{Learn, Create, Innovate};

Robot Operating System - ROS

Introduction

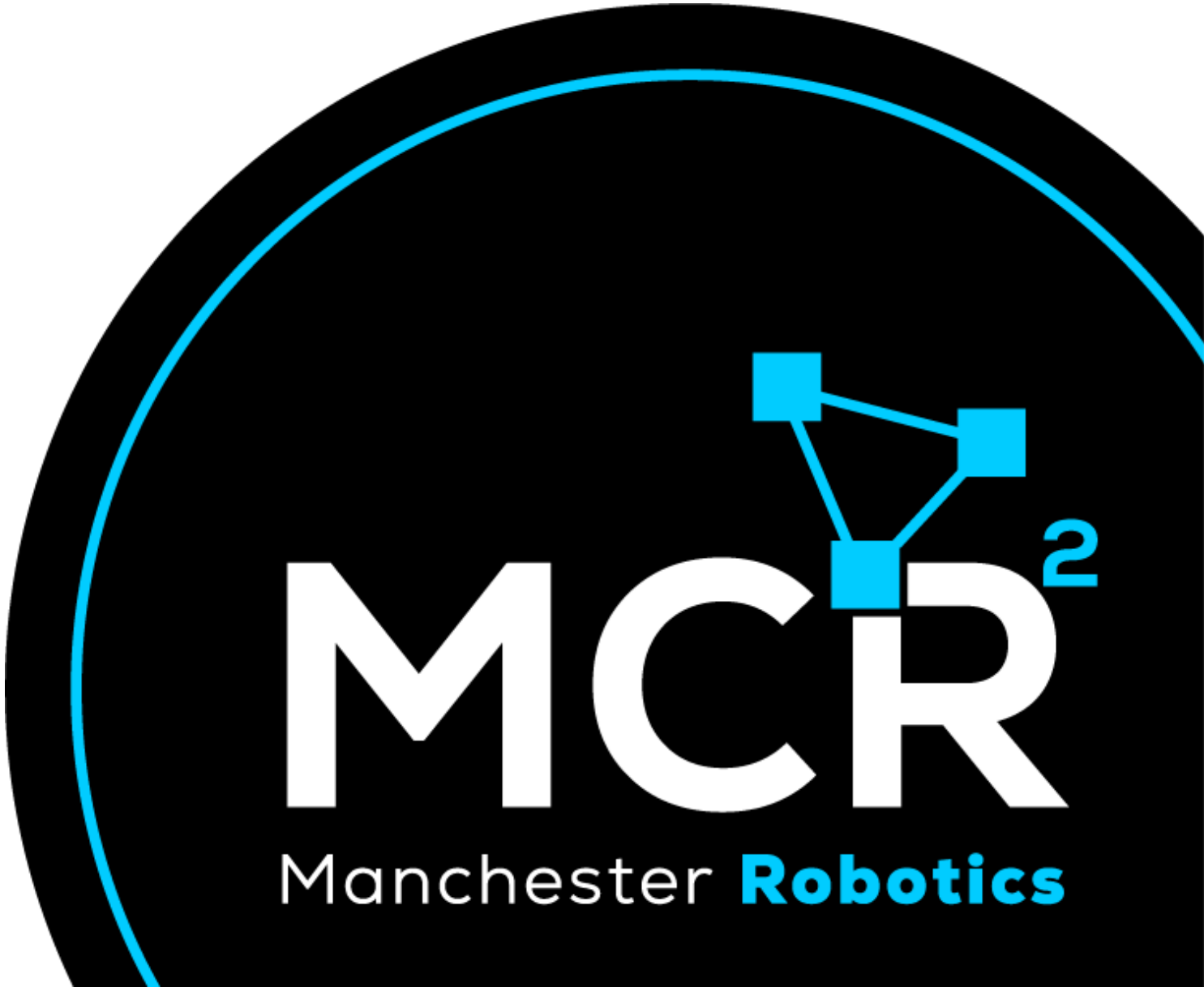
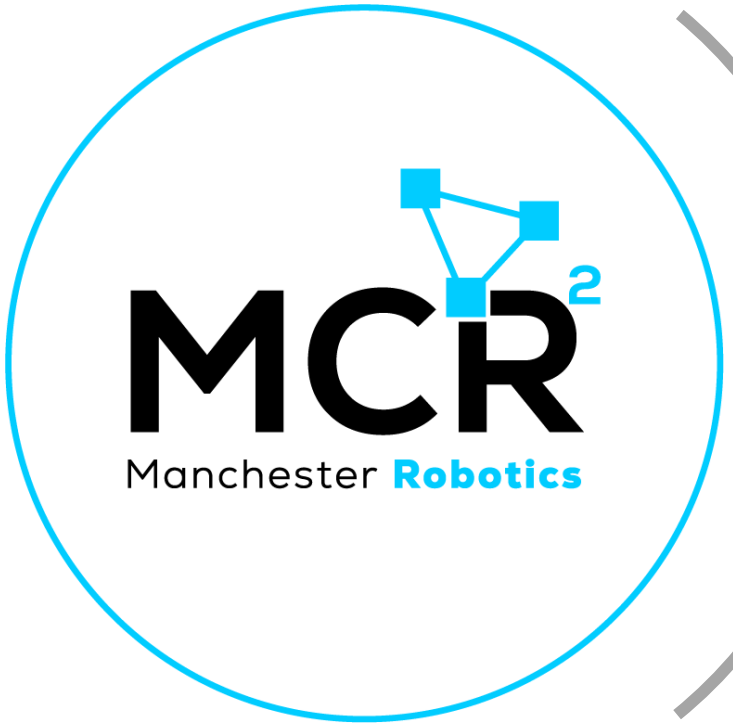


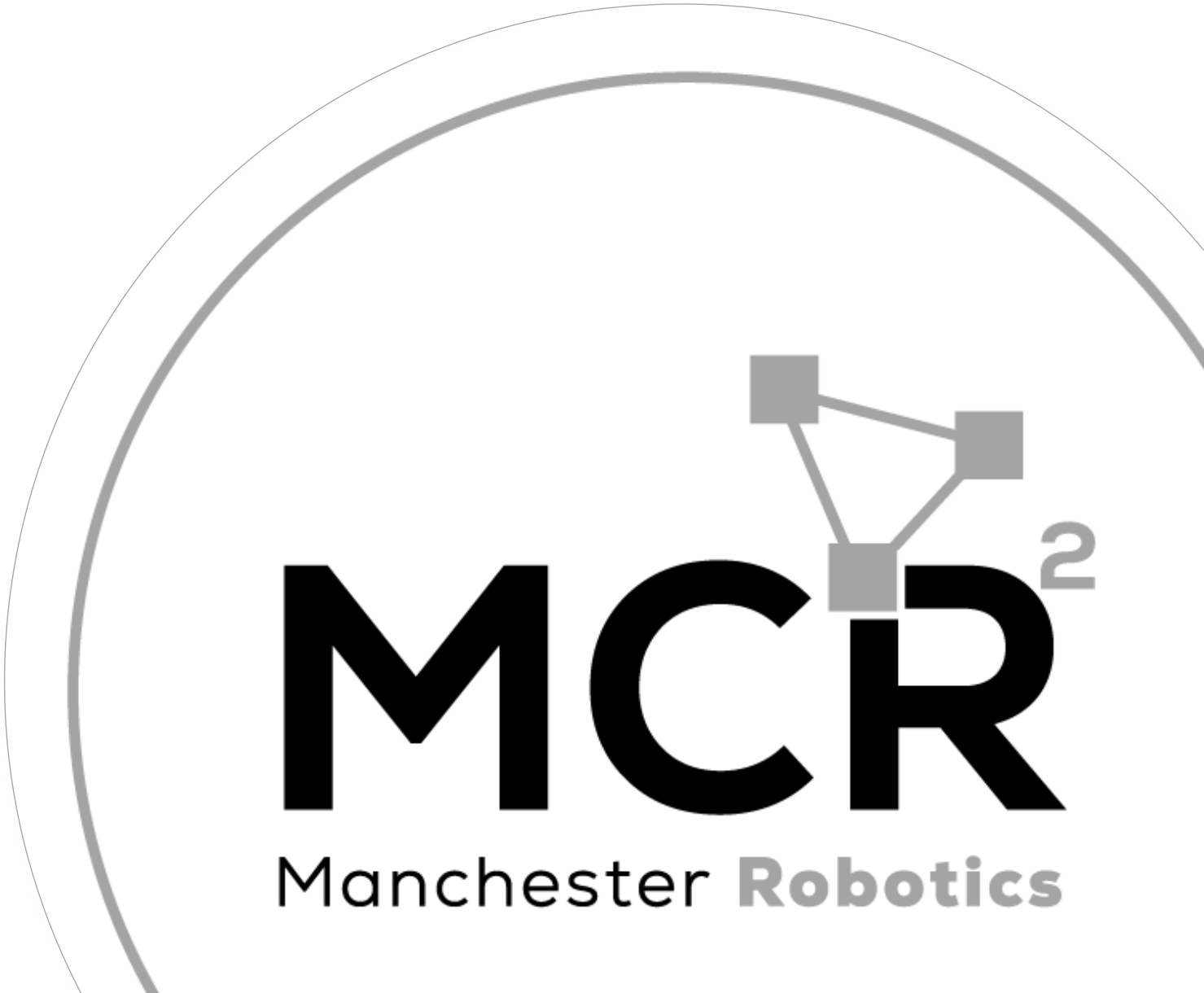
Table of contents



Robot Operating System - ROS

What is ROS?

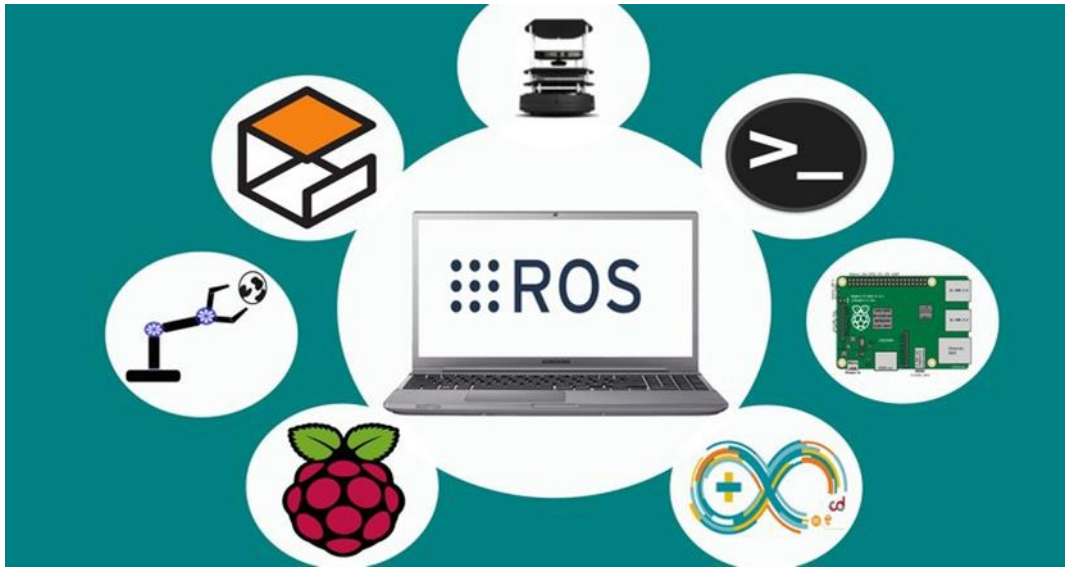
{Learn, Create, Innovate};



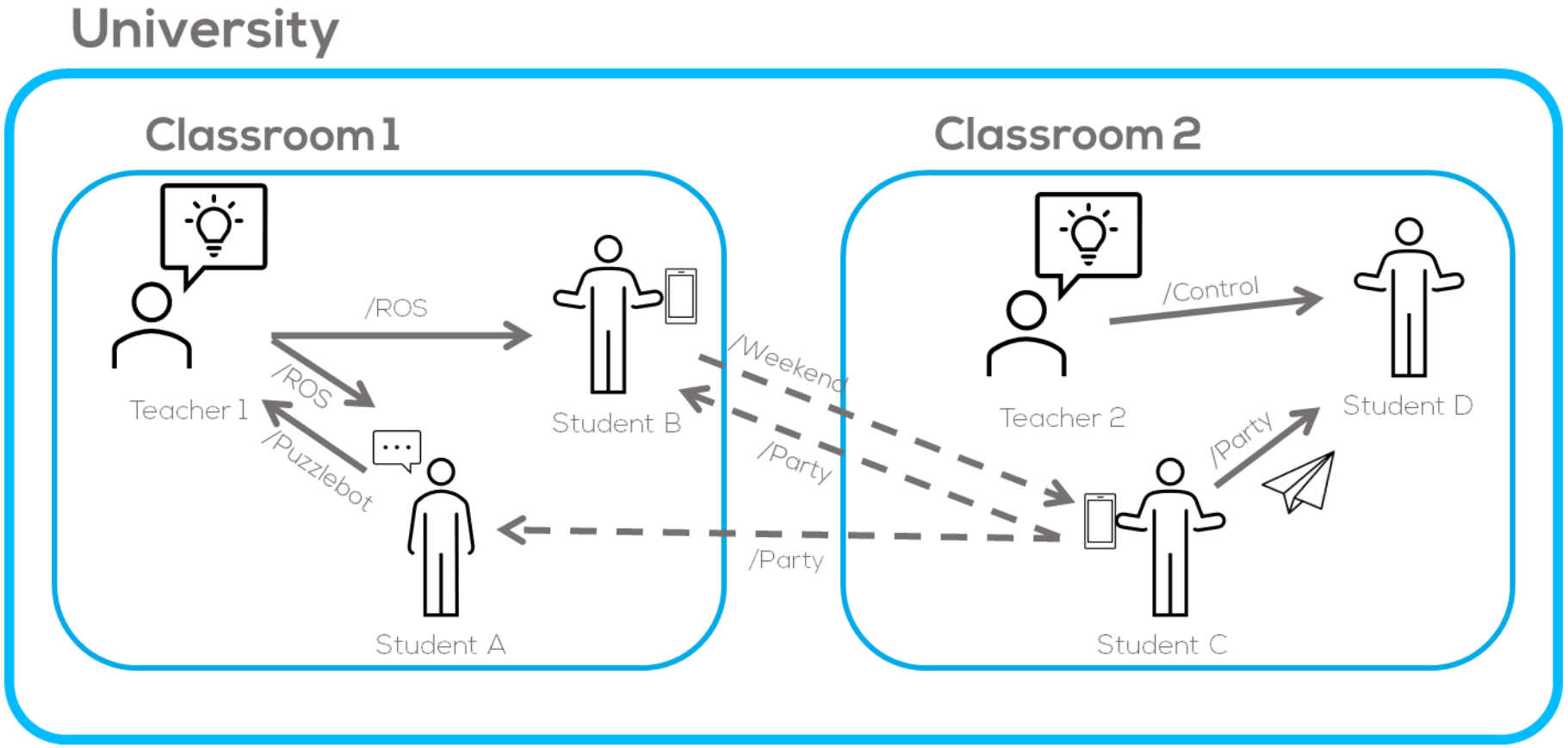
What is ROS?

What is ROS?

“ROS is an open-source framework that helps researchers and developers build and reuse code between robotics applications”



ROS Basics

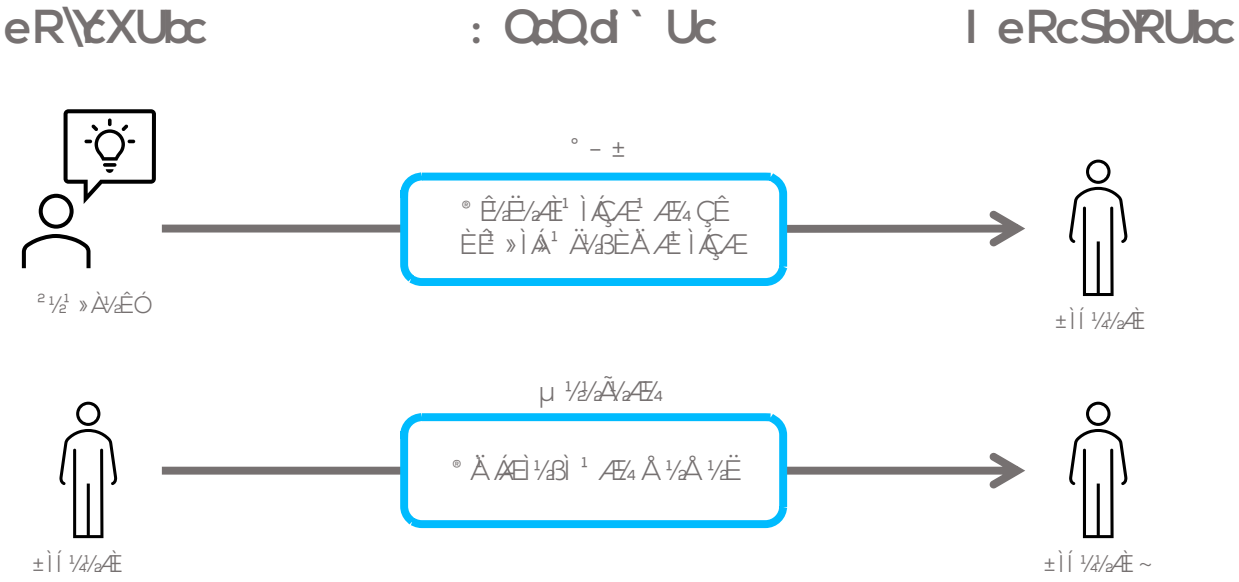


** Any resemblance to reality, is purely coincidental.

ROS Basics

How is the information delivered?

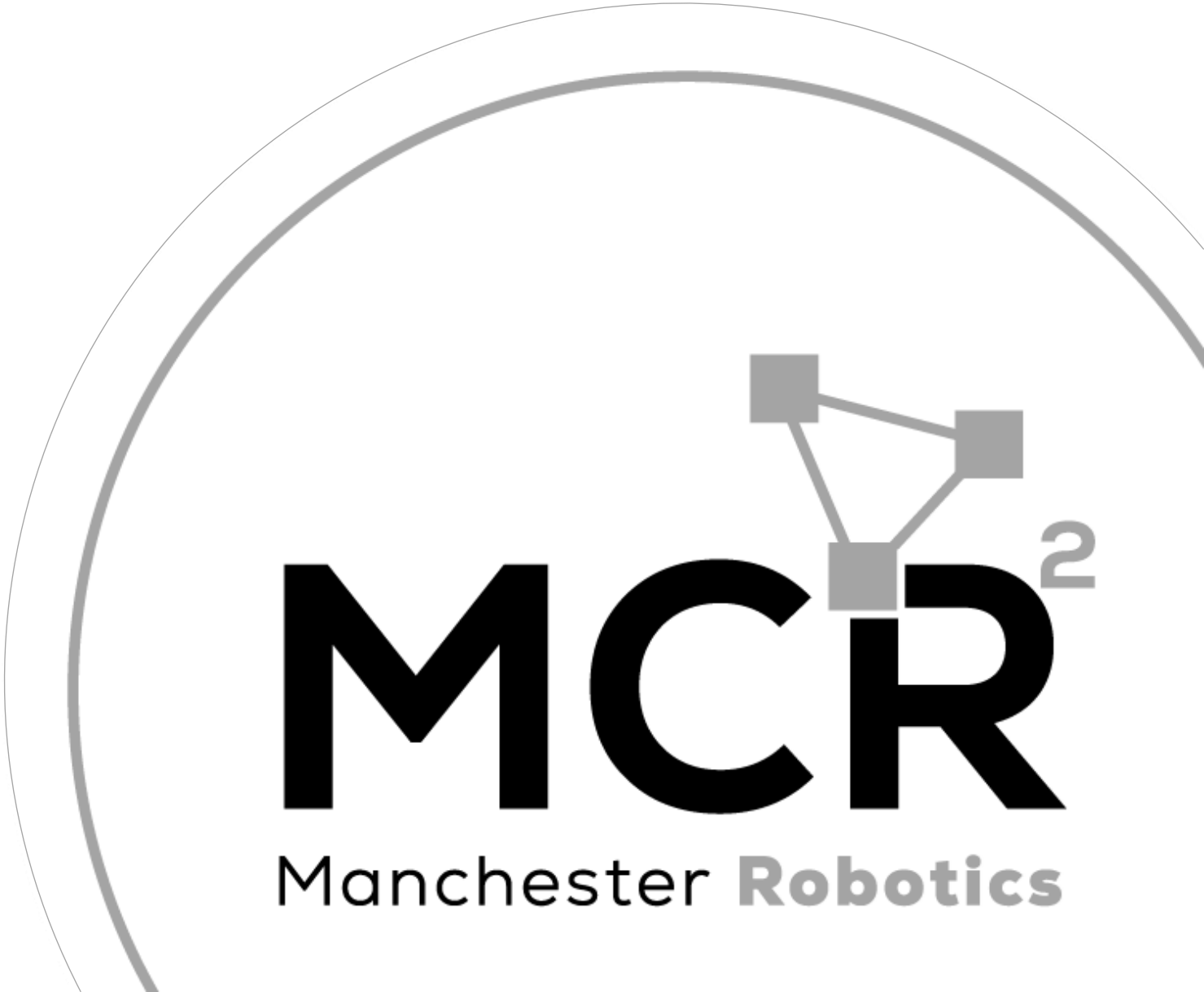
- The information is delivered through messages inside each topic.
- Any message or class has a certain format (is encoded), which both the teacher and the student know off and is expected.
- As an example: Between two students, it is expected some simple messages such as plain text, memes or figures.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



Robot Operating System - ROS

ROS Architecture

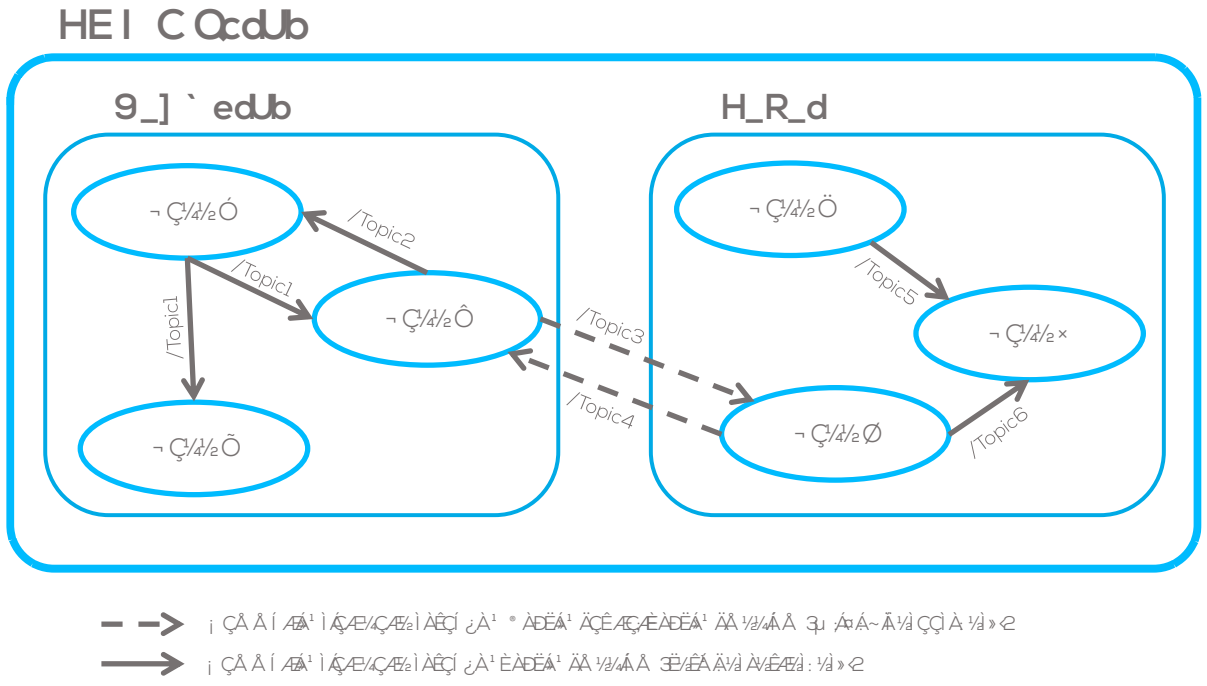
{Learn, Create, Innovate};



ROS Architecture

ROS Master

- Is the framework that allows ROS to work.
- Process Manager that enables the communication between different agents in the network.
- Allows communication between different nodes, in different computers or robots by means of a Client-Server architecture.
- On the previous example, the University and the set of “rules” allow communication between agents.



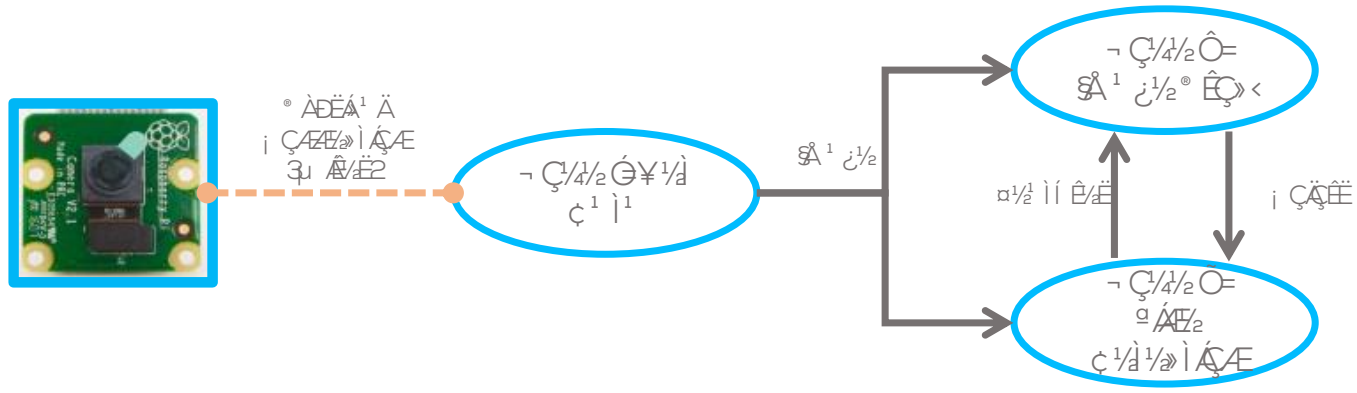
ROS Architecture

ROS Nodes

- 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

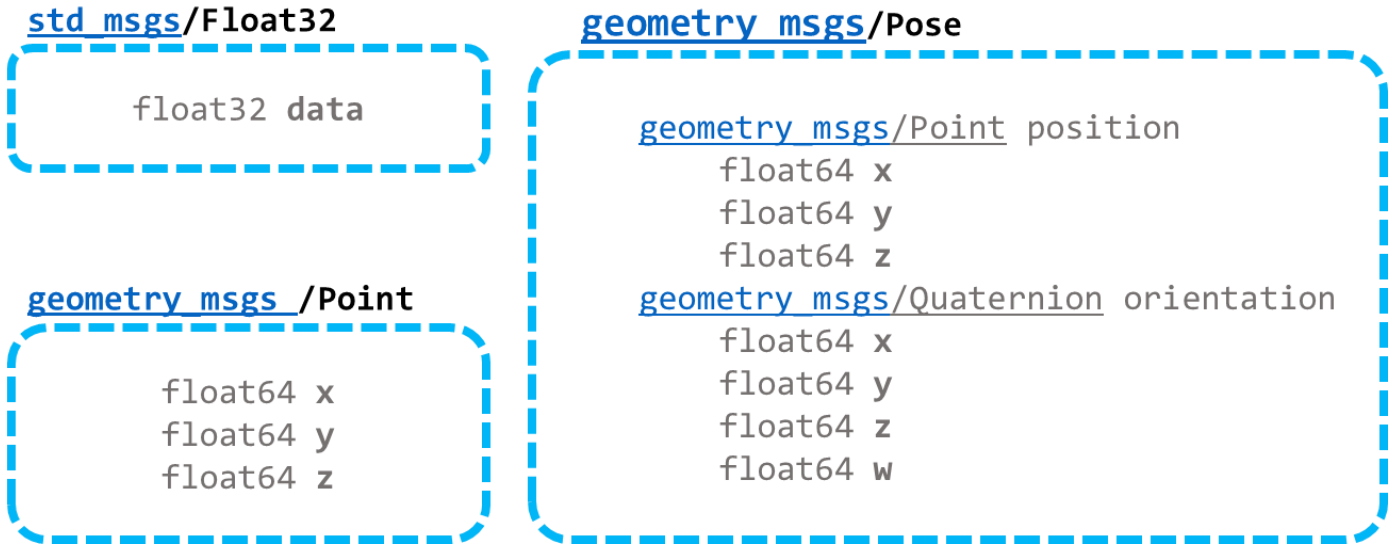
- Topics are named buses over which nodes exchange messages.
- Topics are intended for unidirectional, streaming communication.
- There can be multiple publishers and subscribers to a topic.



ROS Architecture

Messages

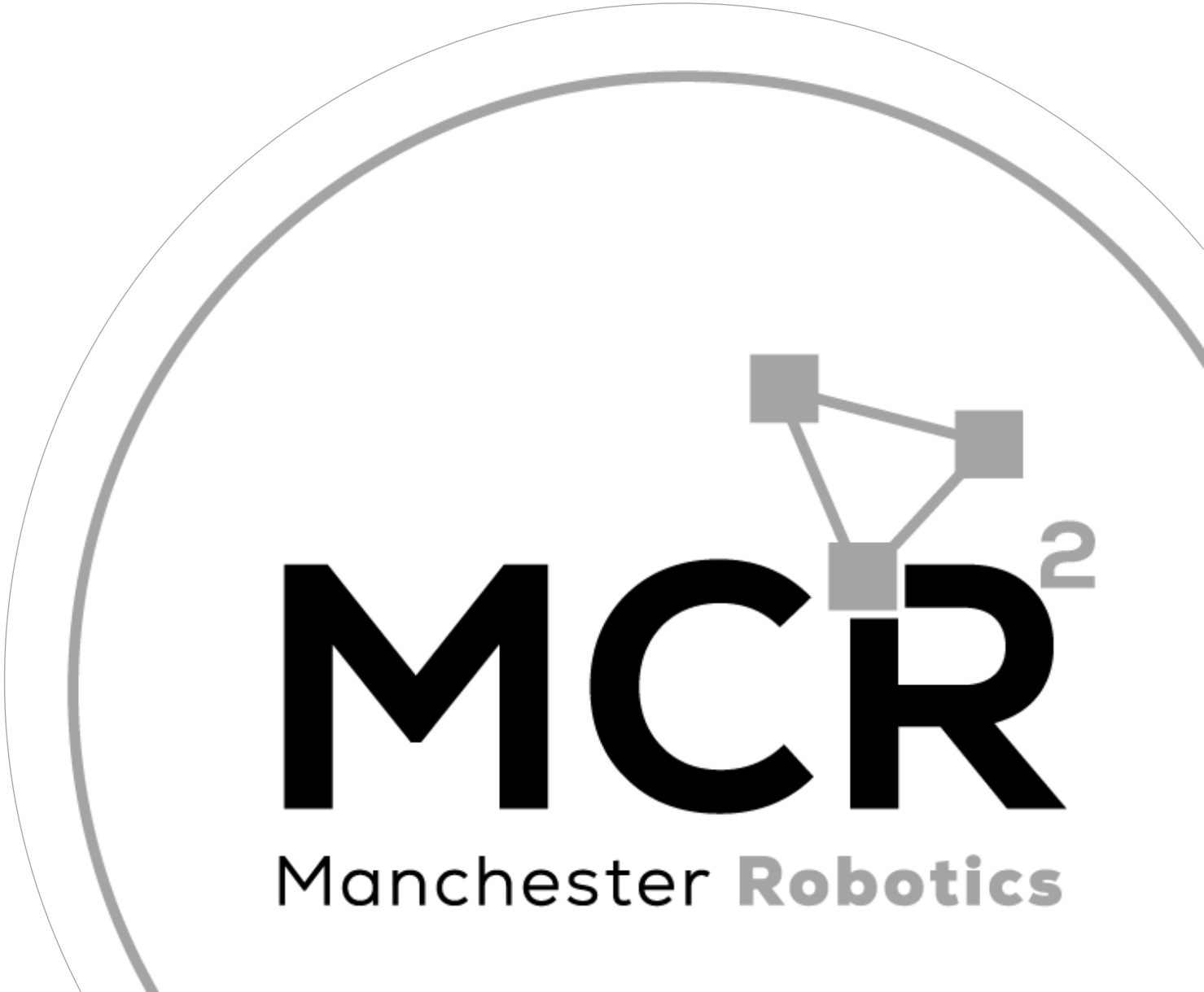
- Nodes communicate with each other by publishing messages to topics.
- A message is a simple data structure, comprising typed fields.
- Many types of data are supported such as integers, floating point, Boolean, etc.
- Messages can include nested structures and arrays.



Robot Operating System - ROS

ROS Example

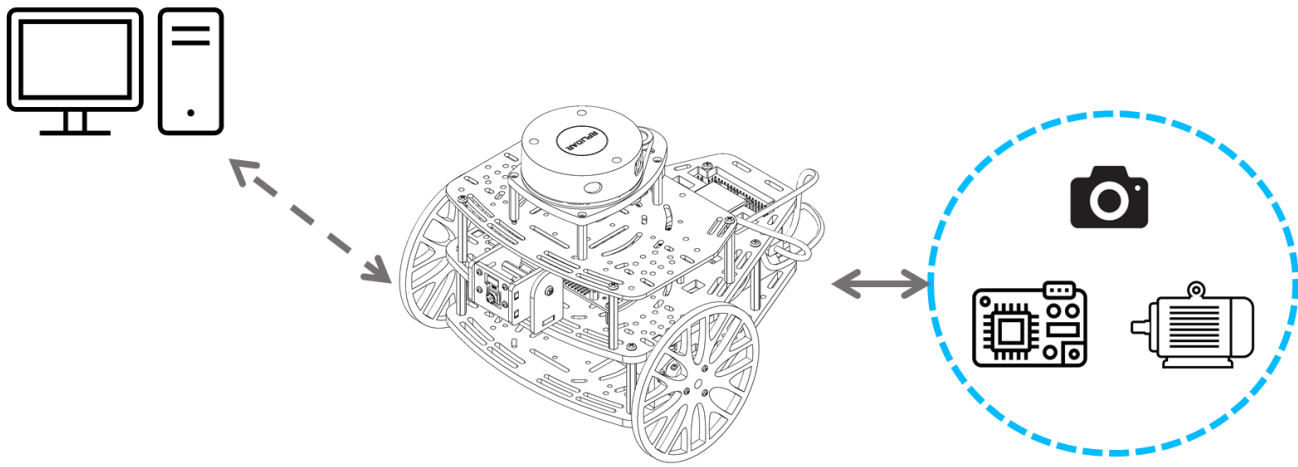
{Learn, Create, Innovate};



ROS Example

A Practical Example

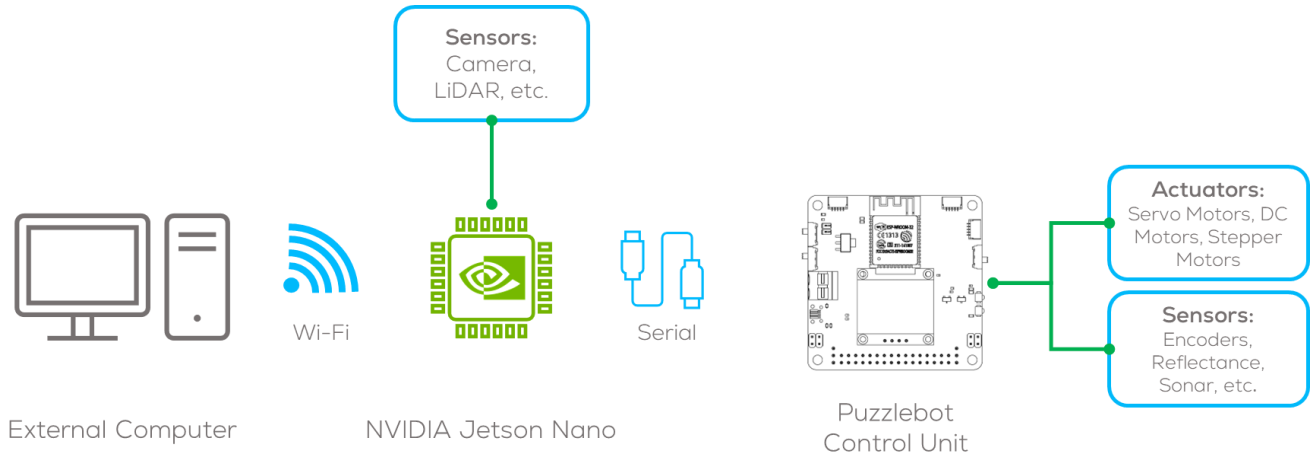
- In this example we are going to analyze a real system and how it can be controlled using ROS.
- The system presented is a mobile robot called Puzzlebot® by Manchester Robotics.
- The robot is comprised of an on-board computing unit from NVIDIA the Jetson Nano used for high level control algorithms and a microcontroller for low level control tasks.
- The robot also contains different sensors and actuators such as motors, encoders, and cameras.



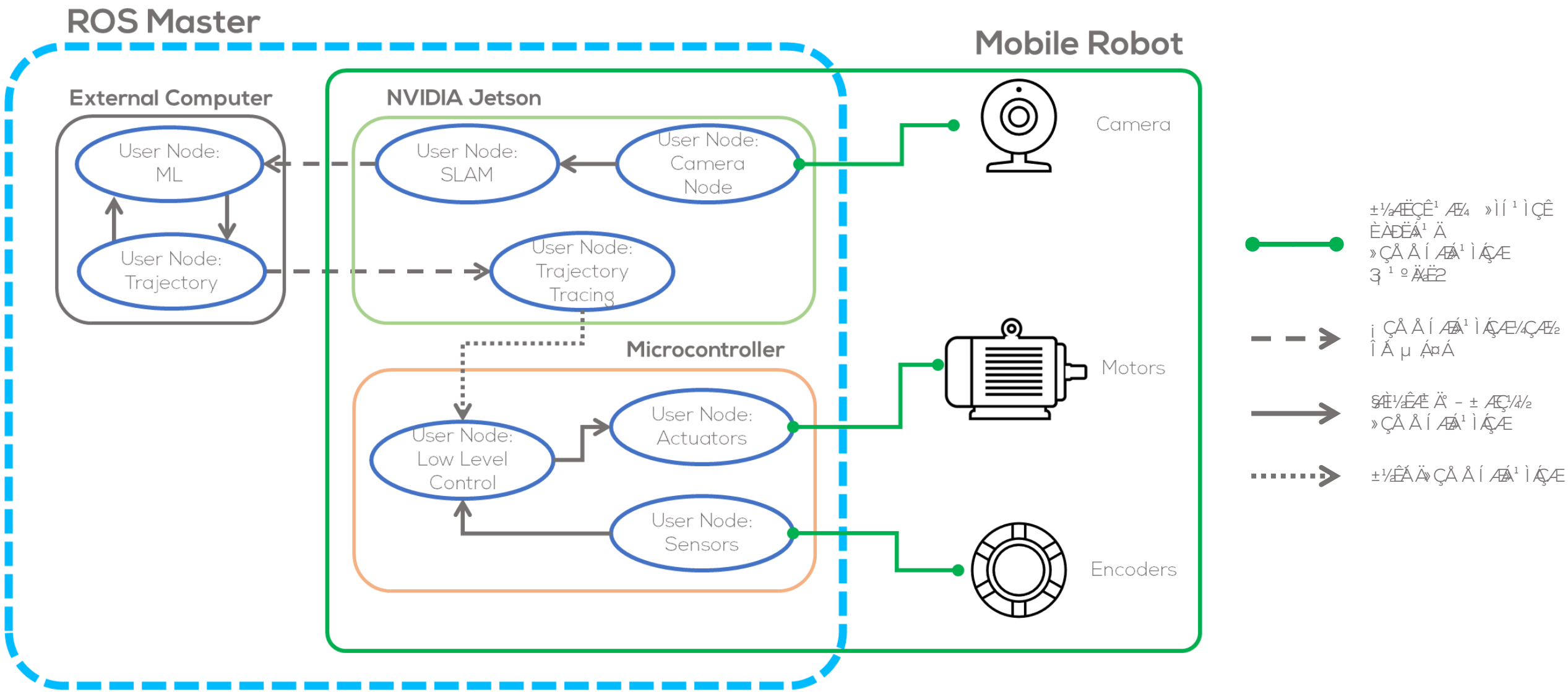
ROS Example

A 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.



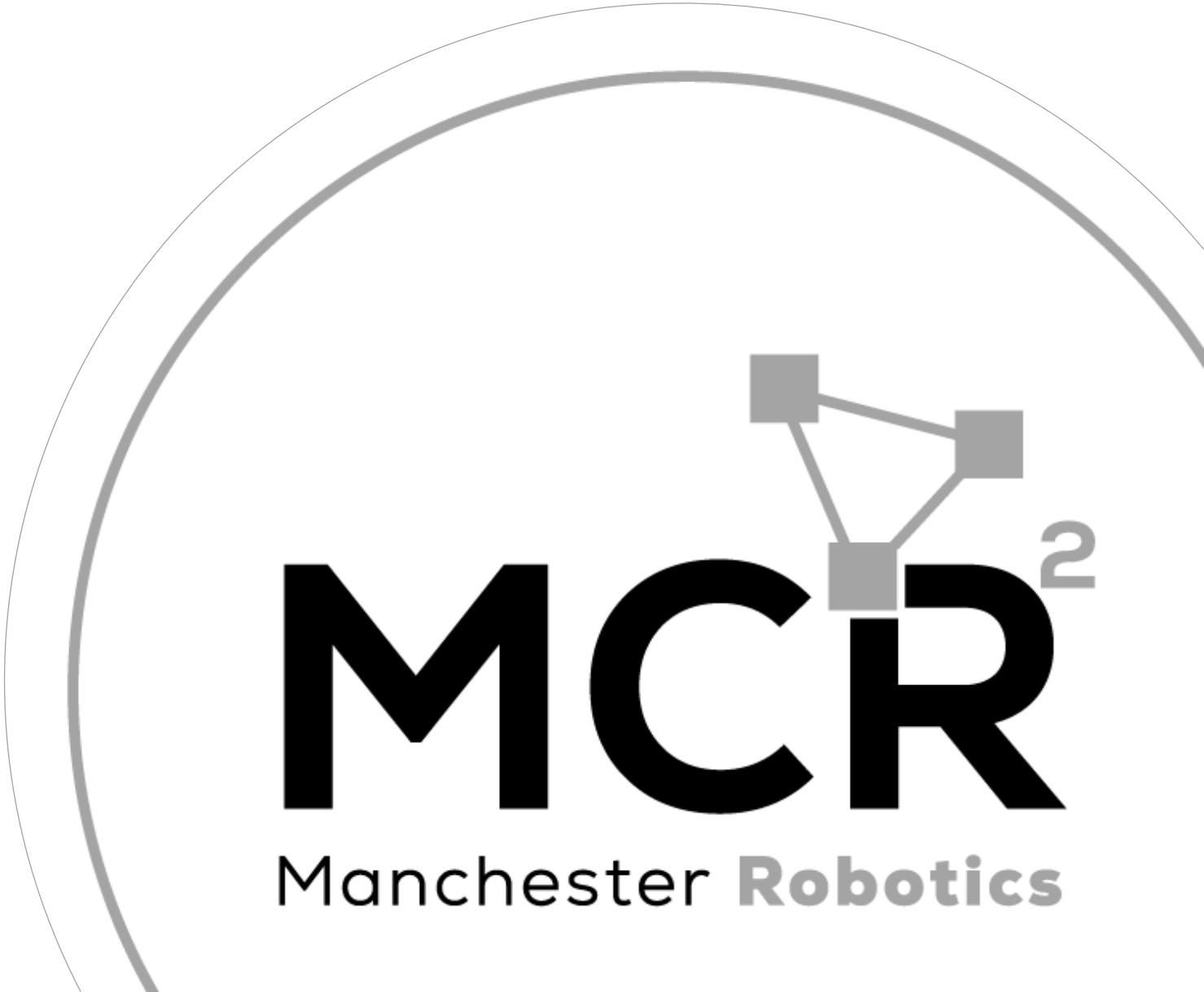
ROS Example


ROS Master
Mobile Robot
External Computer
NVIDIA Jetson
User Node: ML
User Node: Trajectory
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

Robot Operating System - ROS

ROS Organization

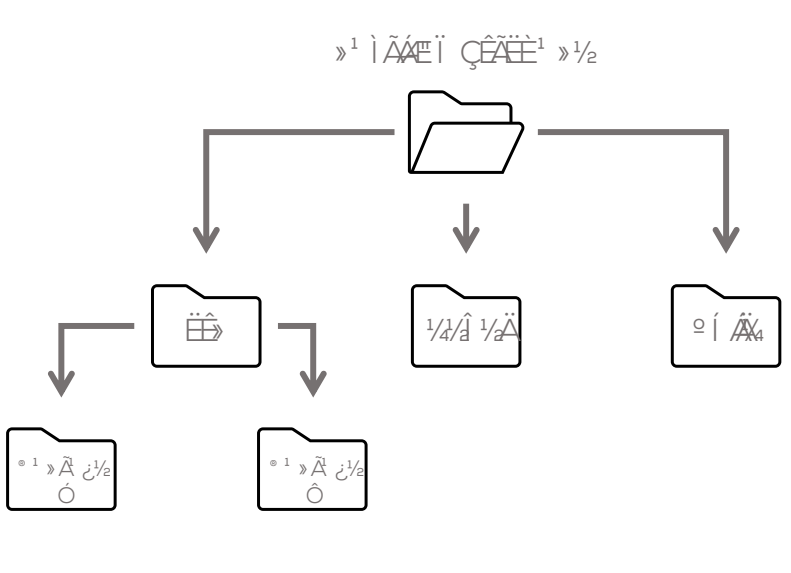
{Learn, Create, Innovate};

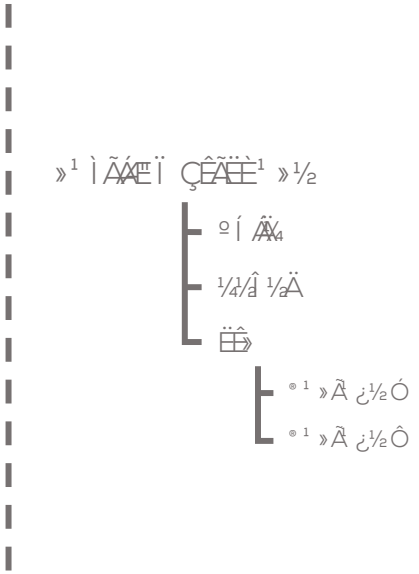


ROS Organization

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).
- The official name for workspaces in ROS is **catkin workspaces**.

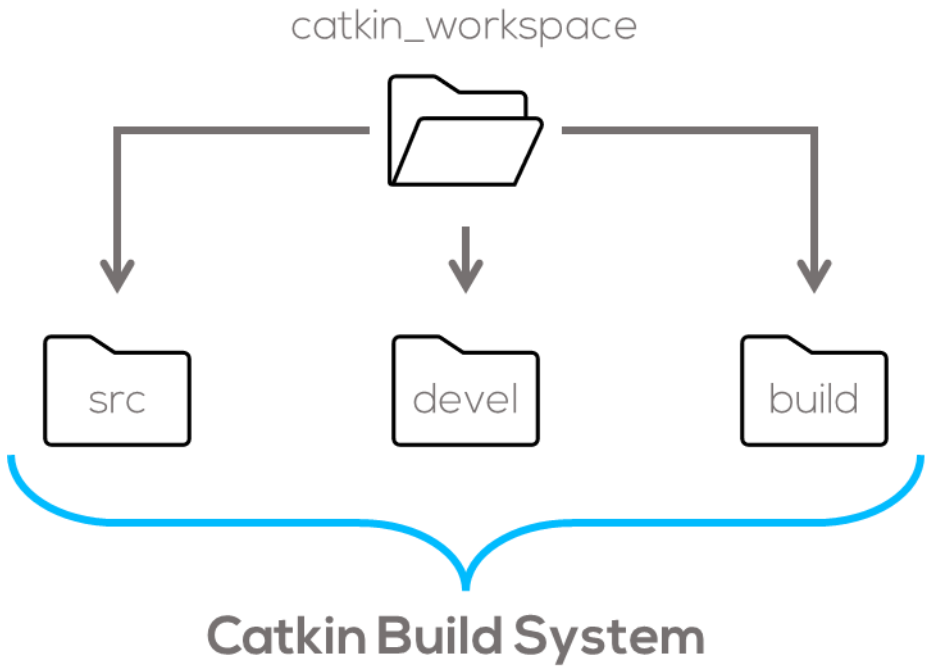




ROS Organization

Catkin Workspace

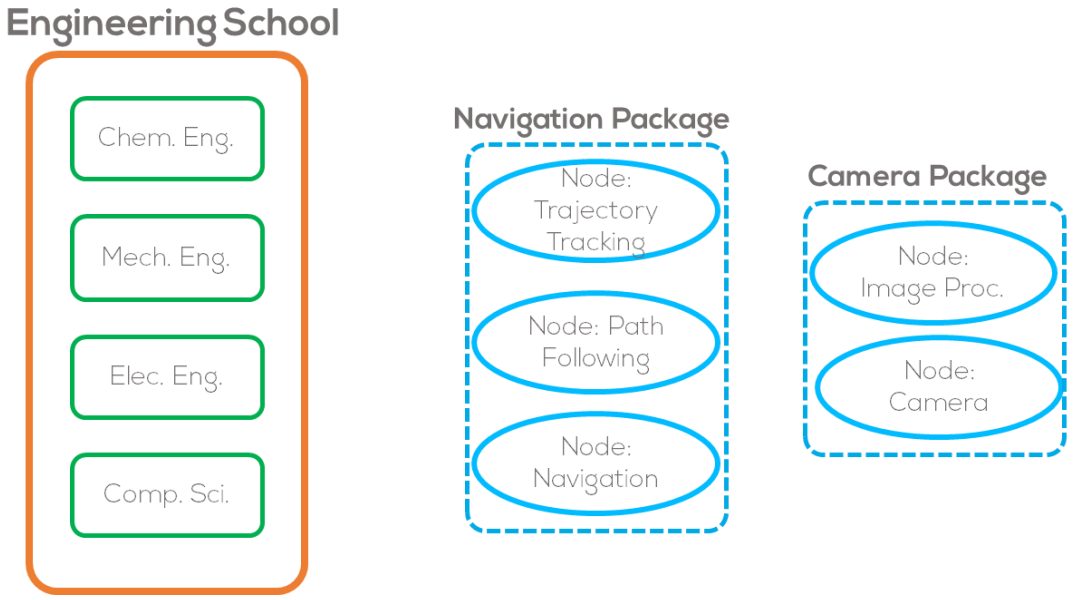
- *src:* Called “Source Space”, contains the source code. This space is where the user creates, copies or edit the source code to build the different packages.
- *build:* This space is reserved by the CMake file to build the packages in the src folder. This file concentrates the cache information and other intermediate files to be used when building the packages. **DO NOT TOUCH!**
- *devel:* Development space, where the built targets and executables are placed, before being installed. **DO NOT TOUCH!**



ROS Organization

ROS Packages

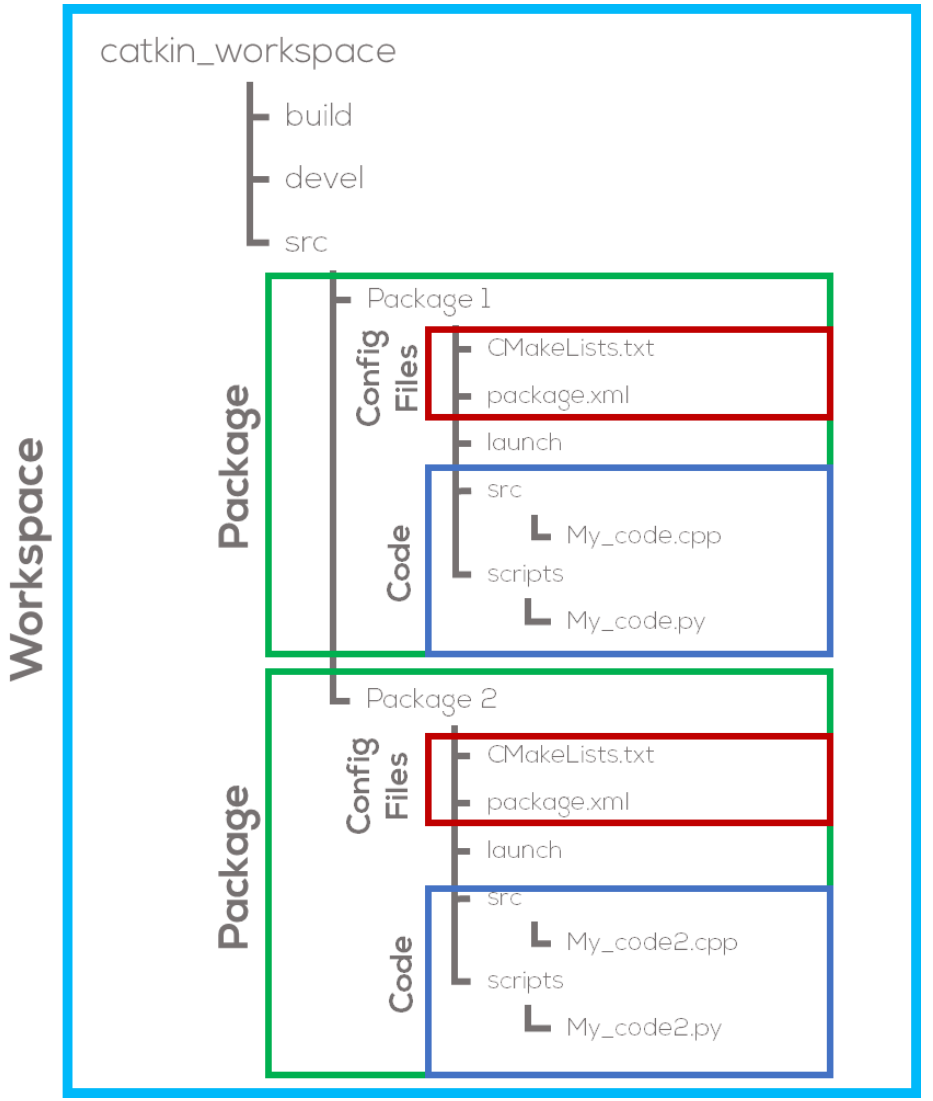
- To keep everything more organized, ROS uses the concept of packages.
- ROS Packages are a way of organizing code related between each other.
- The same way teachers are gathered within the engineering school nodes are gathered in ROS packages.
- Another analogy related with the university, could be 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

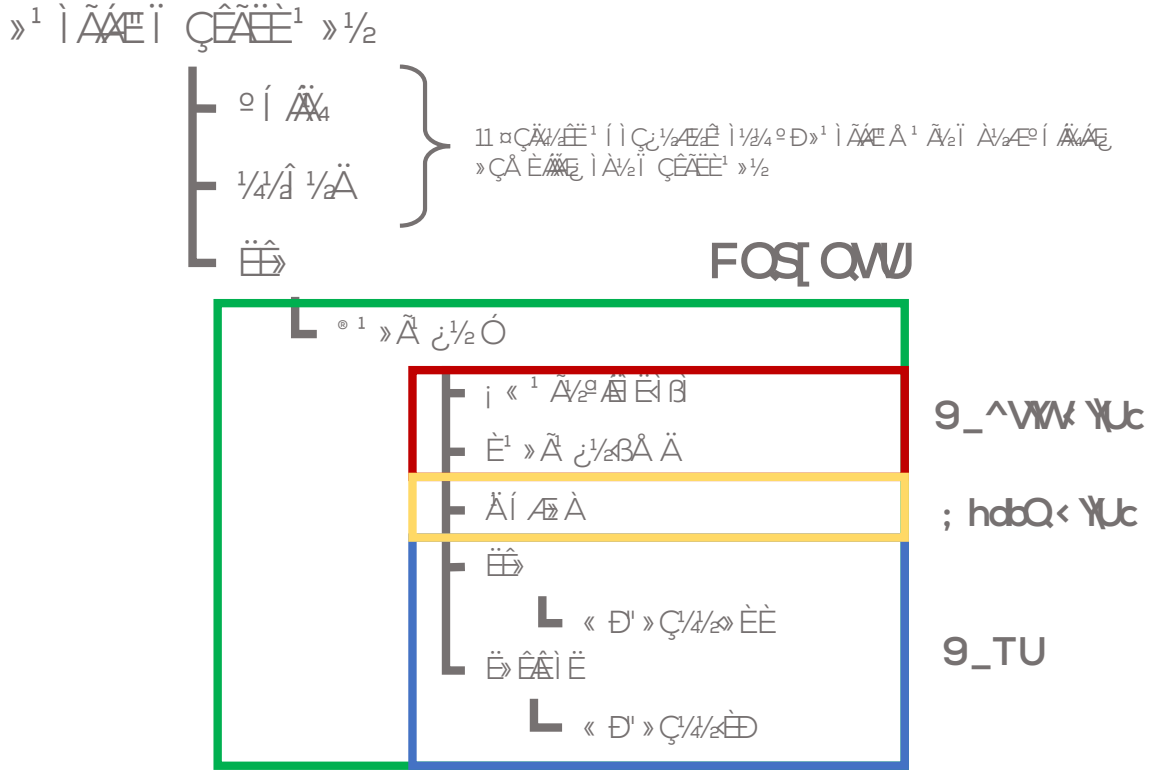
- Each package in ROS requires different attributes to be compiled.
- These attributes are usually dependencies related with other packages, external libraries or custom messages, services and actions.
- The preferred compilation tool is known as **catkin** and uses two separated files, **package.xml** and **CMakeLists.txt**.
- Instructions for the compiler need to be allocated in CMake and Package files.



ROS Organization

A Closer Look Into a ROS Package

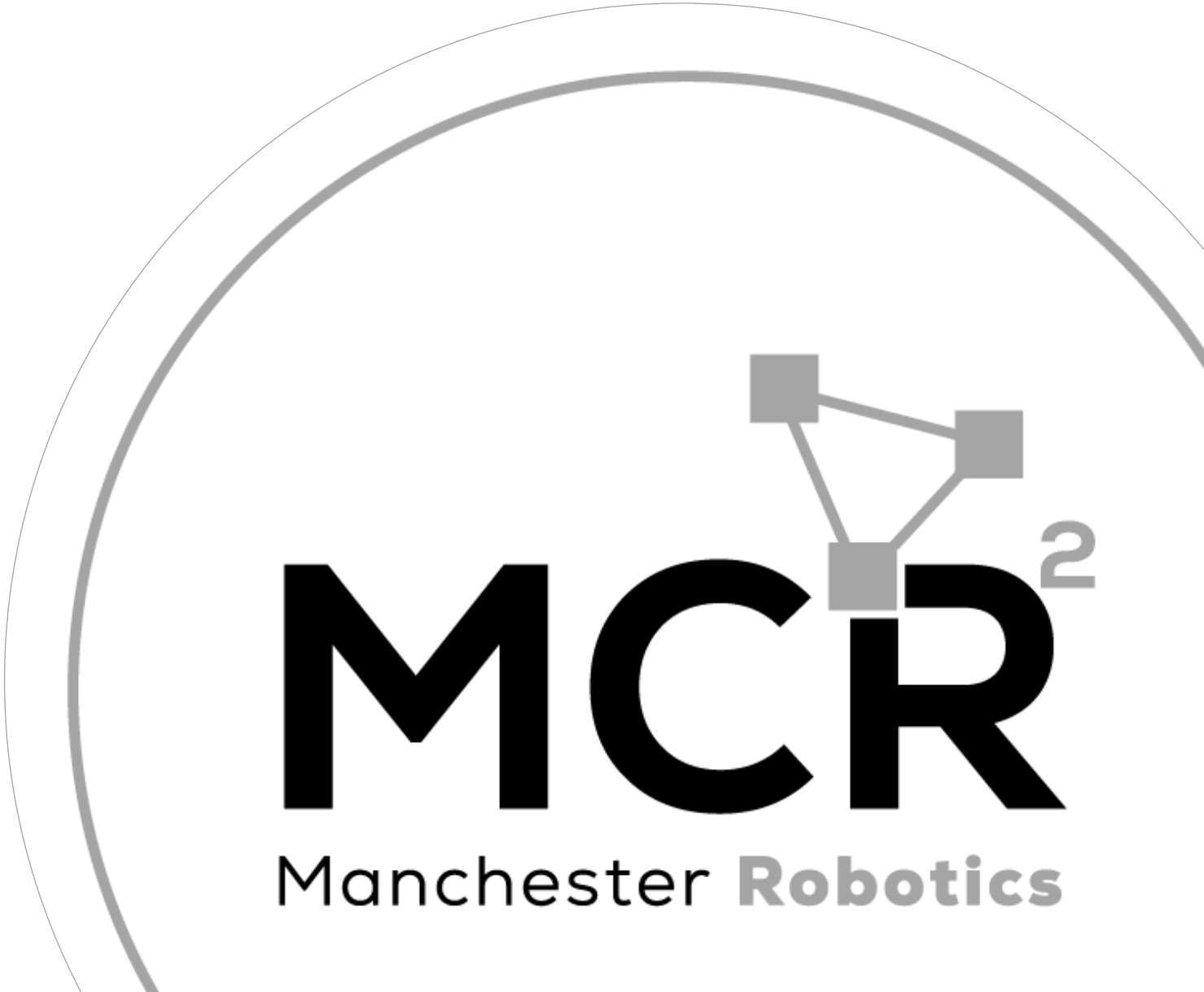
- Packages: Files that can be exported between projects
- Configuration files used to stablish code dependencies.
- Extra files and folders: They are files/folders dedicated for specific tasks. In this case the launch file allow us to execute several nodes at the same time
- Nodes : Code that we will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



ROS Activities

Activity #1: Talker – Listener

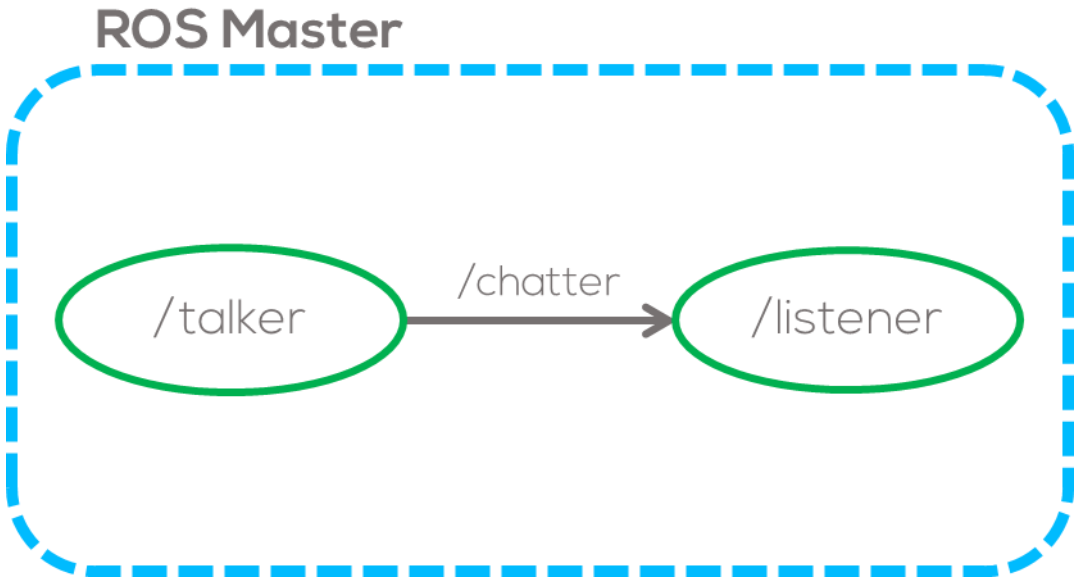
{Learn, Create, Innovate};



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

Creating a package

- The easiest way to create a package is to use the ROS tool catkin_create_pkg as seen here.

```
catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
```

- For this exercise, create a package called basic_comms. The dependencies used are rospy and std_msgs. Open a terminal and type the following

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg basic_comms std_msgs rospy
```

Beware that the command "`catkin_create_package`" must be run inside the "src" folder.

- Once the package is created you will be able to see the package folder in ~/catkin_ws/src.
- Build the package you just created and add it to your environment (more information here)

```
$ cd ~/catkin_ws
$ catkin_make
$ source devel/setup.bash
```

ROS Activity 1

Creating and Configuring the Talker Node

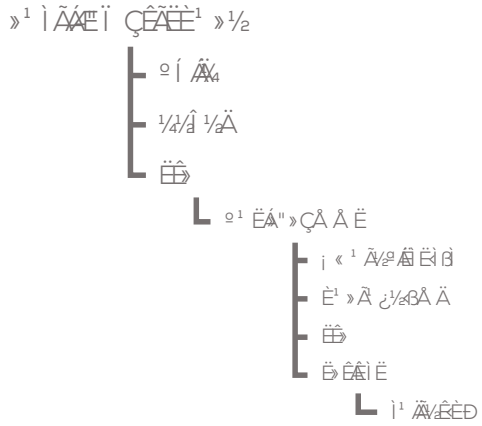
- As a convention python programs (nodes) are storage in a folder called scripts i.e., basic_coms/scripts/talker.py (you can use the basic_comms/src folder as well but is not a standard convention)
- Create a folder for the python scripts that will be used for each node and generate a python script called “talker.py” inside the scripts folder.

```
$ cd ~/catkin_ws/src/basic_comms
$ mkdir scripts
$ cd scripts
$ touch talker.py
```

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

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

- The folder tree should look as follows.



- Open the file CMakeLists.txt in the folder “~/basic_comms/CMakeLists.txt” and find the following line uncomment it (remove the #) and add the highlighted line in yellow. This makes sure the python script gets installed properly and uses the right python interpreter.

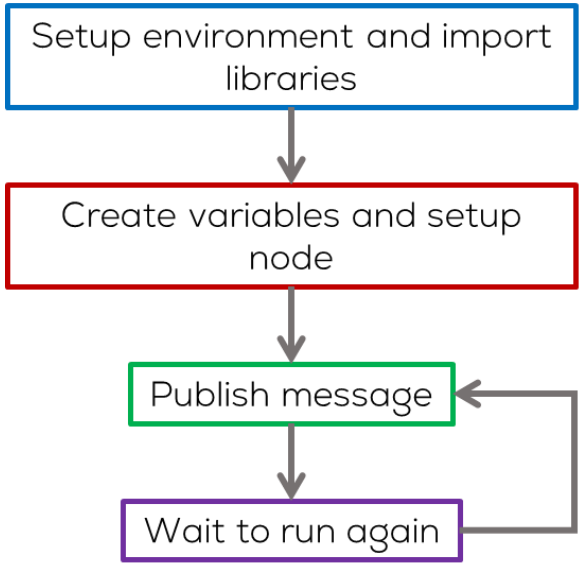
```
catkin_install_python(PROGRAMS scripts/talker.py
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

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".
- Use any word processor or IDE (nano, VS Code, Vim, etc.) to open the file.

ROS Activity 1

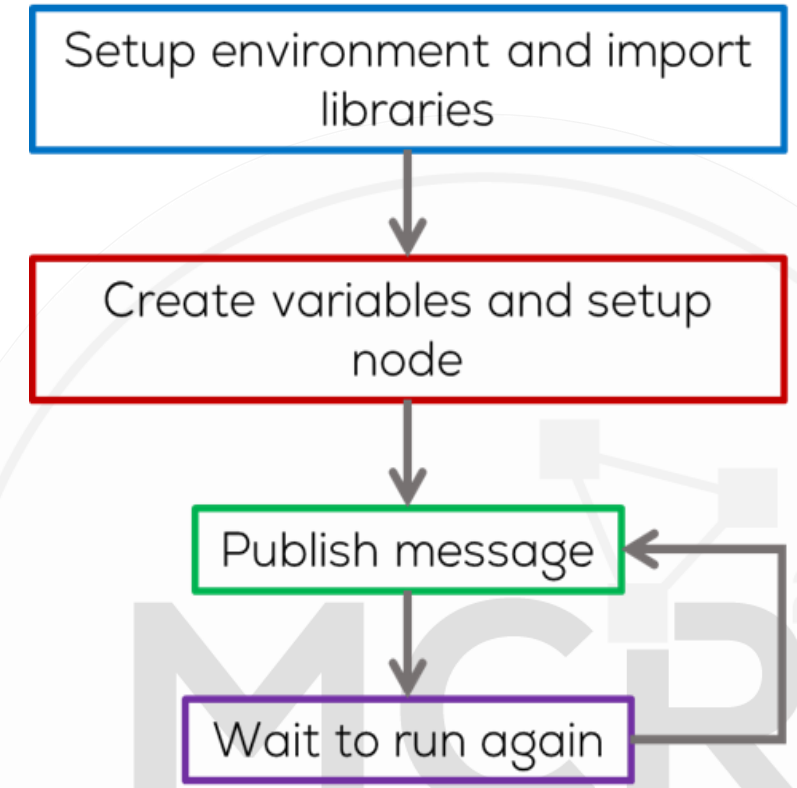
Talker Node

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String
```

```
if __name__ == '__main__':
    pub = rospy.Publisher("chatter", String, queue_size=10)
    rospy.init_node("talker")
    rate = rospy.Rate(10)
```

```
    while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
        pub.publish(hello_str)
```

```
        rate.sleep()
```



ROS Exercise 1

Running the node

- Open a terminal and re-build your package

```
$ cd ~/catkin_ws
$ catkin_make
$ source devel/setup.bash
```

- Run ROS

```
$ roscore
```

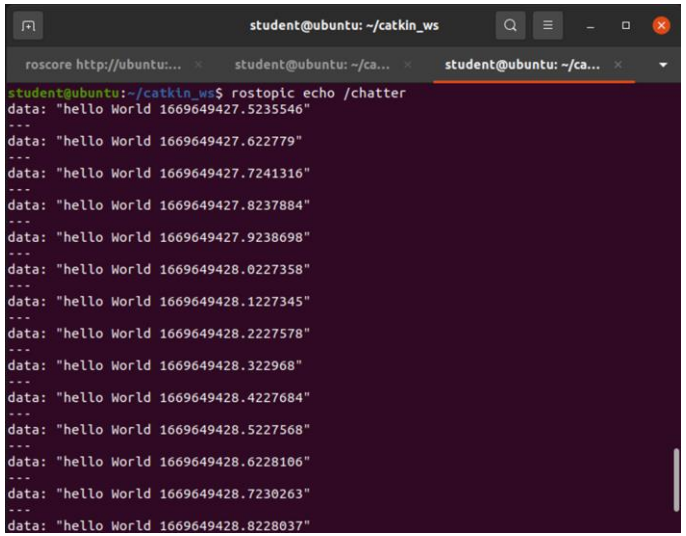
- Open a new terminal and run the node you just made using the following command

```
$ rosrun basic_comms talker.py
```

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

```
$ rostopic echo /chatter
```

Results



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

ROS Tools

- roscore
  - Must be executed before working with ROS.
  - Handles the correct functionality of the network
- rostopic
  - Contains useful tools related with topics
  - `rostopic list` : *Lists the topics available.*
  - `rostopic echo [topic name]` *: Prints the message in a topic.*
  - `rostopic pub [topic name] [topic type] [message]`*: Sends a message into a topic.*
  - `rostopic hz [topic name]` *: Publishing rate of a topic.*
- rosrun
  - `rosrun [package] [node] :` Executes a node.
- rosnode
  - Displays information related with the nodes
  - `rosnode info [node]` : Displays information about a node.
  - `rosnode kill [node]` : Stops a node.
- Other ROS tools can be found here.

ROS Activity 1

Listener Node

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

```
$ cd ~/catkin_ws/src/basic_comms/scripts
$ touch listener.py
```

- Give permission for the node to be executed

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

- Add the node name to the CMakeLists.txt ("~/basic_comms/CMakeLists.txt") in the same section as in the previous node separated by a space

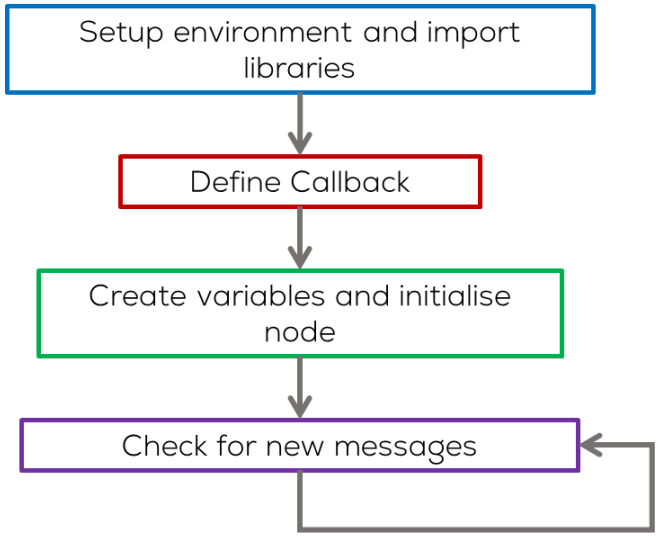
```
catkin_install_python(PROGRAMS scripts/talker.py scripts/listener.py
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

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.

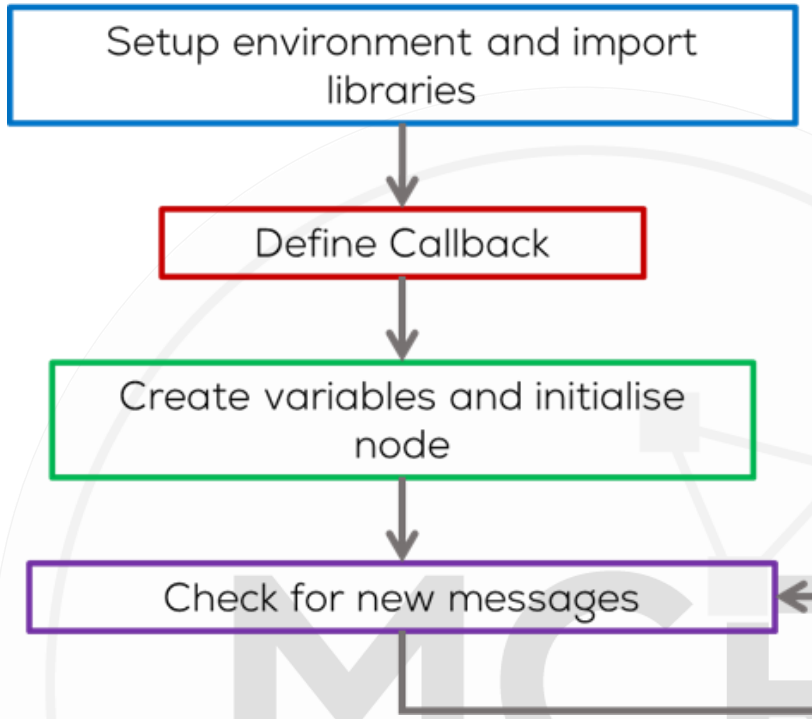


- Your code will be written in the file "listener.py".
- Use any word processor or IDE (nano, VS Code, Vim, etc.) to open the file .

ROS Activity 1

Listener Node

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def callback(msg):
    rospy.loginfo("I heard %s", msg.data)

if __name__ == '__main__':
    rospy.init_node('listener')
    rospy.Subscriber("chatter", String, callback)

    rospy.spin()
```



ROS Activity 1

Running the node

- Open a terminal and re-build your package

```
$ cd ~/catkin_ws
$ catkin_make
$ source devel/setup.bash
```

- Run ROS

```
$ roscore
```

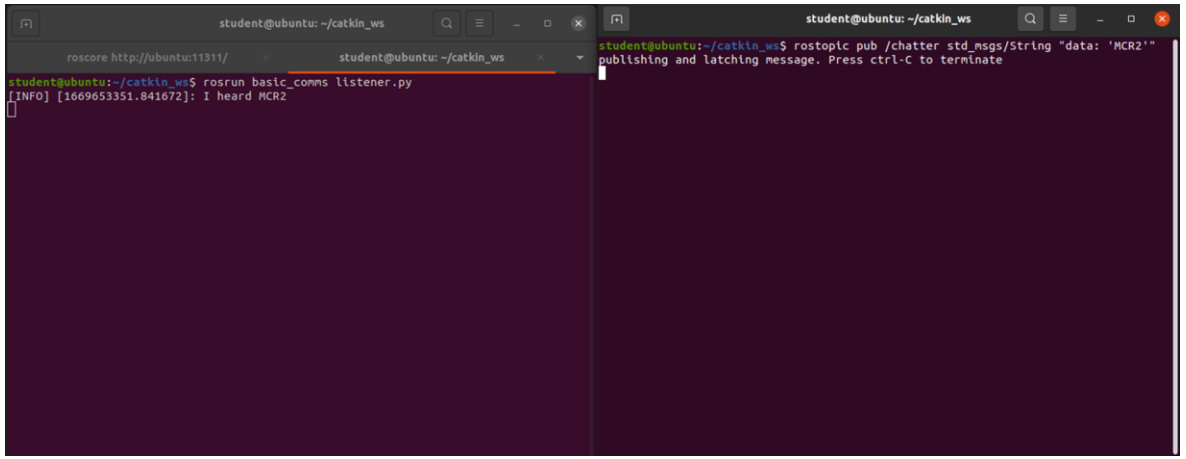
- Open a new terminal and run the node you just made using the following command

```
$ rosrun basic_comms listener.py
```

- To visualize the output of the node, open another terminal and use the command *“pub”* to publish a message manually as follows

```
$ rostopic pub /chatter std_msgs/String “data : ‘MCR2’”
```

Results

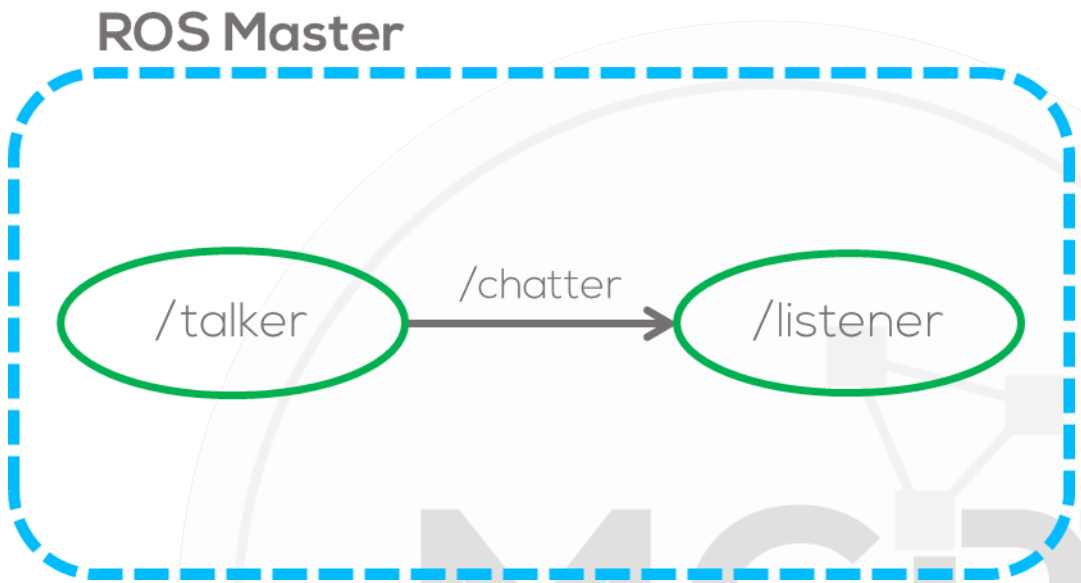


Press “Ctrl+c” at each open terminal to stop the nodes and ROS

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 manually

- Open a terminal and run ROS

```
$ roscore
```

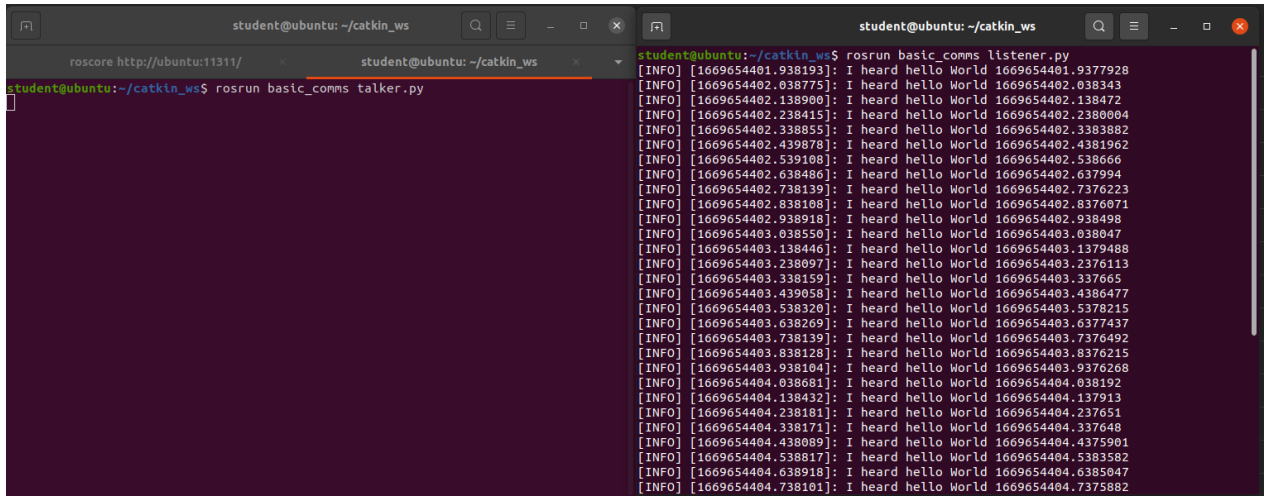
- Open a new terminal and run the talker node you just made using the following command

```
$ rosrun basic_comms talker.py
```

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

```
$ rosrun basic_comms listener.py
```

Results



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

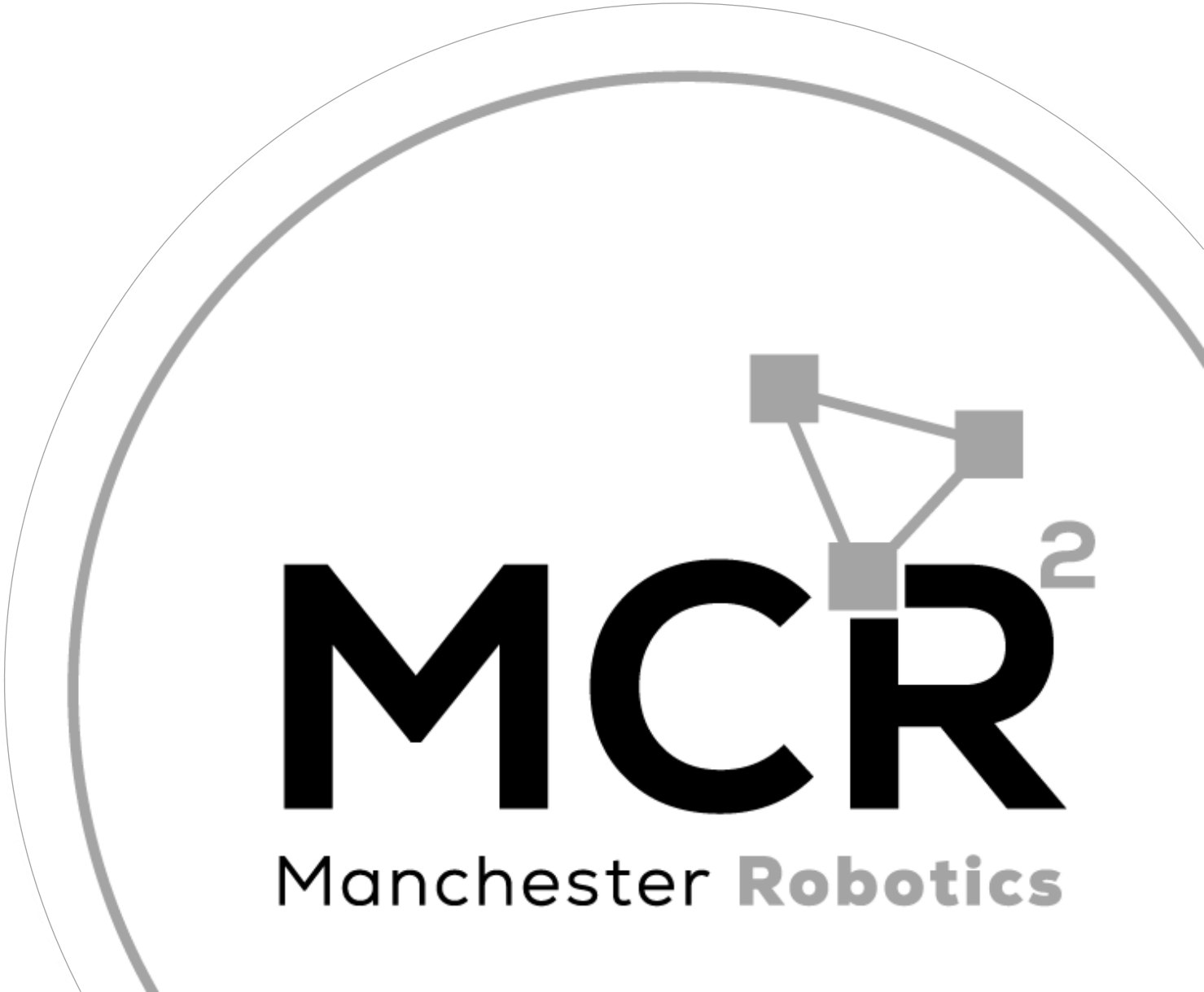
ROS Tools

- roslaunch
  - *roslaunch [package] [launch]* : Allows to launch multiple nodes with a single command by calling .launch files
- rosbag
  - Records the activity of the topics in the network .
    - *rosbag record*
    - *rosbag info [bag file]*
    - *rosbag play [bag file]*
- Config files
  - Set and load parameters into the system that can be accessed by any node.
- rqt
  - Visualization tool that provides graphical information of the nodes and system status.
    - *rosrun rqt_plot*: Displays scalar data published to ROS topics.
    - *rosrun rqt_graph*: Displays a visual graph of the processes running in ROS and their connections.
- Other ROS tools can be found here.

Robot Operating System - ROS

ROS Launch Files

{Learn, Create, Innovate};



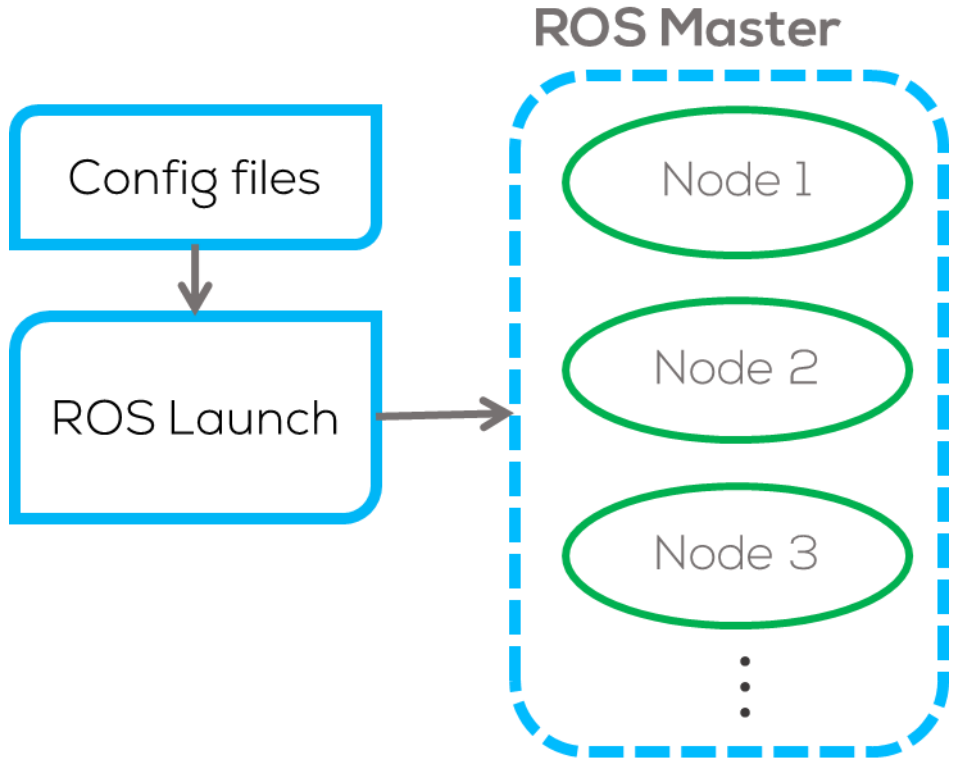
ROS Launch file

ROS Launch files

- Launch files are sets of commands written in xml that allow executing various scripts at the same time.
- The general syntaxis (of this XML file) is the following

```
<?xml version="1.0"?>
<launch>
      [Body of the launch file]
</launch>
```

- This syntaxis allows to run any object used within the ROS 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 you project.
- An extensive documentation can be found here

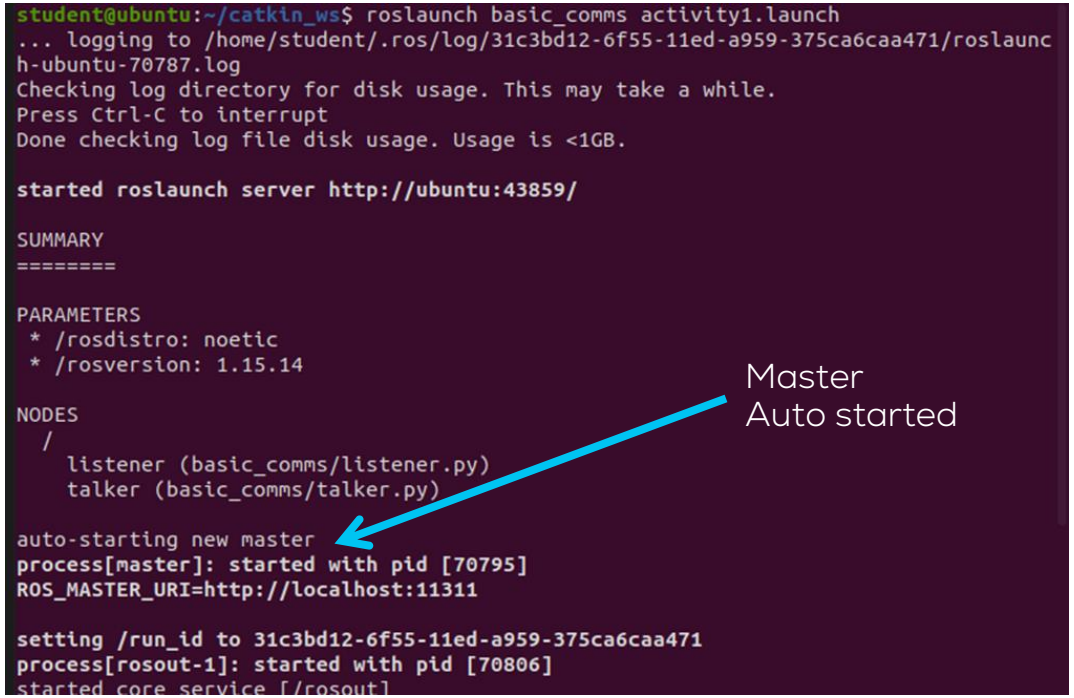


ROS Launch file

- Roslaunch has the capability to initialise a roscore automatically, alongside the nodes specified in it.
- 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.
- The ROS Master will shut down when the roslaunch process is killed (ctrl+c).

Warning: When working with multiple launch files or nested launch files, this can cause problems since killing the launch file from where the ROS Master is being executed, will cause shut down the ROS Master. In consequence, the remaining nodes are left without a master causing it to crash.

It is recommended that when working with multiple launch files the ROS Master is independent from the launch files.



ROS Launch file

ROS Launch files characteristics

- Running a node

```
<node name="listener" pkg="basic_comms" type="listener.py"
output="screen"/>
```

- Running another file or launch file

```
<include file="$(dirname)/other.launch" />
```

- Set parameters

```
<param name="publish_frequency" type="double" value="10.0" />
```

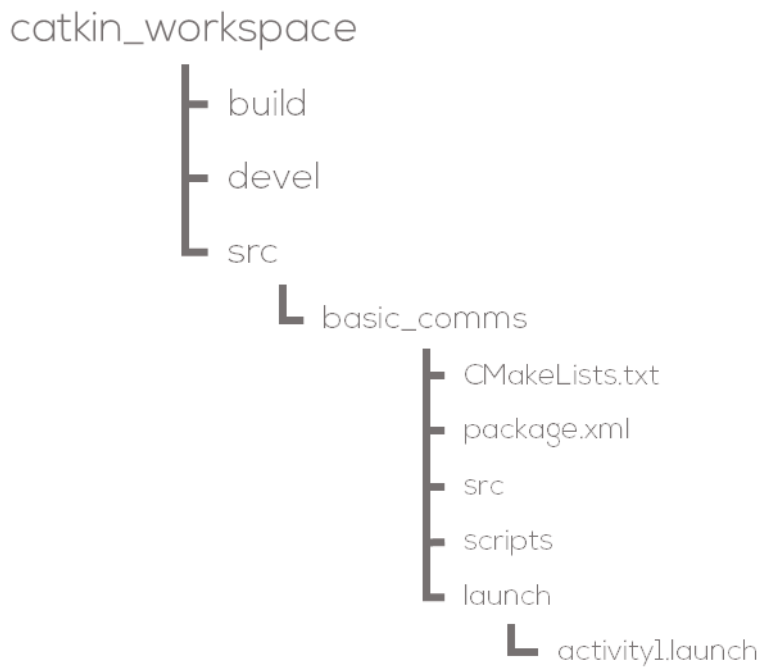
- Pass arguments to the launch file

```
<arg name="camera_id" value="cam_3" />
```

- Load files into the system

```
<rosparam command="load"
file="$(findpackage_name)/config/file_name.yaml" />
```

- ROS Launch files are usually located inside a folder called "launch" in each package.



ROS Activity 1(Launch file)

ROS Launch file for Exercise 1

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

```
$ cd ~/catkin_ws/src/basic_comms
$ mkdir launch
$ cd launch
$ touch activity1.launch
```

- Your code will be written in the file “activity1.launch”.
- Use any word processor or IDE (nano, VS Code, Vim, etc.) to open the file.
- For this exercise, the talker and listener nodes will be launched.

- Launch files do not require to be compiled.
- Roscore will be automatically launched (no need to do roscore).
- The following code will launch the two nodes previously done (talker and listener).

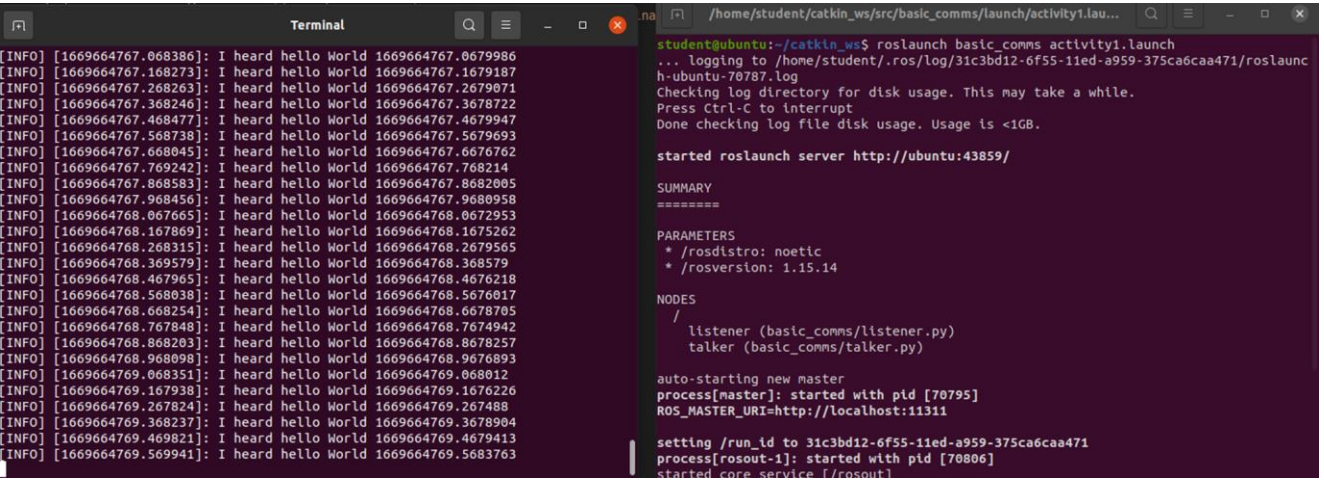
```
<?xml version=“1.0”?>

<launch>

   <node pkg=“basic_comms" type=“talker.py" name=“talker“
   output="screen" />

   <node pkg=“basic_comms" type=“listener.py" name=“listener"
   output="screen" launch-prefix="gnome-terminal --command" />

</launch>
```

ROS Activity 1 (Launch file)

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

```
$ roslaunch basic_comms activity1.launch
```

- The output you should see is the following



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

```
$ rosrun rqt_graph rqt_graph
```

** Be aware that the ROS launch starts a roscore automatically. But when the launch file is killed the roscore will be shut down.

Q&A

Questions?

{Learn, Create, Innovate};

MCR² Manchester Robotics

{Learn, Create, Innovate};

Thank you