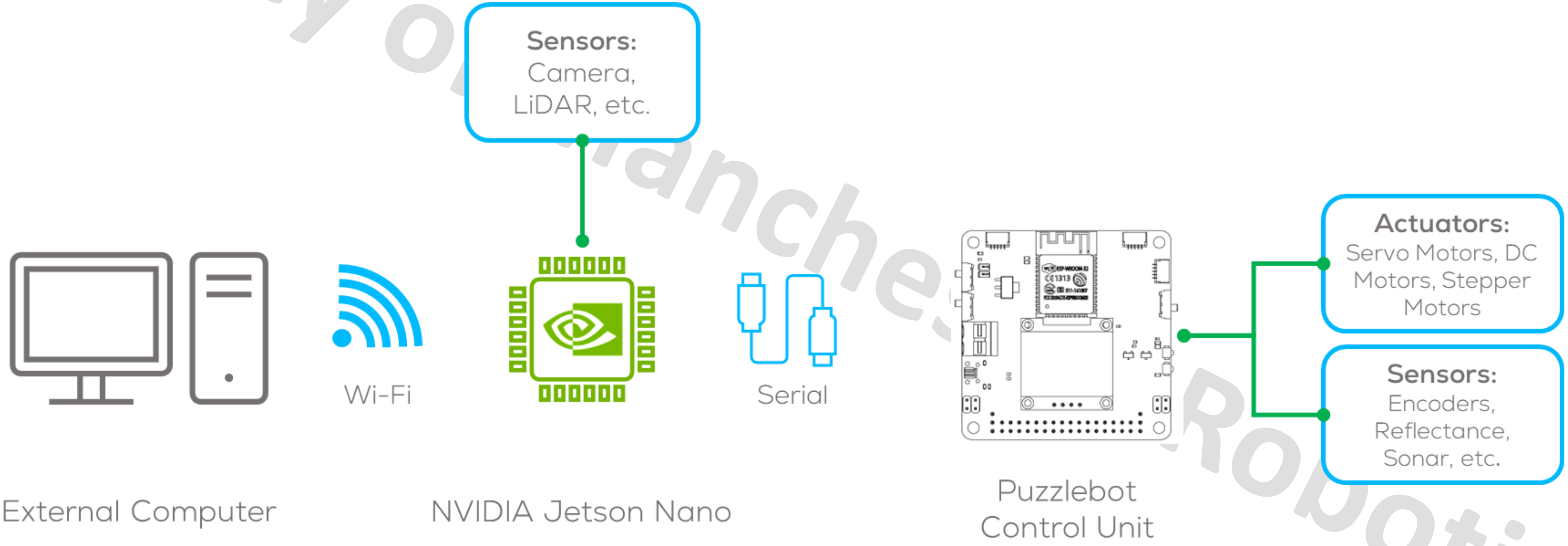
- The system presented is a mobile robot called Puzzlebot® by Manchester Robotics.
- The robot is comprised of an on-board computing unit from NVIDIA the Jetson Nano used for high level control algorithms and a microcontroller for low level control tasks.
- The robot also contains different sensors and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



ROS Practical Example

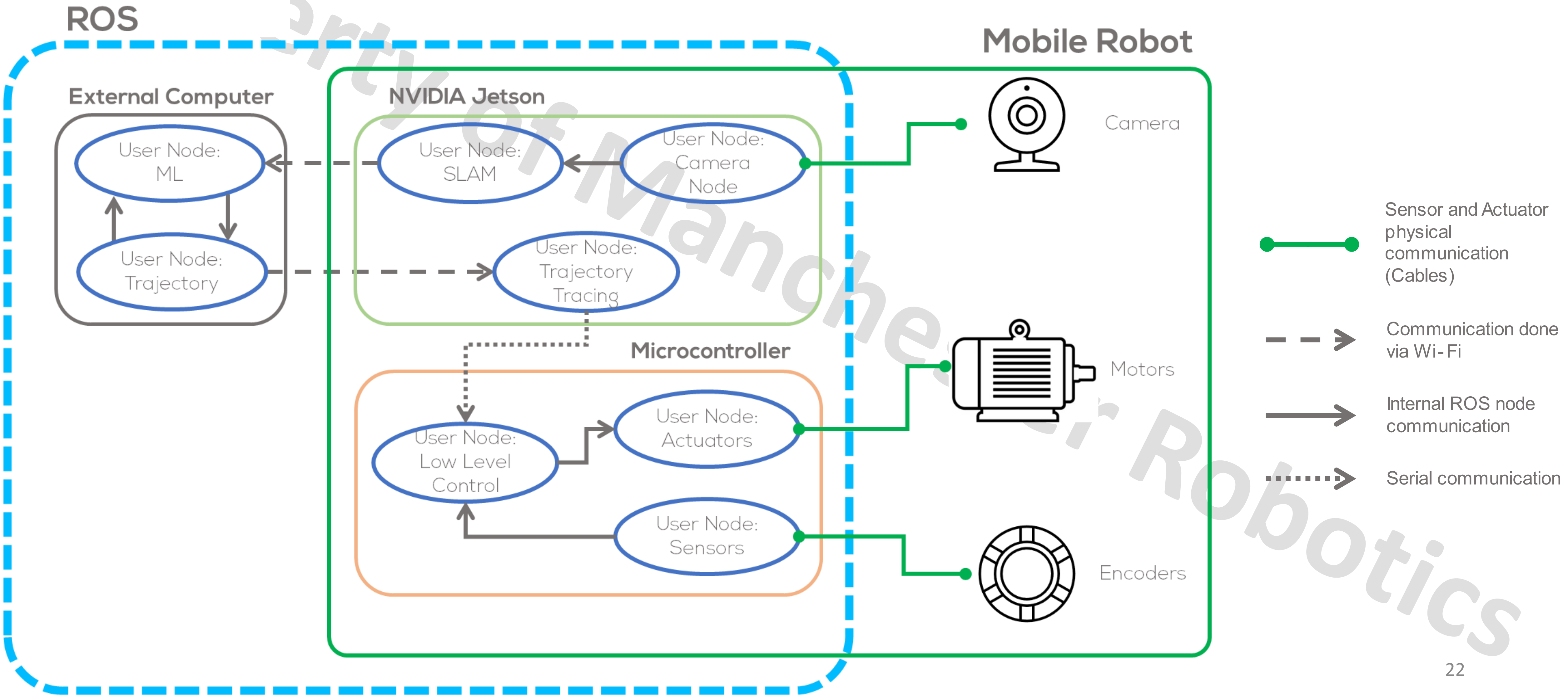
- The robot is controlled by a computer running a ROS node and communicated to the robot via Wi-Fi.
- The on-board computer runs Ubuntu as OS and uses ROS nodes to perform different tasks and communicate with the external computer, microcontroller and the peripherals.
- The microcontroller contains several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

ROS Example



Sensors: Camera, LiDAR, etc.
Wi-Fi
Serial
Actuators: Servo Motors, DC Motors, Stepper Motors
Sensors: Encoders, Reflectance, Sonar, etc.
External Computer
NVIDIA Jetson Nano
Puzzlebot Control Unit

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

Hands-on theoretical activity

`{Learn, Create, Innovate};`

MCR²

Manchester **Robotics**

Workspace

- ROS projects are organized in workspaces, which are a collection of grouped codes called packages.
- Commonly there is a *src* subdirectory. Inside that subdirectory is where the source code of ROS packages will be located. Typically, the directory starts empty.

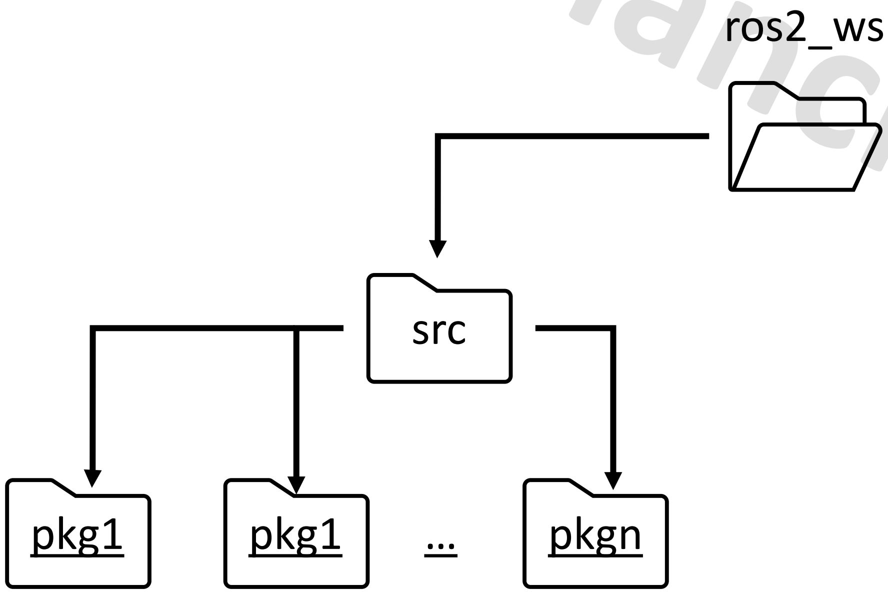
- *colcon* is an iteration on the ROS build tools *catkin_make*, *catkin_make_isolated*, *catkin_tools* and *ament_tools*.
- *colcon* does out of source builds. By default, it will create the *build*, *install* and *log* directories as peers of the *src* directory:

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.

Create a workspace

- Create a directory to contain our workspace
  - mkdir -p ~/ros2_cadi_ws/src
  - cd ~/ros2_cadi_ws



Build the workspace

- Before building a workspace make sure you are in the root of it.
  - cd ~/ros2_cadi_ws

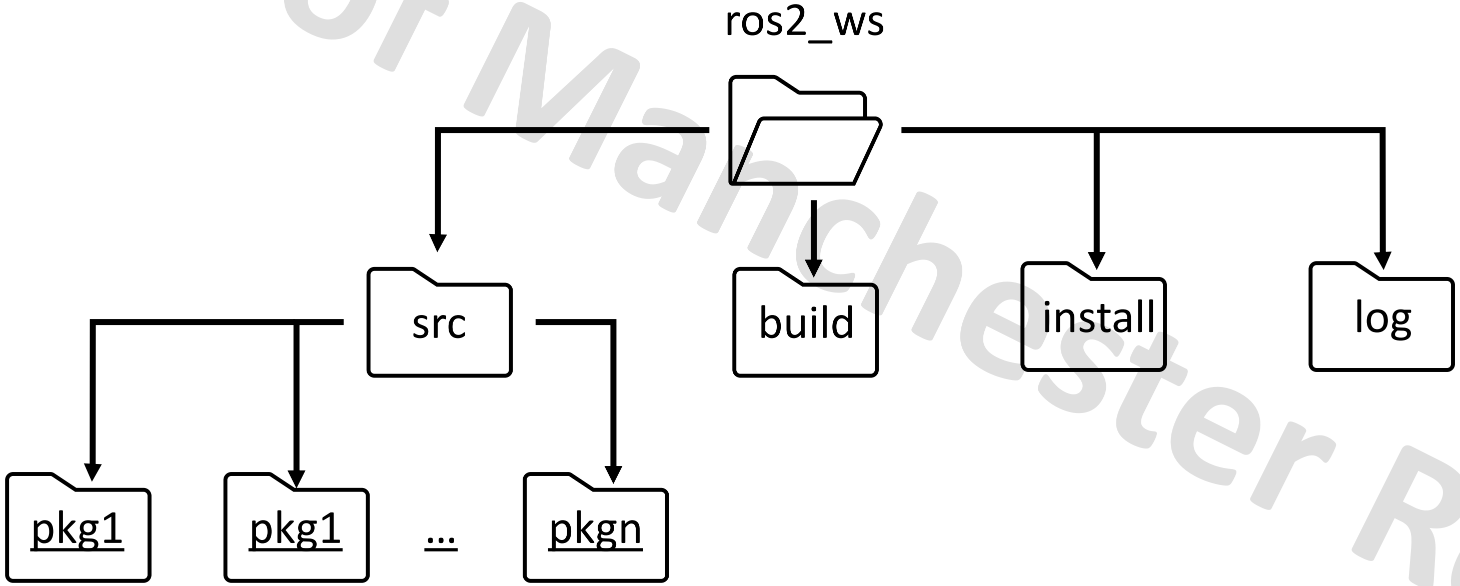
- colcon supports the option --symlink-install.
- This allows the installed files to be changed by changing the files in the source space (e.g. Python files or other not compiled resourced) for faster iteration.
  - colcon build

After building the workspace

- To run tests for the just built packages
  - colcon test

- 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

Workspace



ROS2 Package

- A package can be considered a container for ROS2 code.
- Packages allow to release ROS2 work and allow others to build and use it easily.
- Package creation in ROS2 uses ament as its build system and colcon as its build tool.
- CMake or Python can be used to create a package.

ROS2 CMake Package

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

Simplest file structure

- my_package/
-     CMakeLists.txt
-     package.xml

ROS2 Python Package

Minimum required content:

- 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

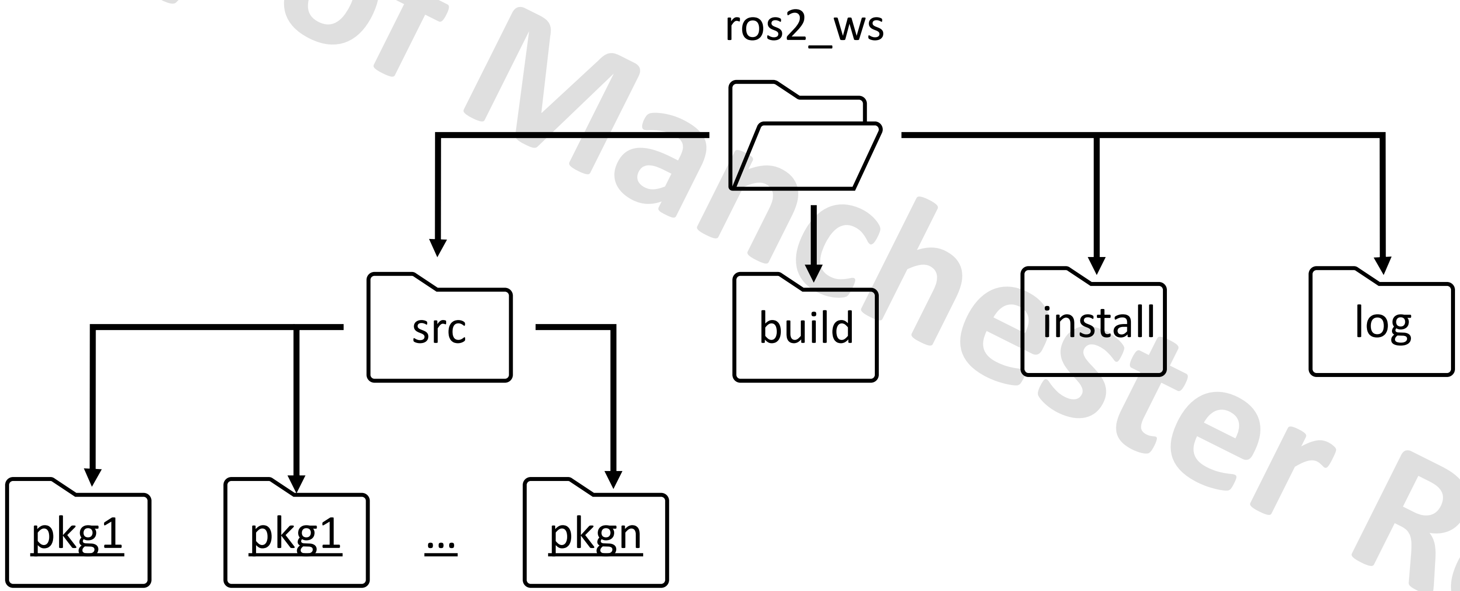
Simplest file structure

- my_package/
-     setup.py
-     package.xml
-     resource/my_package

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”.

Workspace



Create a new package

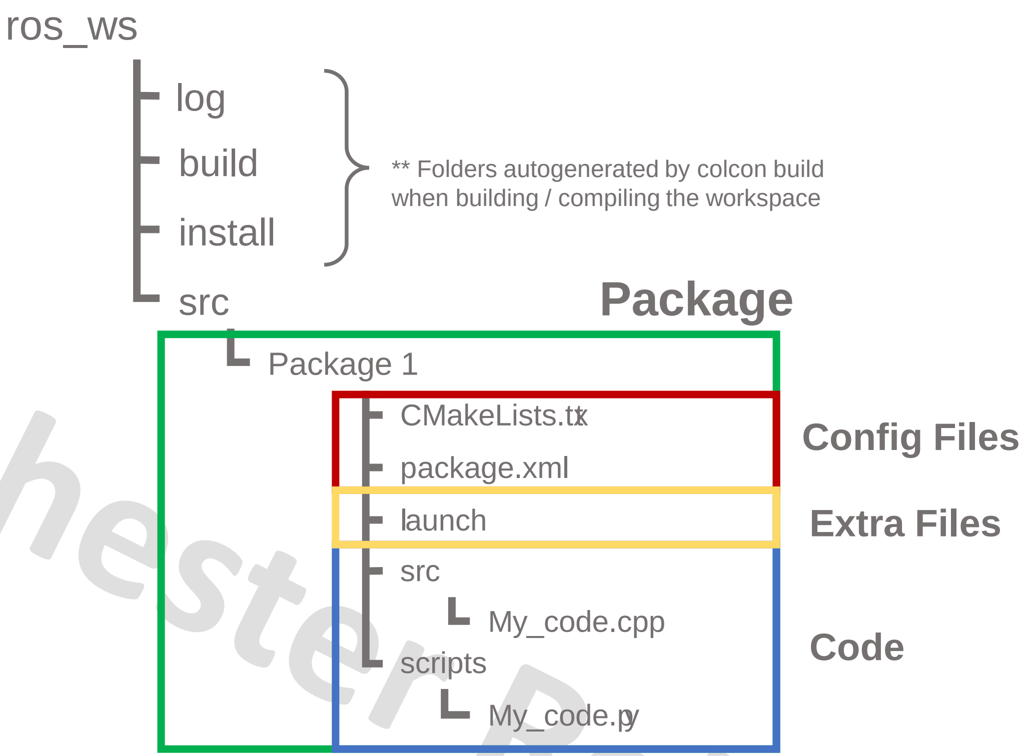
- To create a package
  - C++
  - ros2 pkg create --build-type ament_cmake <package_name>

  - Python
  - ros2 pkg create --build-type ament_python <package_name>

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.
- The preferred compilation tool is known as **colcon** and uses two separated files, **package.xml** and **CMakeLists.txt**.
- Instructions for the compiler need to be allocated in CMake and Package files.



ros_ws

log

build

install

src

** Folders autogenerated by colcon build when building / compiling the workspace

Package

Package 1

CMakeLists.txt

package.xml

launch

src

My_code.cpp

scripts

My_code.py

Config Files

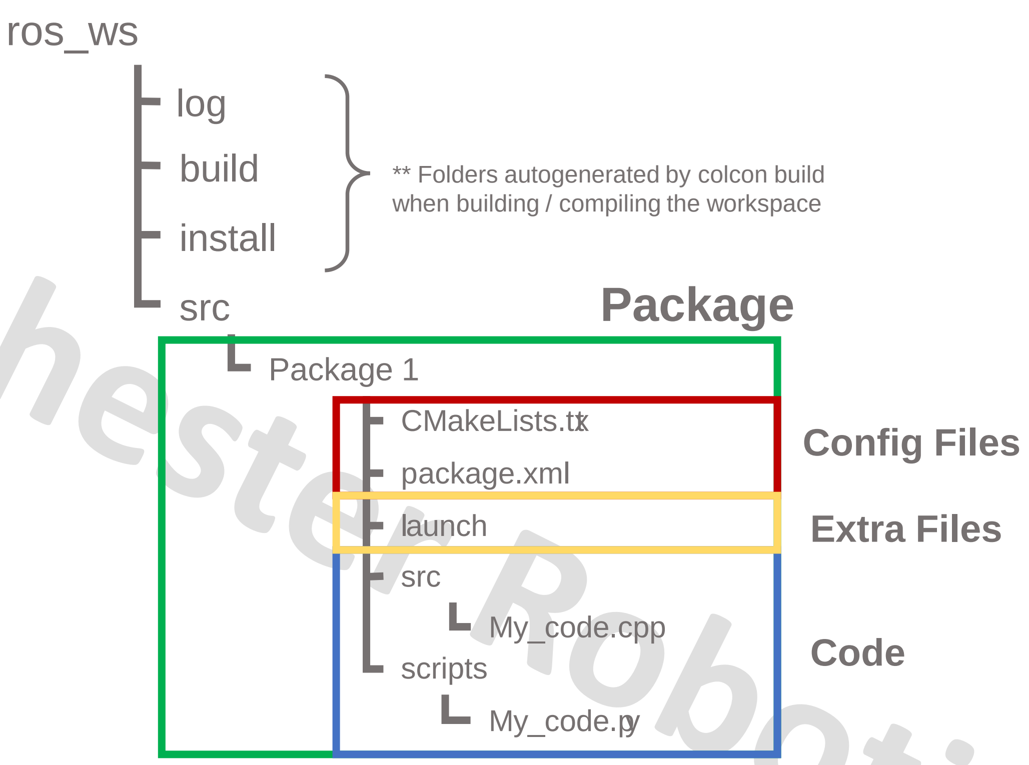
Extra Files

Code

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.



** Folders autogenerated by colcon build when building / compiling the workspace

ROS Activities

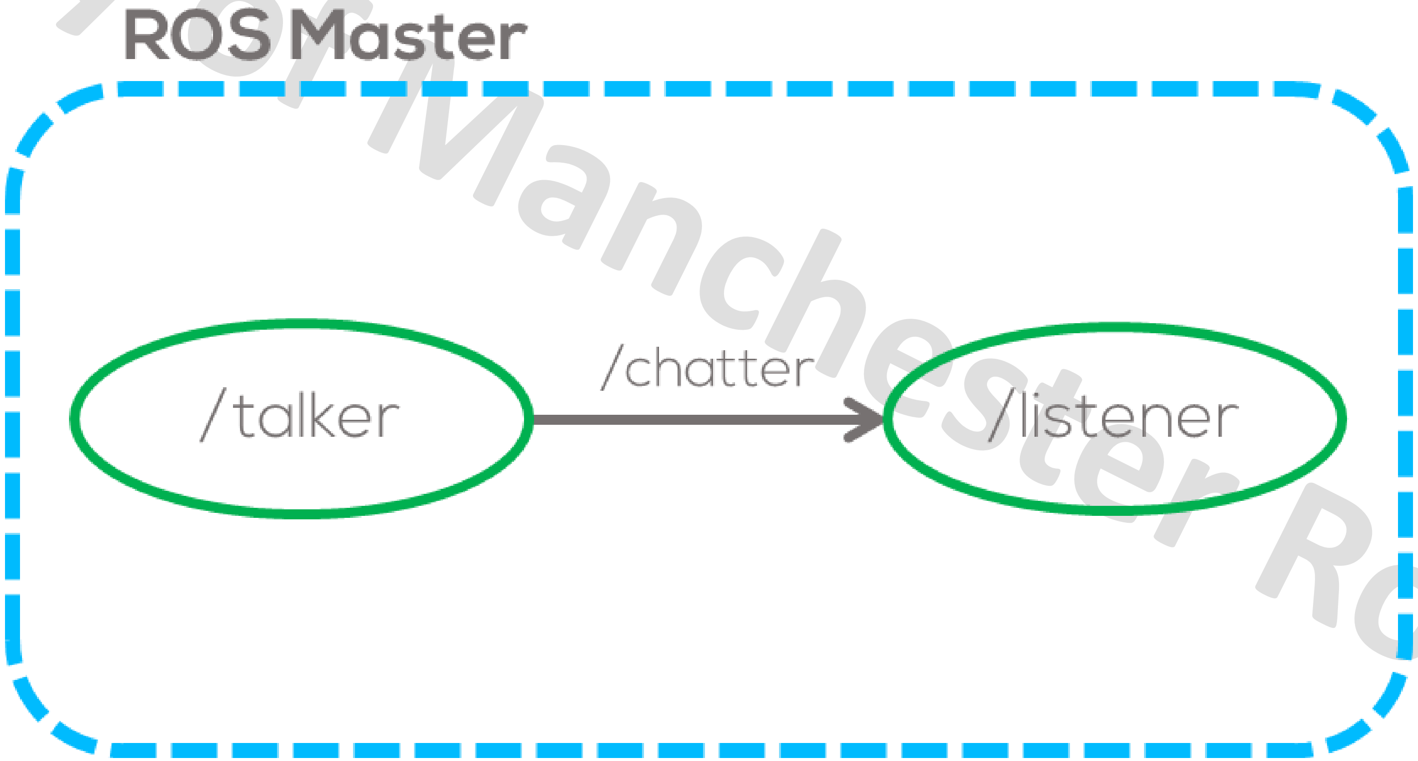
Activity #1 : Talker – Listener

{Learn, Create, Innovate};

MCR2 Manchester Robotics

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 Master
/talker
/chatter
/listener

ROS Activity 1 – Requirements

- Ubuntu in VM (MCR2 VM) or dual booting
- ROS installed (if not follow the steps in this link and select full installation)
- Workspace “ros2_ws” created following the steps here (if you are using the VM this is already done for you).

ROS Activity 1 – create package

- Create a package called *basic_comms*. Open a terminal and type the following
  - ros2 pkg create --build-type ament_python --node-name talker basic_comms
  - Beware that the command 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
  - colcon build

ROS Activity 1 – validate package

- Source the current workspace
  - source install/setup.bash

- Check the executable that was recently created
  - ros2 run basic_comms talker

Talker node

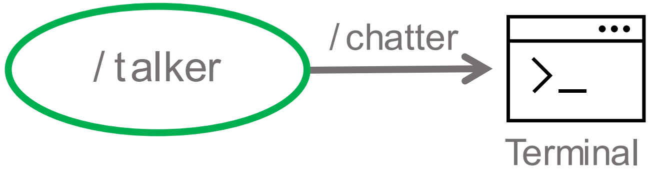
- Create the publisher node file in ~/ros2_ws/src/basic_comms/basic_comms/src
  - touch talker.py
  - chmod +x talker.py

- 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, …)

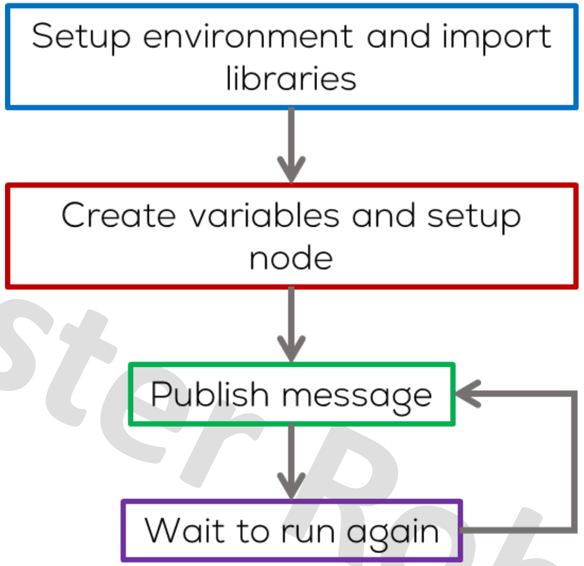
ROS Activity 1

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.


Setup environment and import libraries

Create variables and setup node

Publish message

Wait to run again

- Your code will be written in the file "talker.py".
- Use any word processor or IDE (nano, VS Code, Vim, etc.) to open the file.

OOP - Talker node

Imports

```
import rclpy
from rclpy.node import Node

from std_msgs.msg import String
```

Constructor

```
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_callback)
        self.i = 0
```

Timer callback

```
    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
```

Main

```
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()
```

Execute code

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

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>
```

Add an entry point

- Open the setup.py file

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

ROS Exercise 1 – 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
```

Listener node

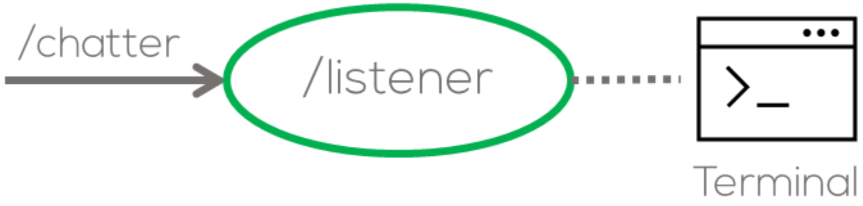
- Create the publisher node file in ~/ros2_ws/src/basic_comms/basic_comms/src
  - touch listener.py
  - chmod +x listener.py

- 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, …)

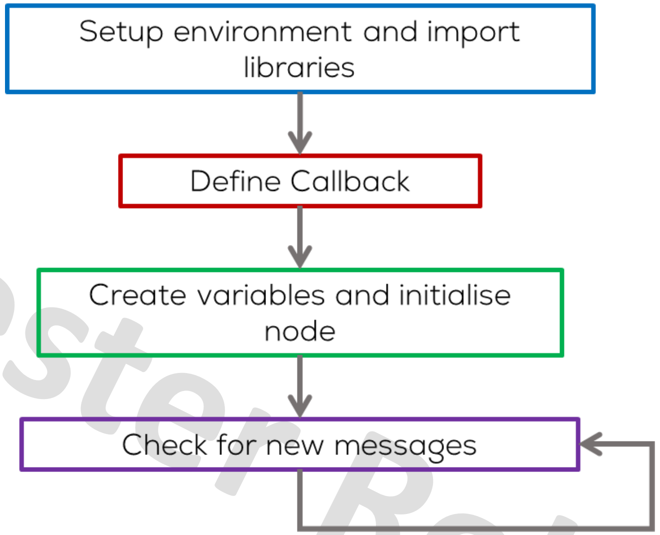
ROS Activity 1

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 thefollowing.


Setup environment and import libraries
Define Callback
Create variables and initialise node
Check for new messages

- Your code will be written in the file "listener.py".
- Use any word processor or IDE (nano, VS Code, Vim, etc.) to open the file .

OOP - Listener node

Imports

```
import rclpy
from rclpy.node import Node

from std_msgs.msg import String
```

Constructor

```
class MinimalSubscriber(Node):

    def __init__(self):
        super().__init__('listener_node')
        self.subscription = self.create_subscription(
            String,
            'chatter',
            self.listener_callback,
            10)
        self.subscription  # prevent unused variable warning
```

Timer callback

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

Main

```
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()
```

Execute code

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

ROS Exercise 1 – 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 listener
```

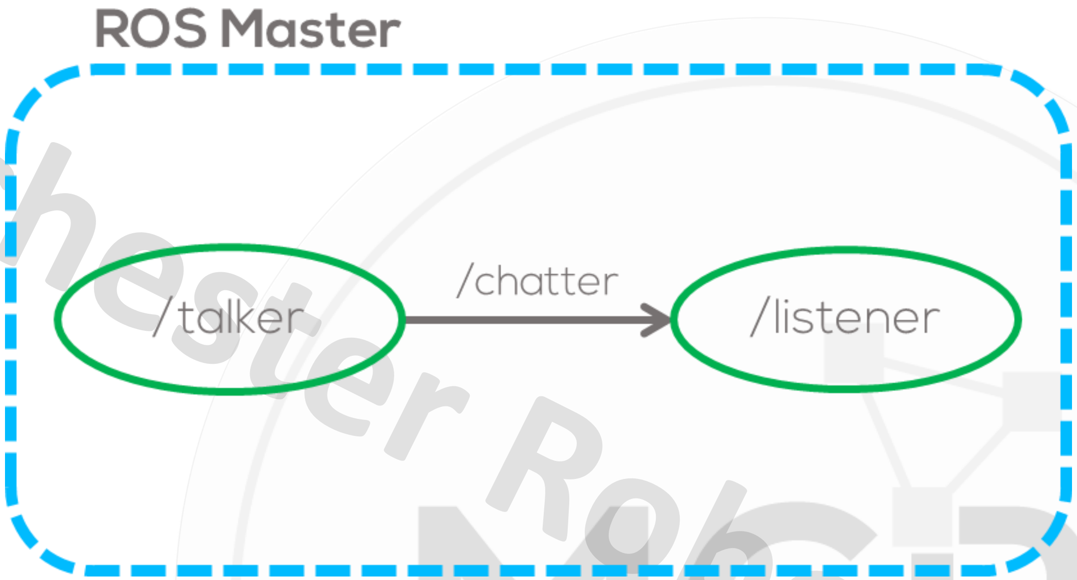
- To visualize the output of the node, open another terminal and use the command “echo” as follows

```
$ ros2 topic pub --once /chatter std_msgs/msg/String "data: hola"
```

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

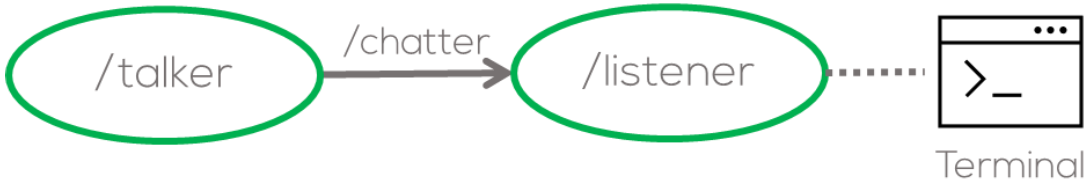
- Open a new terminal and run the talker node you just made using the following command

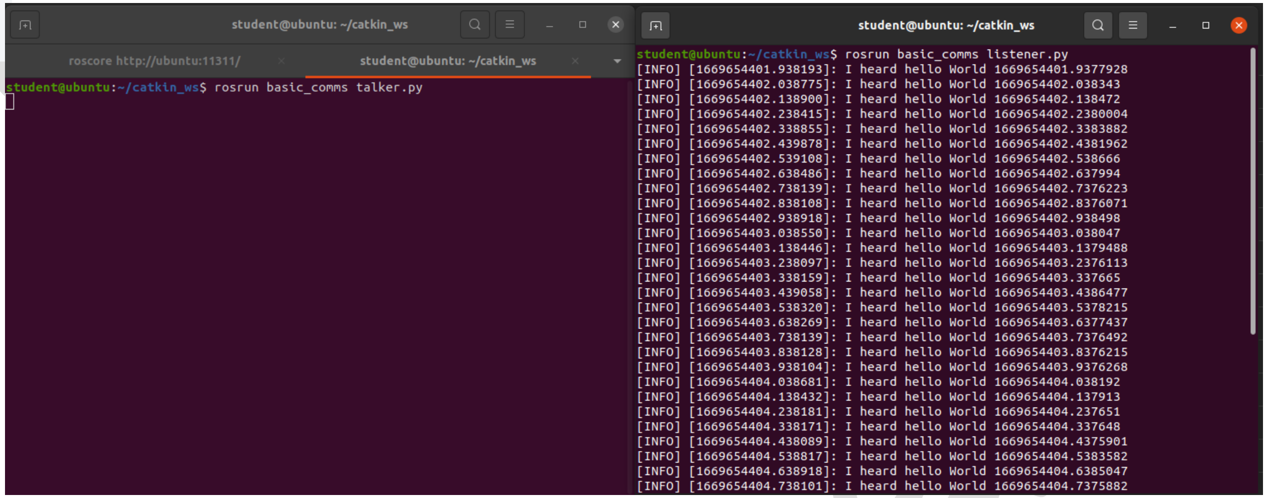
```
$ source install/setup.bash
$ rosrun basic_comms talker.py
```

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

```
$ source install/setup.bash
$ rosrun basic_comms listener.py
```

Results




student@ubuntu: ~/catkin_ws
roscore http://ubuntu:11311/
student@ubuntu: ~/catkin_ws
student@ubuntu: ~/catkin_ws

```
student@ubuntu:~/catkin_ws$ rosrun basic_comms talker.py
```

```
student@ubuntu:~/catkin_ws$ rosrun basic_comms listener.py
[INFO] [1669654401.938193]: I heard hello World 1669654401.9377928
[INFO] [1669654402.038775]: I heard hello World 1669654402.038343
[INFO] [1669654402.138900]: I heard hello World 1669654402.138472
[INFO] [1669654402.238415]: I heard hello World 1669654402.2380004
[INFO] [1669654402.338855]: I heard hello World 1669654402.3383882
[INFO] [1669654402.439878]: I heard hello World 1669654402.4381962
[INFO] [1669654402.539108]: I heard hello World 1669654402.538666
[INFO] [1669654402.638486]: I heard hello World 1669654402.637994
[INFO] [1669654402.738139]: I heard hello World 1669654402.7376223
[INFO] [1669654402.838108]: I heard hello World 1669654402.8376071
[INFO] [1669654402.938918]: I heard hello World 1669654402.938498
[INFO] [1669654403.038550]: I heard hello World 1669654403.038047
[INFO] [1669654403.138446]: I heard hello World 1669654403.1379488
[INFO] [1669654403.238097]: I heard hello World 1669654403.2376113
[INFO] [1669654403.338159]: I heard hello World 1669654403.337665
[INFO] [1669654403.439058]: I heard hello World 1669654403.4386477
[INFO] [1669654403.538320]: I heard hello World 1669654403.5378215
[INFO] [1669654403.638269]: I heard hello World 1669654403.6377437
[INFO] [1669654403.738139]: I heard hello World 1669654403.7376492
[INFO] [1669654403.838128]: I heard hello World 1669654403.8376215
[INFO] [1669654403.938104]: I heard hello World 1669654403.9376268
[INFO] [1669654404.038681]: I heard hello World 1669654404.038192
[INFO] [1669654404.138432]: I heard hello World 1669654404.137913
[INFO] [1669654404.238181]: I heard hello World 1669654404.237651
[INFO] [1669654404.338171]: I heard hello World 1669654404.337648
[INFO] [1669654404.438089]: I heard hello World 1669654404.4375901
[INFO] [1669654404.538817]: I heard hello World 1669654404.5383582
[INFO] [1669654404.638918]: I heard hello World 1669654404.6385047
[INFO] [1669654404.738101]: I heard hello World 1669654404.7375882
```

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

Robot Operating System - ROS

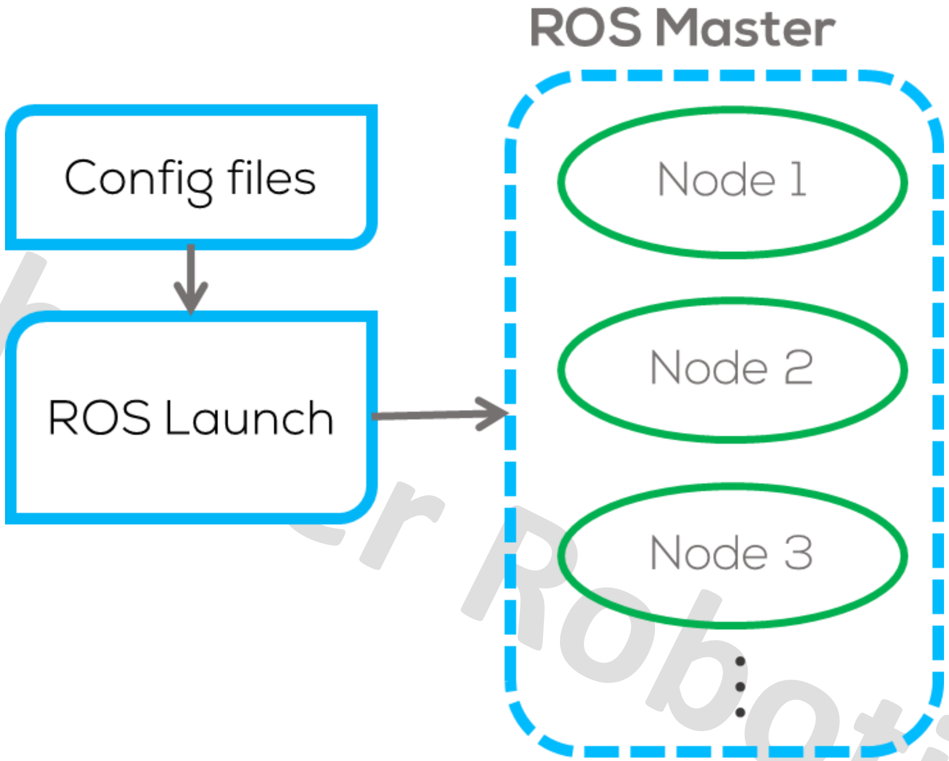
ROS Launch Files

`{Learn, Create, Innovate};`

MCR²
Manchester **Robotics**

ROS2 Launch file

- Launch files are sets of commands written in Python, xml, and yaml that allow executing various scripts at the same time.
- Launch files allow to run any object used within the ROS2 architecture and has a wide variety of tools that allow to parametrize the launch file so that it can be adapted to the requirements of a project.



Launch files

- Launch files allow you to start up and configure a number of 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>

ROS2 Launch file

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

ROS Activity 1(Launch file)

- For exercise 1, it is possible to develop a ROS Launch file as follows.

```
$ cd ~/ros2_ws/src/basic_comms
$ mkdir launch
$ cd launch
$ touch chatter_launch.py
$ chmod +x chatter_launch.py
```

- Your code will be written in the file “activity1.launch”.
- For this exercise, the talker and listener nodes will be launched.
- The following code will launch the two nodes previously done (talker and listener).

ROS2 Launch file

```python
import os
from ament_index_python import get_package_share_directory
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch_ros.actions import Node
```

Imports

```python
def generate_launch_description():

    talker_node = Node(
            package='basic_comms',
            executable='talker',
            output='screen',
        )

    listener_node = Node(
            package='basic_comms',
            executable='listener',
            output='screen',
        )
```

Launch body

```python
    l_d = LaunchDescription([talker_node, listener_node])

    return l_d
```

Set launch content

Dependencies

- Add the following line to the package.xml file
  - `<exec_depend>ros2launch</exec_depend>`

- Add the following line to the setup.py file
  - Make sure it is located inside `data_files`

  -
    ```
    (os.path.join('share', package_name, 'launch'), glob(os.path.join('launch', '*launch.[pxy][yma]*')))
    ```
  - Do not forget to include the following lines at the top
    - `import os`
    - `from glob import glob`

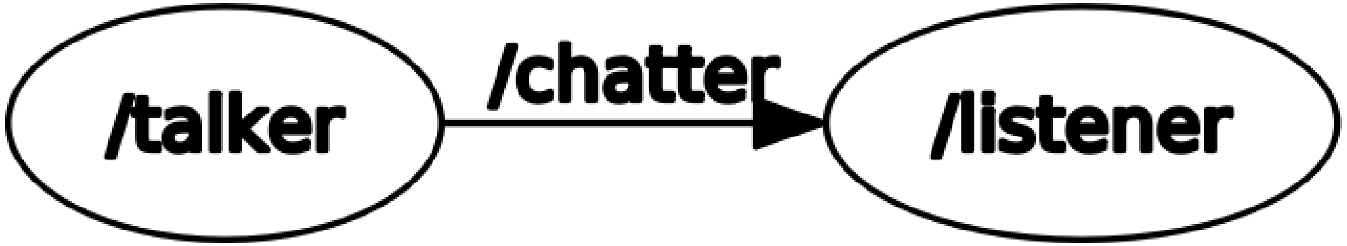
ROS Activity 1 (Launch file)

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

```
$ colcon build
$ source install/setup.bash
$ roslaunch basic_comms chatter.launch.py
```

- 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};

MCR^2

Manchester **Robotics**

Thank You

Robotics For Everyone

{Learn, Create, Innovate};

MCR^2

Manchester **Robotics**

T&C

Terms and conditions

{Learn, Create, Innovate};

MCR2 Manchester Robotics

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.*