{Learn, Create, Innovate};

Robot Operating System – ROS

Introduction

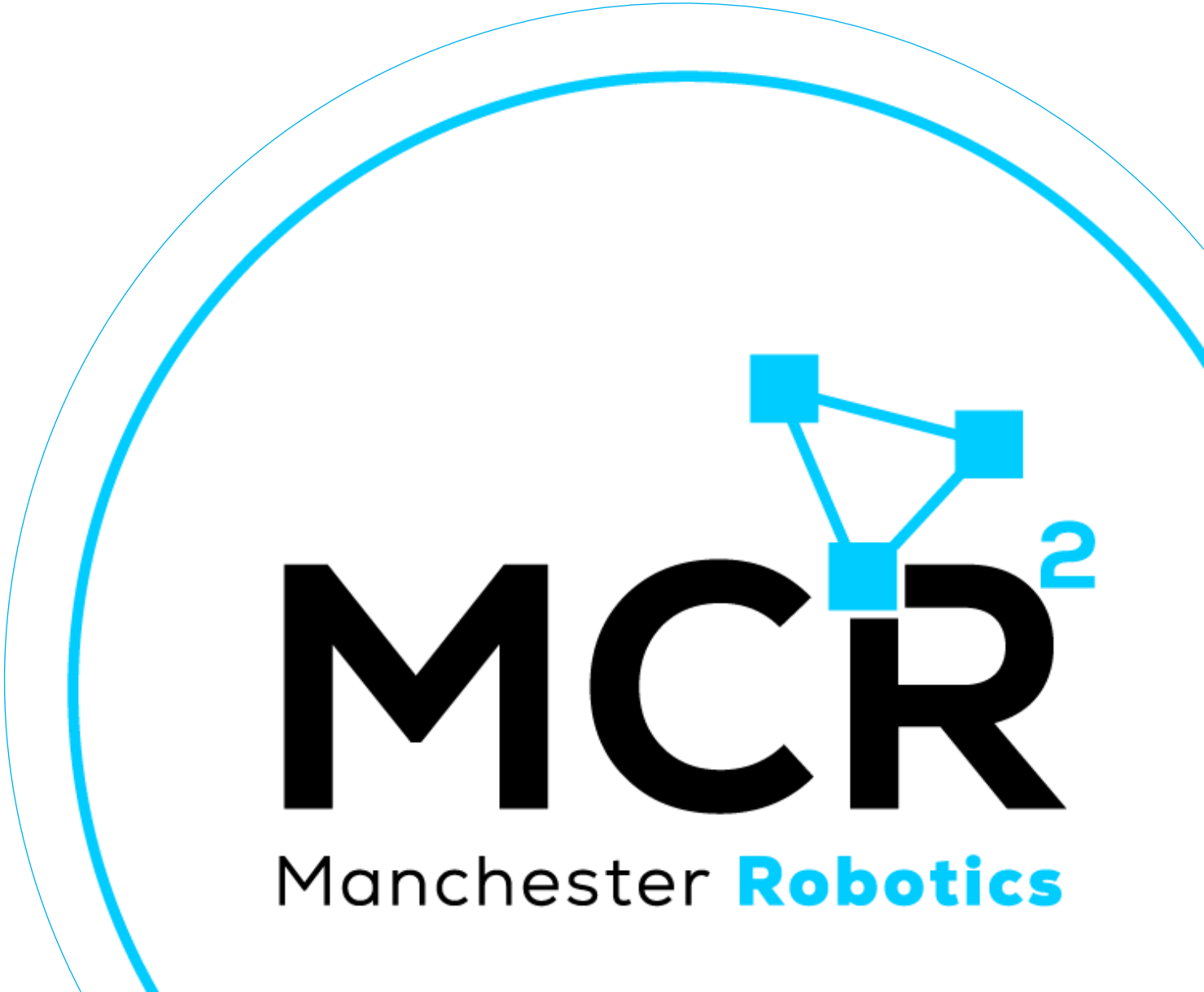
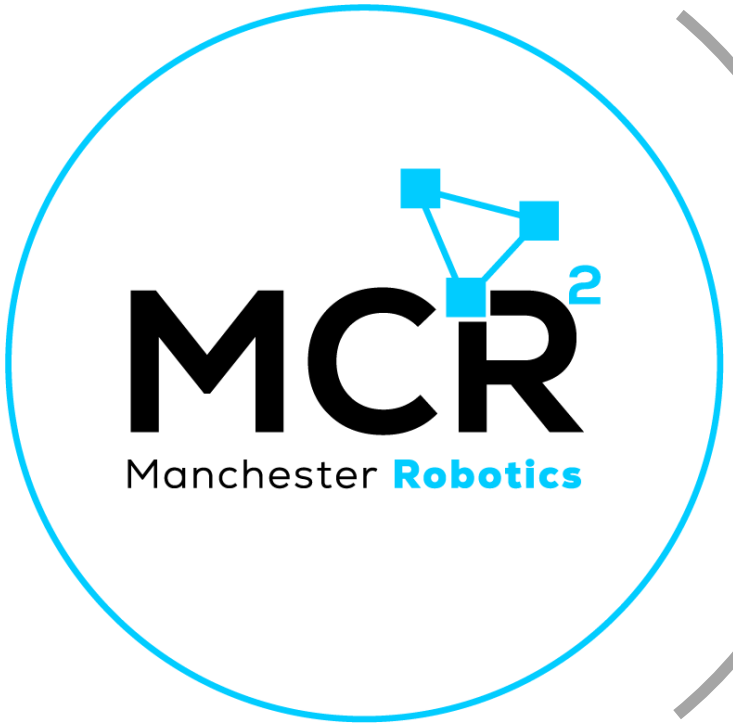


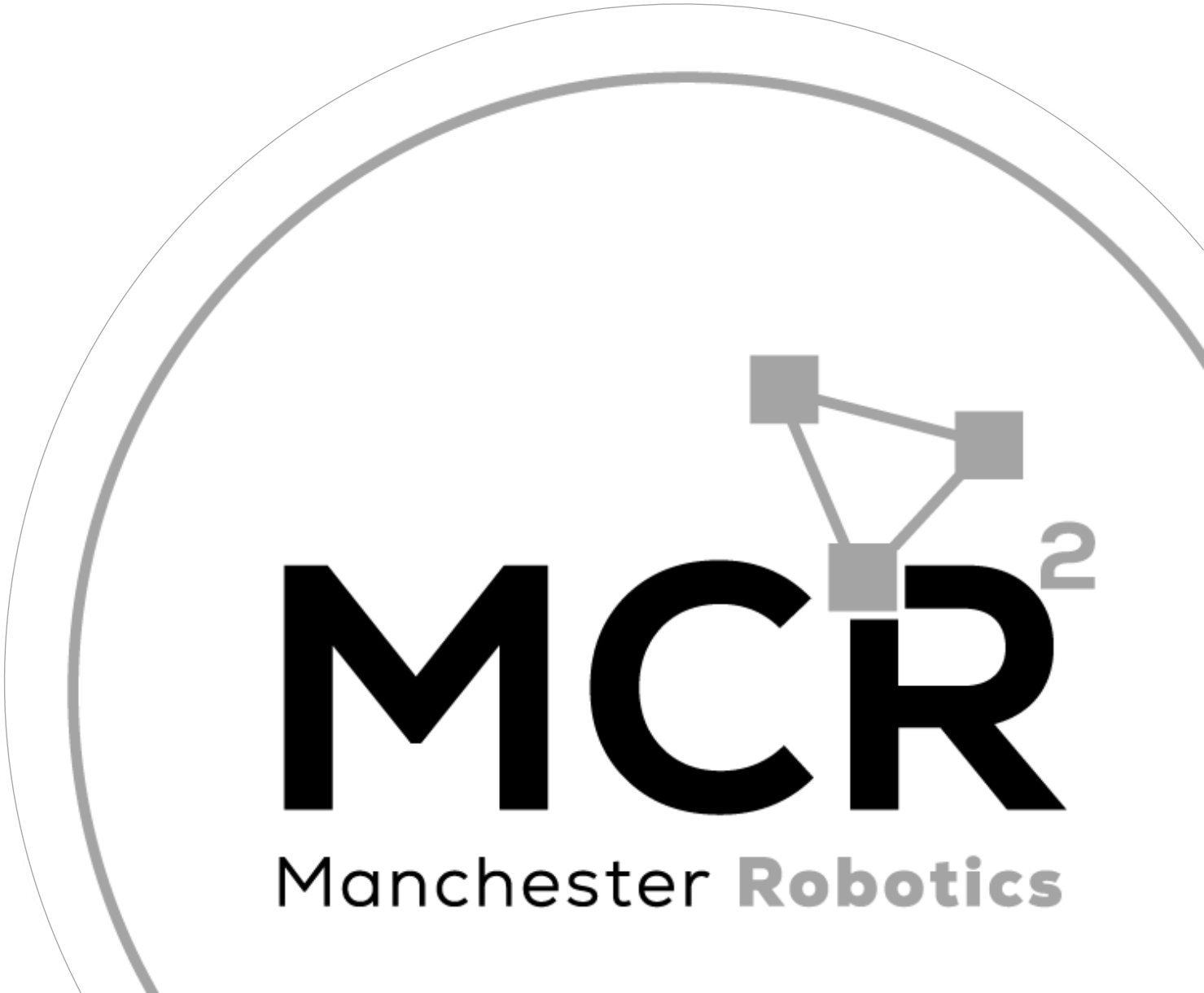
Table of contents



Robot Operating System – ROS

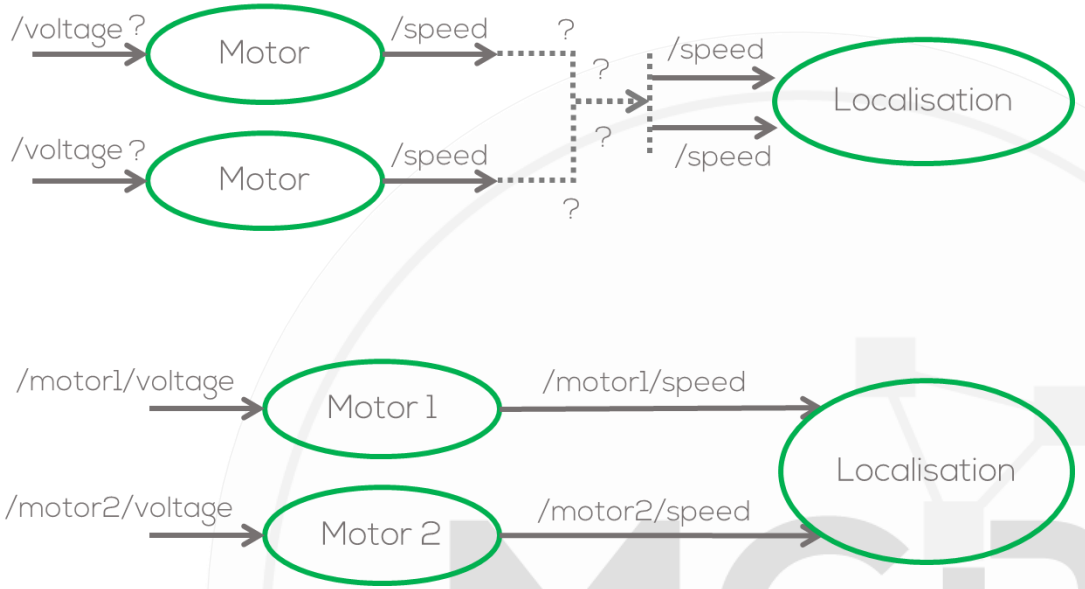
Namespaces

{Learn, Create, Innovate};



ROS Namespaces

- Imagine the following problem: you have a node that simulates a motor, and you require to simulate two (or more) motors using the same code.
- The problem in ROS will be the naming convention for the nodes and the topics to which the motor node subscribes, and where it publishes; since they will both be the same.
    - One simple solution will be to change the name of the nodes and topics manually by generating multiple .py files. For complex system this is not a good option. (What would happen if I require 10 motors?)
- Namespaces then become the best option to deal with name collisions, when systems become more complex.



ROS Namespaces

- A namespace in ROS can be viewed as a directory that contains items with different names.
- The items can be nodes, topics or other namespaces (hierarchy)
- There are several way to define the namespaces. The easiest way is via command line, which is very easy but for larger projects is not recommended.
- In this presentation, the file roslaunch will be used to define the namespaces.

Namespaces in ROS Example

For this example two talker nodes and two listener nodes will be generated using namespaces.

1. Open the launch file of the previous Talker and Listener Example (activity1.launch).
2. Modify it as follows

```
<?xml version="1.0" ?>
<launch>
   <group ns = "Group1">
      <node name="talker" pkg="basic_comms" type="talker.py" output="screen"    launch-
      prefix="gnome-terminal --command" />

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

   <group ns = "Group2">
      node name="talker" pkg="basic_comms" type="talker.py" output="screen" launch-
      prefix="gnome-terminal --command" />

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

</launch>
```

ROS Namespaces

ROS Launch

```
<?xml version="1.0" ?>
<launch>
  <group ns = "Group1">
    <node name="talker" pkg="basic_comms" type="talker.py"
    output="screen"    launch-prefix="gnome-terminal --command" />

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

  <group ns = "Group2">
    node name="talker" pkg="basic_comms" type="talker.py"
    output="screen" launch-prefix="gnome-terminal --command" />

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

</launch>
```

Group Namespace

Group Namespace

1. Execute the Launch file

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

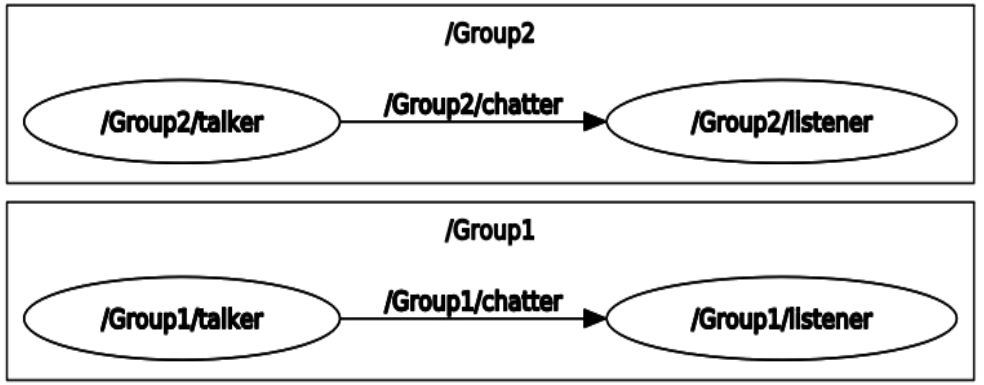
2. Execute the following command in a new terminal

```
$ rostopic list
```

3. In a new terminal, execute the rqt_graph to visualise the nodes

```
$ rosrun rqt_graph rqt_graph
```

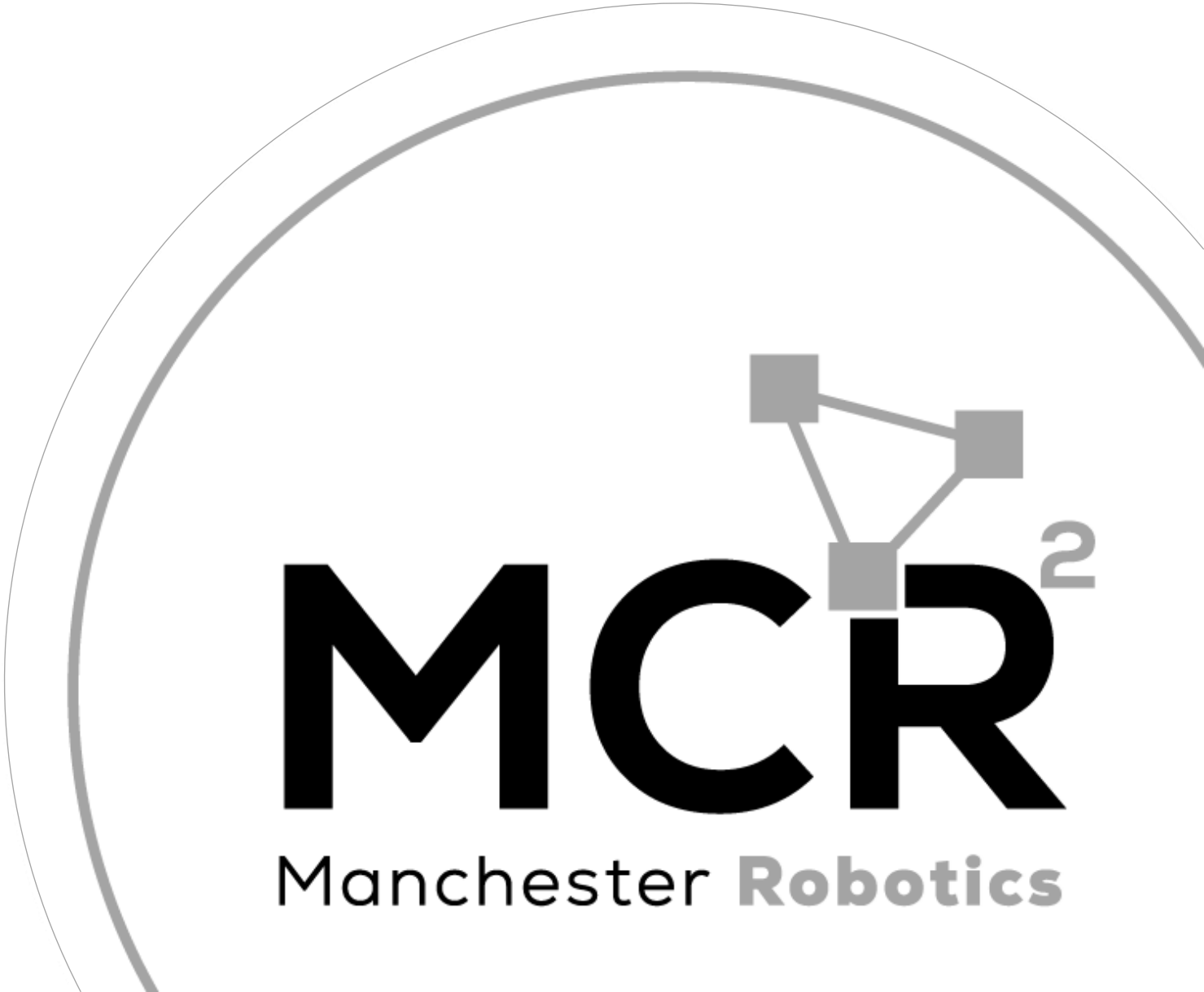
```
student@ubuntu:~$ rostopic list
/Group1/chatter
/Group2/chatter
/rosout
/rosout_agg
student@ubuntu:~$
```



Robot Operating System – ROS

Parameter Files

{Learn, Create, Innovate};



ROS Parameters

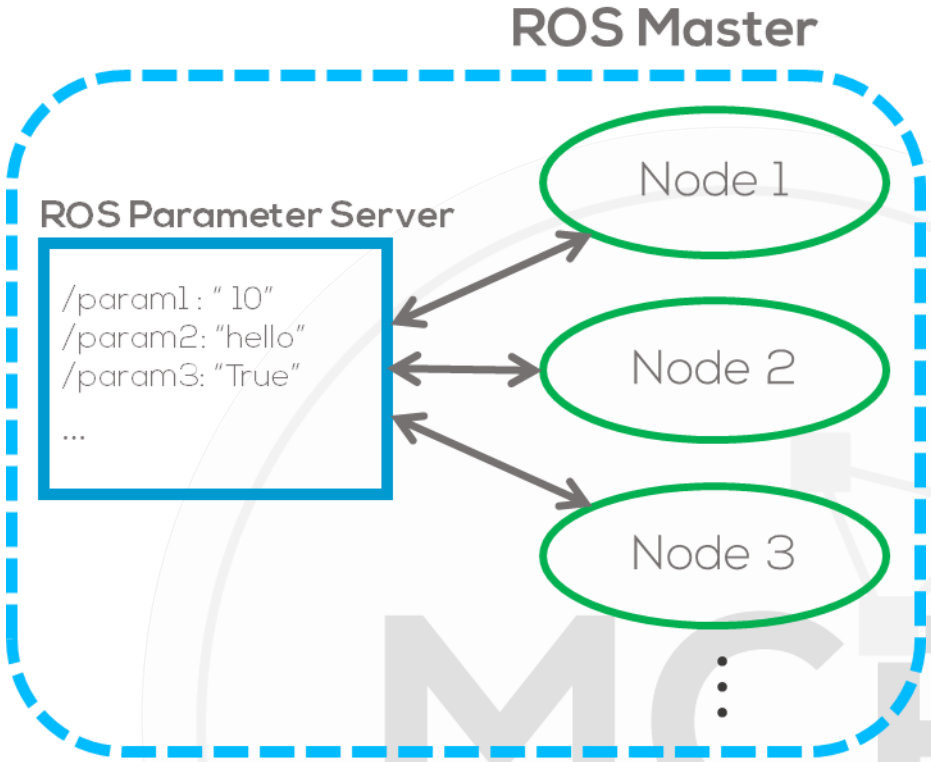
- Any software application, specially in robotics requires parameters.
- Parameters are variables with some predefined values that are stored in a separate file or hardcoded in a program such that the user has easy access to change their value.
- At the same time parameters can be shared amongst different programs to avoid rewriting them or recompiling the nodes (C++)
- In robotics, parameters are used to store values requiring tunning, robot names, sampling times or flags.
- ROS encourage the usage of parameters to avoid making dependencies or rewriting nodes.
- To this end, ROS uses a “dictionary” to store and share the parameters to be used by its nodes. This dictionary is called Parameter Server.

ROS Parameters

ROS Parameter Server

- As stated before, ROS allows to load variables into a server, run by the master which is accessible by all the nodes of the system.
- Nodes use this server to store and retrieve parameters at runtime.
- These parameters, are used to load settings, robot constants, or other data that may be used in different scenarios where the same code is applied.
- This is a globally viewed static library.
- Parameters are composed of a name and a datatype. ROS can use the following types of parameters. More information here.

- 32-bit integers
- Booleans
- iso8601 dates
- strings
- doubles
- lists
- base64-encoded binary data



ROS Parameters

ROS Parameter Server

- In ROS parameters can be set in different ways.
- By the user in the launch files to be used by the nodes.
- They can also be taken from the command line and passed directly into the nodes.
- Set by nodes themselves.
- Parameters can be global (global namespace), within a local namespace (to a group of nodes only) or private i.e., specific to a node.
- It useful to point out that the parameter server depends on the execution of the ROS master not to the node execution time. Therefore, if you kill a node but not the master, the parameter will keep the value if it was modified.

ROS Parameter command line tools

- rosparam
  - *`rosparam set [param]:`* set a parameter
  - *`rosparam get [param]:`* get a parameter
  - *`rosparam delete [param]:`* delete parameter
  - *`rosparam list:`* list parameter names

** This is only a list with the most basic and used command line parameter tools. Other parameter command line tools can be seen here.

Parameters example

ROS Parameter Example (Setting the parameters)

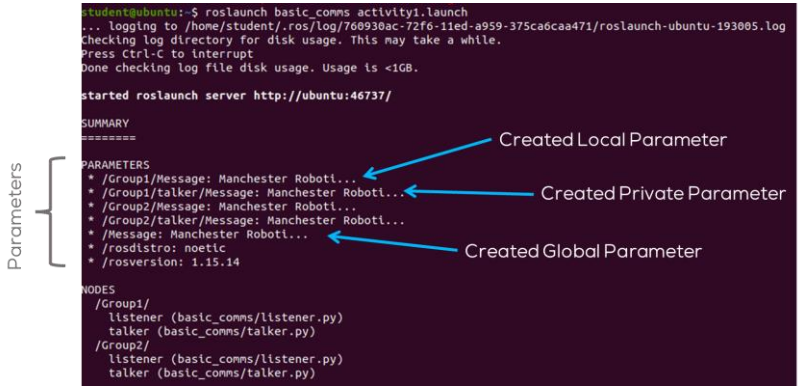
For this example, a global, a local, and a private parameters will be declared in the launch file.

1. Open the previous example Launch file “activity1.launch”, and overwrite it as follows

```
<?xml version="1.0" ?>
<launch>

<param name = "Message" value = "Manchester Robotics Global!" />

   <group ns = "Group1">
            <param name = "Message" value = "Manchester Robotics Local!" />

            <node name="talker" pkg="basic_comms" type="talker.py" output="screen" launch-prefix="gnome-terminal --command" >
                  <param name = "Message" value = "Manchester Robotics Private!" />
            </node>
            <node name="listener" pkg="basic_comms" type="listener.py" output="screen" launch-prefix="gnome-terminal --command" />
   </group>

   <group ns = "Group2">
            <param name = "Message" value = "Manchester Robotics Local!" />
            <node name="talker" pkg="basic_comms" type="talker.py" output="screen" launch-prefix="gnome-terminal --command" >
                        <param name = "Message" value = "Manchester Robotics Private!" />
            </node>
            <node name="listener" pkg="basic_comms" type="listener.py" output="screen" launch-prefix="gnome-terminal --command" />
   </group>

</launch>
```

Parameters example

- Save the launch file and execute it as defined previously.
- You will see the parameters being displayed when ROS initializes



- Open a terminal and type the following command to view the parameters in ROS

```
$ rosparam list
```

```
student@ubuntu:~$ rosparam list
/Group1/Message
/Group1/talker/Message
/Group2/Message
/Group2/talker/Message
/Message
/rosdistro
/roslaunch/uris/host_ubuntu__46737
/rosversion
/run_id
```

Launch file parameters

- The parameters declared outside the group's namespace are classified as global parameters. They are in the "root" of the directory therefore they only start with the forward slash "/". For this case, the parameter is depicted as "/Message"
- If the parameter is in the namespace of the group, the address decreases one level in this case "/Group1/Message".
- Finally, if the parameter is located inside the node definition it goes down another level "/Group1/Talker/Message"

** This is just part of the launch file

Parameters example

ROS Parameter Example (Getting the parameters)

This example will show hot to use the parameters inside the nodes.

The command t retrieve parameters in python is the following

```
rospy.get_param(<parameter_name>, <default_value>)
```

- Open the file “talker.py”
- Modify the line

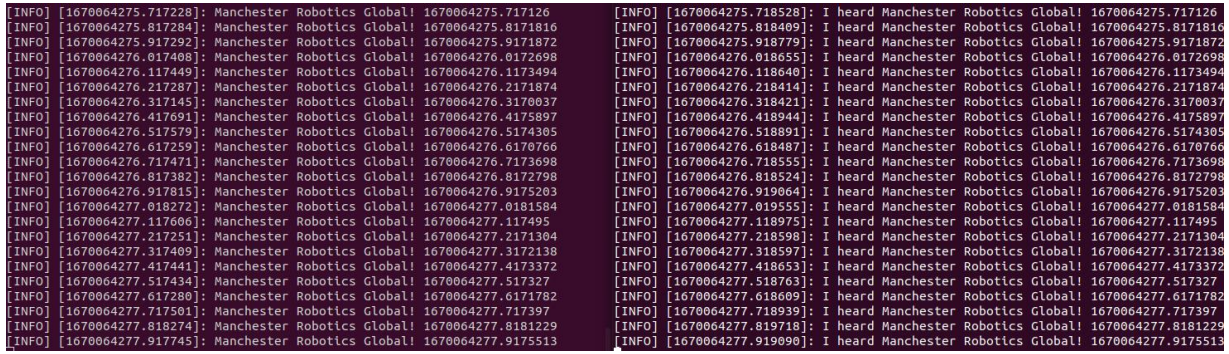
```
hello_str = “hello world %s” % rospy.get_time()
```

- For the following

```
hello_str = rospy.get_param("/Message", "No Parameter Found")  + " "
   + str(rospy.get_time())
```

- Save the file and execute it using the roslaunch file.

- The following results must be shown.



- For accessing the local namespace parameters and the private parameters modify the previous code accordingly. Run the code as before, and you should see the other parameters being printed. More information here.

Local Parameter

```
hello_str = rospy.get_param("Message", "No Parameter Found")  + " " +
   str(rospy.get_time())
```

Private Parameter

```
hello_str = rospy.get_param(“~Message", "No Parameter Found")  + " "
   + str(rospy.get_time())
```

**As in ubuntu use the tilde ~ for private addresses.

Parameter files

Parameter Files

- The previous way of defining parameters is very useful, but for the case when having to define many different parameters this can become very inefficient.
- ROS offers the capability to define parameters using a parameter file.
- These types of files are configuration files, written in YAML. These files are commonly used in other languages to set up parameters or variables.
- The following example will be used to show how to create config files and interact with them.
- The parameters set up in the config files (YAML Files) can be declared as before global, local to a parent namespace or private.
- The way to define the hierarchy of a parameter, like in python depends on spacing.

```
Message: "Parameter YAML File Global“

Group1:
    Message: "Parameter YAML File local G1"
    talker:
              Message: "Parameter YAML File private G1“
```

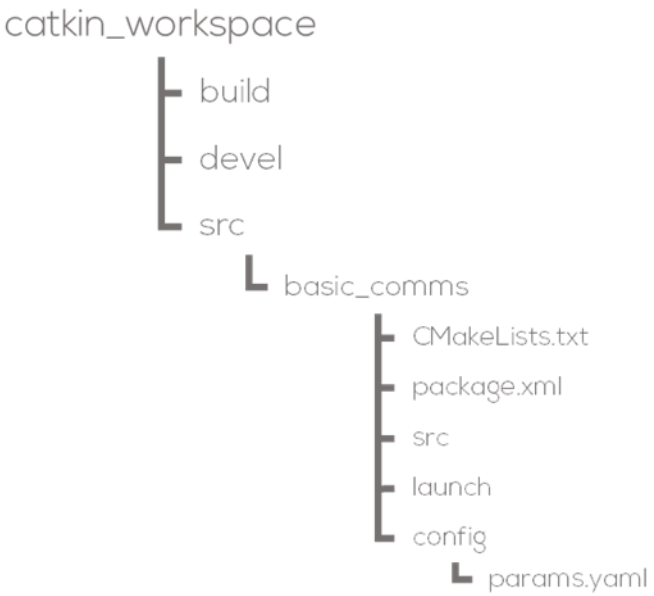
- For this case, “Message” is a global parameter which will be named “/Message” since its located in the root directory.
- The same applies for the following “Message” Parameters, since they will be located a namespace level and a private level for the node “talker”.

Parameter files

Parameter files example

For this example, we will use the previous example.

1. Inside the “basic_comms” package, create a folder named “config”.
2. Inside the config folder create a file named “params.yaml” (the file can have any name just make sure follow a convention properly).



Parameter files

3. Open the newly created parameter file (params.yaml) and edit it as follows

```
Message: "Parameter YAML File Global“

Group1:
    Message: "Parameter YAML File local G1"
    talker:
             Message: "Parameter YAML File private G1“

Group2:
    Message: "Parameter YAML File G2"
    talker:
             Message: "Parameter YAML File private G2"
```

4. Save it and close the file.
5. Open the roslaunch file (activity1.launch)
6. Modify the launch file as follows, save it and close it.

```
<?xml version="1.0" ?>
<launch>

    <rosparam file = "$(find basic_comms)/config/params.yaml" command = "load" />

    <group ns = "Group1">
        <node name="talkerG1" pkg="basic_comms" type="talker.py" output="screen"
    launch-prefix="gnome-terminal --command" />
        <node name="listenerG1" pkg="basic_comms" type="listener.py" output="screen"
    launch-prefix="gnome-terminal --command" />
    </group>

    <group ns = "Group2">
        <node name="talkerG2" pkg="basic_comms" type="talker.py" output="screen"
    launch-prefix="gnome-terminal --command" />
        <node name="listenerG2" pkg="basic_comms" type="listener.py" output="screen"
    launch-prefix="gnome-terminal --command" />
    </group>

</launch>
```

- The first line is where the address of the params.yaml file is located. The label “$(find basic_comms)” allows to the launch file to determine that the path to the config file is inside the package “basic_coms”.

Parameter files

6. Run the modified launch file.
7. Open a new terminal and type **`rosparam list`**

```
student@ubuntu:~/catkin_ws$ rosparam list
/Group1/Message
/Group1/talker/Message
/Group2/Message
/Group2/talker/Message
/Message
/rosdistro
/roslaunch/uris/host_ubuntu__34625
/rosversion
/run_id
```

8. To access each parameter as before, the **`get_parameter`** function must be modified as follows.

Global Parameter

```
hello_str = rospy.get_param("/Message", "No Parameter Found")  + " "
   + str(rospy.get_time())
```

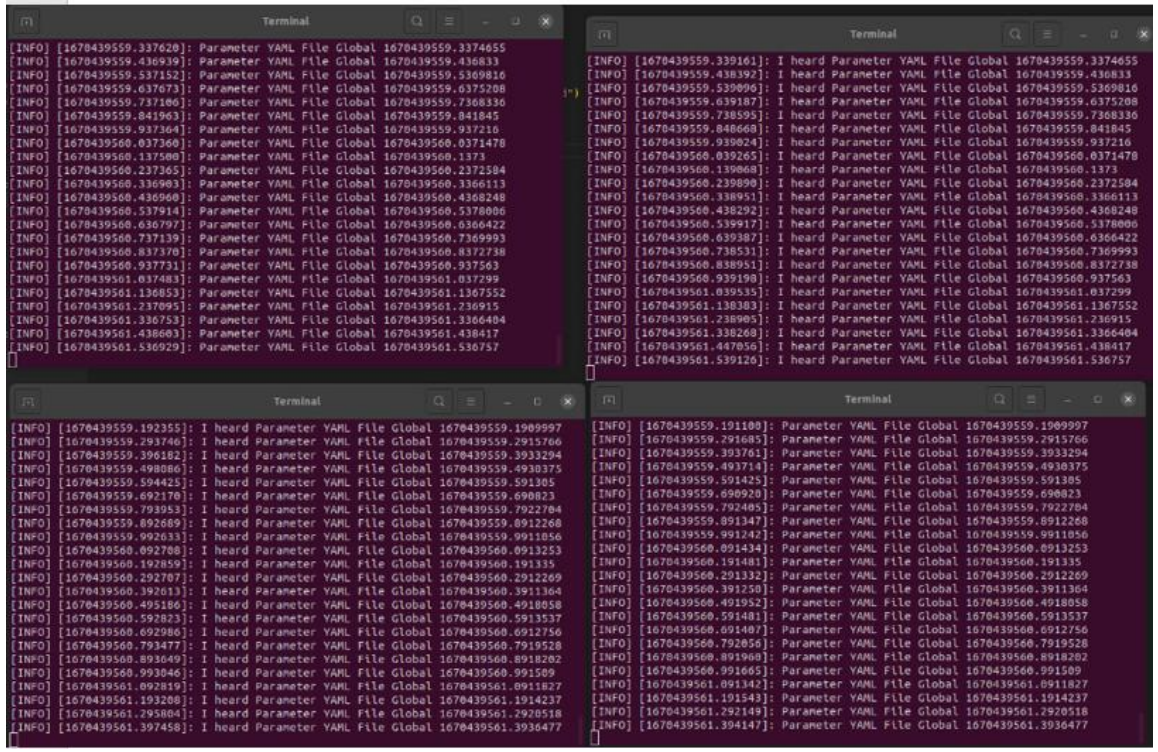
Local Namespace Parameter

```
hello_str = rospy.get_param("Message", "No Parameter Found")  + " " +
   str(rospy.get_time())
```

Private Parameter

```
hello_str = rospy.get_param("~Message", "No Parameter Found")  + " "
   + str(rospy.get_time())
```

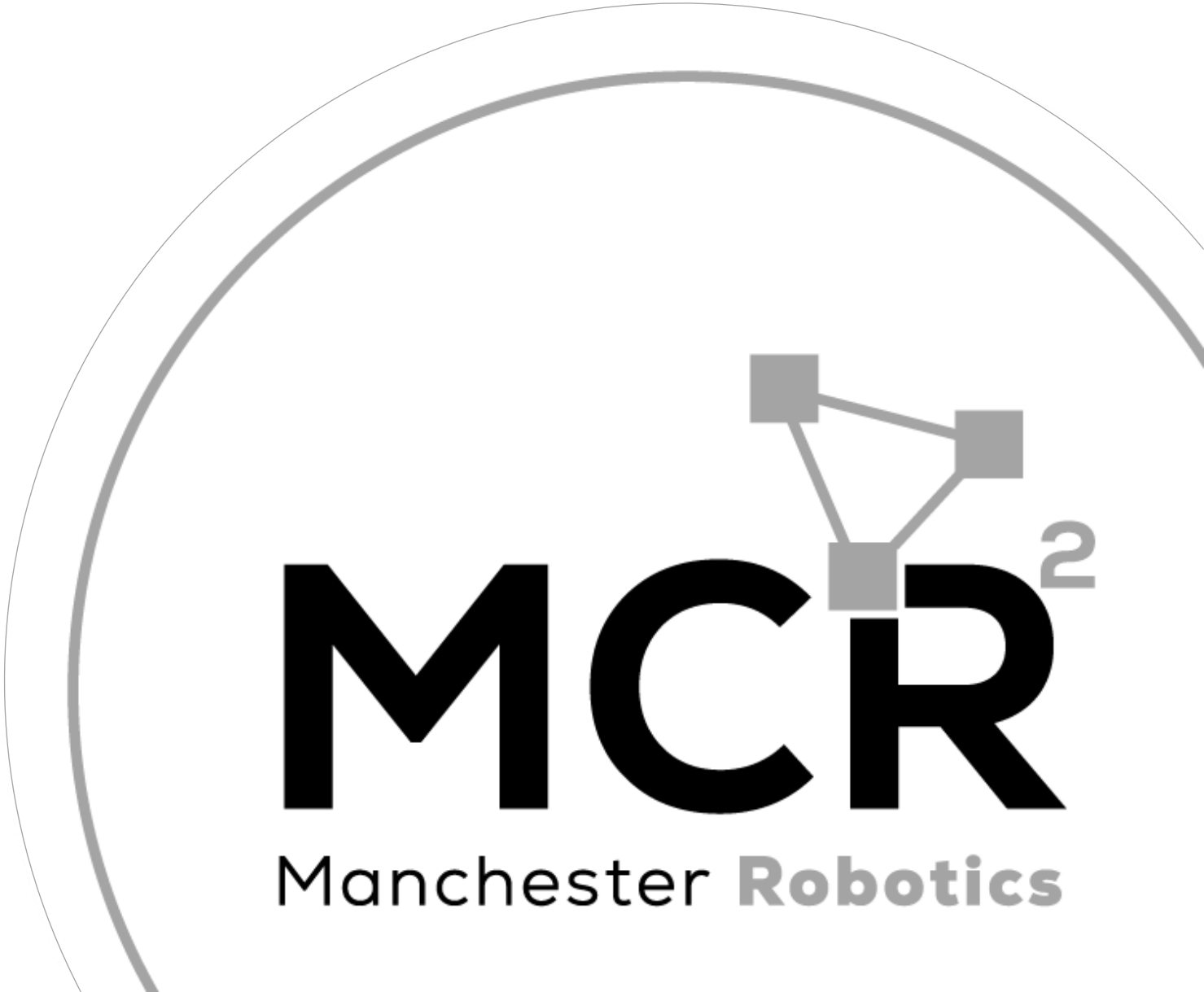
- The Following image, shows the result when using the global parameter.



Robot Operating System – ROS

Custom Messages

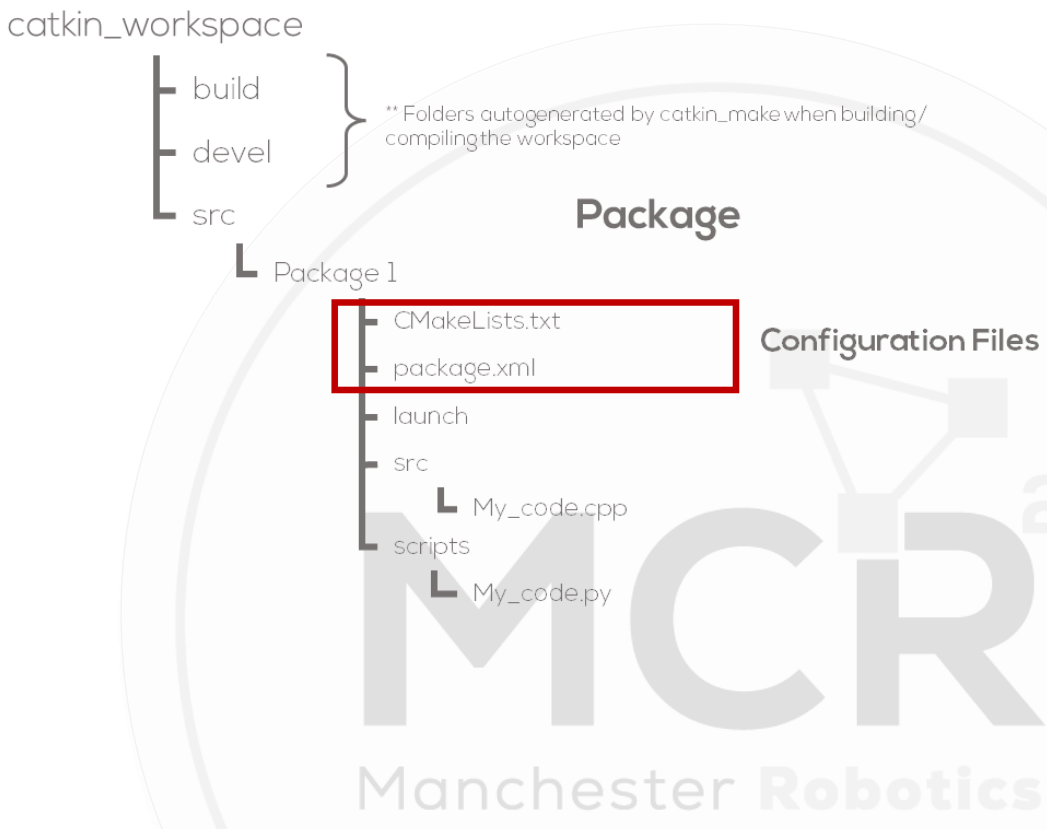
{Learn, Create, Innovate};



ROS Packages

Configuration Files

- As seen before, packages are the “buildable units” of ROS code. Therefore, to build a package it is necessary to build the entire package as one piece.
- In most projects, the user would require to build multiple packages.
- ROS has the package tool called Catkin to help build multiple packages at once.
    - Catkin, lets us group the packages into a workspace, and allows us to build all the packages inside the workspace.
- To do this, Catkin require different configuration files to properly define the dependencies and properties (meta information) for the different packages.
- Such packages are called CMakeLists.txt and package.xml



ROS Packages

CMake Files

- Since catkin, allows the user to compile different languages (like cpp). The compilers or executables usually require many libraries, external files or flags to the compiler to be able to compile and run a program.
- Build tools are used to make this process easier.
- Build tools, allow to describe the setup of the project (set of rules) at a higher level to generate the specific flags required by the program to run.
- In Linux these build tools are commonly known as make files.
- Make files sometimes require information from the system, like paths to libraries, etc. When sharing a project, the paths to those libraries may change.
- CMake is a tool (build system generator) that allow the user to deal with the issue by searching for the library paths other machines and generate the make file.

ROS Custom messages

- ROS has some predefined messages like the *std_messages, geometric_messages*, etc.
- Sometimes, the message structures are required to be altered for a custom application.
- ROS allows the user to customise the messages and create new messages.
- Custom messages are a way to personalise your own messages for a specific purpose or application.
- Custom messages are created by the user, and must be linked to the package where they will be used.
- More information here, here and here .

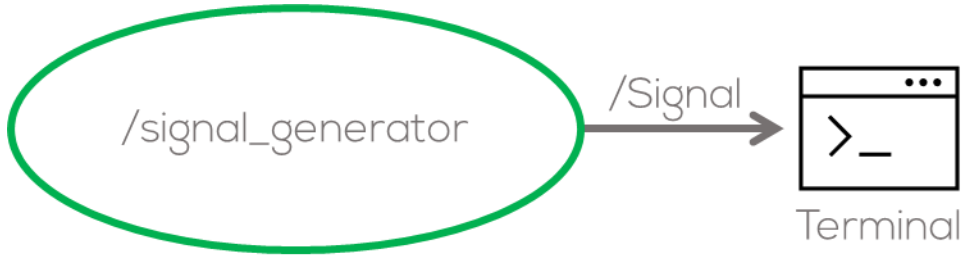
ROS Custom Message Example

This example will help the student to understand the concepts of custom messages, how to implement them and how to add them to the package.

- For this example, a simpler version of the previous activity (Signal Generator) code will be used.
  - It is encouraged for the students to use their coursework code and modify it to use a custom message.
- Since this is not a challenge, we will use the "basic_comms" package for this example. **The student can follow the same steps in any package**.

ROS Custom messages

- For this example, a new message will be generated to contain two different values of data.
- In comparison with the previous exercise where the user had to publish two different messages, each one in a different topic.
- For this example only one topic will be necessary, and the message will include all the information required in the previous exercise.




```
Custom msg  signal_msg
Float32 time_x
Float32 signal_y
```

ROS Custom messages

ROS Custom Message Creation

To start let's create the custom message

1. In the “basic_comms” package (or the student's own package), create a folder named “msg”
2. Inside the folder “msg” create a file called “signal_msg.msg”
3. Open the file using a text editor and write the following.

```
float32 time_x
float32 signal_y
```

4. Save the file. The custom message has been created!

ROS Custom Message Configuration

The following step is to tell ROS that a custom message will be used, where the message located and its dependencies.

To do this, the CmakeList.txt and the Package.xml will be modified (Follow each step carefully).

1. Open the CMakeLists.txt, of the “basic_comms” package (or the students’ package) and find the following line and modify it as follows

```
find_package(catkin REQUIRED COMPONENTS  rospy  std_msgs
   message_generation)
```

- This line is telling ROS to add the message_generation component (to create new messages), when compiling the program.

ROS Custom messages

2. Find the following lines, uncomment it and modify them as follows.

```
add_message_files(
   FILES
   signal_msg.msg
)


generate_messages(
  DEPENDENCIES
  std_msgs
)
```

- These lines tells ROS the name of the message files contained in the msg folder, and the dependencies of such messages (for more complex message you must add the dependencies here).

3. Modify the *catkin_package* line as follows. This line tells the dependencies used by catkin.

```
 catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES basic_comms
  CATKIN_DEPENDS message_runtime
#  DEPENDS system_lib
)
```

4. Done! The configuration for using a custom message has been finished.

ROS Custom messages

ROS Custom Message Usage

The following code, as stated before, is just a simple example on how to use the newly created message. This code is based on the previous challenge.

- Firstly, the new message must be imported from the library

```
from basic_comms.msg import signal_msg
```

- Define the new publisher and its attributes, to publish the new message as follows.

```
signal_pub=rospy.Publisher("signal",signal_msg, queue_size=10)
```

- Finally, fill out the message with the data to be sent and publish it as follows

```
msg.time_x = time
msg.signal_y = signal

signal_pub.publish(msg)
```

```
#!/usr/bin/env python
import rospy
import numpy as np
from basic_comms.msg import signal_msg  #Import the message to be used

if _name=='__main_':

    ## Declare the new message to be used
    signal_pub=rospy.Publisher("signal",signal_msg, queue_size=10)

    rospy.init_node("signal_generator")
    rate = rospy.Rate(10)
    init_time = rospy.get_time()
    msg = signal_msg()

    while not rospy.is_shutdown():
        time = rospy.get_time()-init_time
        signal = np.sin(time)

        ## Fill the message with the required information
        msg.time_x = time
        msg.signal_y = signal

        ## Publish the message
        signal_pub.publish(msg)

        rospy.loginfo("The signal value is: %f at a time %f", signal, time)
        rate.sleep()
```

ROS Custom messages

ROS Custom Message Usage

- Save the previous file and close it. (do not forget to make it executable *`sudo chmod +x file.py`*)
- Recompile the workspace using "*`catkin_make`*".
- Open a new terminal and start a *`roscore`*.
- Open another terminal and print the topics using *`rostopic list`*
- Echo the topic (*`rostopic echo /signal`*) to verify if the information is being sent.
- The following results are expected.

```
student@ubuntu:~/catkin_ws$ rostopic list
/rosout
/rosout_agg
/signal
student@ubuntu:~/catkin_ws$ rostopic echo /signal
time_x: 27.50098419189453
signal_y: 0.6985356211662292
---
time_x: 27.600685119628906
signal_y: 0.6238420009613037
---
time_x: 27.700420379638672
signal_y: 0.5429235100746155
---
time_x: 27.801164627075195
signal_y: 0.4557102620601654
---
time_x: 27.900787353515625
signal_y: 0.3649194538593292
---
time_x: 28.00063133239746
signal_y: 0.27029848098754883
---
```

```
^Cstudent@ubuntu:~/catkin_ws$ rosrun basic_comms signal_generator.py
[INFO] [1670443755.638832]: The signal value is: 0.000009 at a time 0.000009
[INFO] [1670443755.739726]: The signal value is: 0.100815 at a time 0.100986
[INFO] [1670443755.839710]: The signal value is: 0.199555 at a time 0.200903
[INFO] [1670443755.939380]: The signal value is: 0.296138 at a time 0.300647
[INFO] [1670443756.039081]: The signal value is: 0.389734 at a time 0.400342
[INFO] [1670443756.139903]: The signal value is: 0.480423 at a time 0.501137
[INFO] [1670443756.239789]: The signal value is: 0.565372 at a time 0.600884
[INFO] [1670443756.339344]: The signal value is: 0.644636 at a time 0.700547
[INFO] [1670443756.440031]: The signal value is: 0.718255 at a time 0.801291
[INFO] [1670443756.539740]: The signal value is: 0.783903 at a time 0.900927
[INFO] [1670443756.639406]: The signal value is: 0.841832 at a time 1.000668
[INFO] [1670443756.739007]: The signal value is: 0.891333 at a time 1.100277
[INFO] [1670443756.840123]: The signal value is: 0.932515 at a time 1.201317
```

ROS Activity

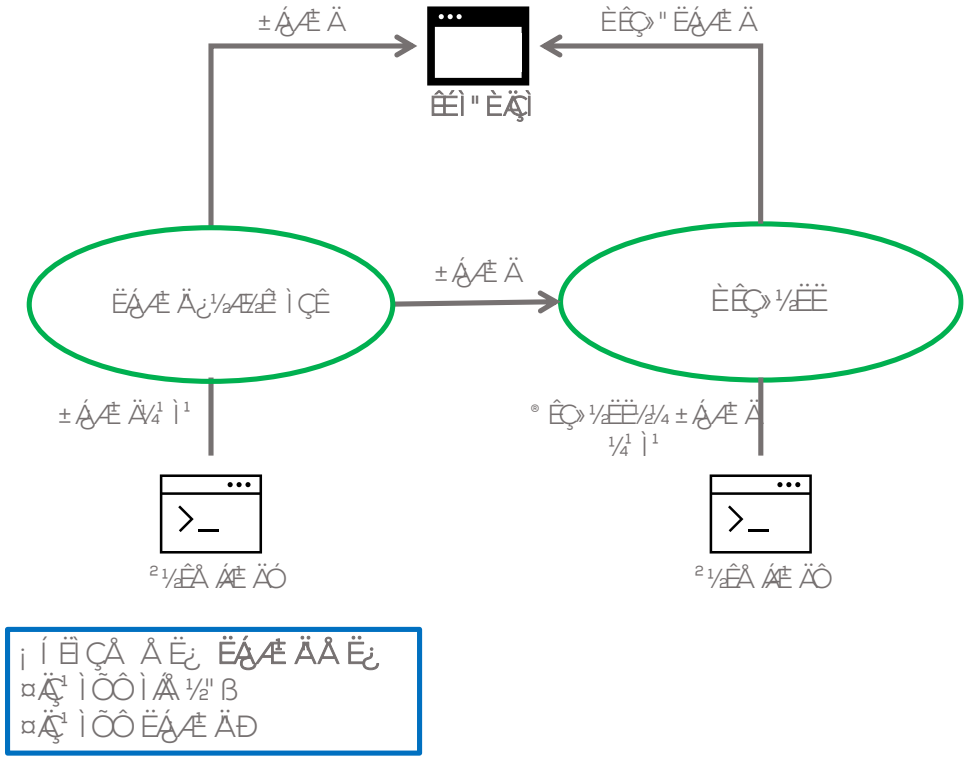
Activity

- Parametrise the “process” and “signal_generator” Nodes (created in the previous challenge) subscribe to the new topic and use the custom message.
- Make a roslaunch file for both nodes.

Hints: import the signal message and don’t forget to declare the variable (to be global) appropriately inside the call-back function

```
from basic_comms.msg import signal_msg
```

```
signal_data = 0
time_data = 0

def callback(msg):
    global signal_data, time_data
    signal_data = msg.signal_y
    time_data = msg.time_x
```



Q&A

Questions?

{Learn, Create, Innovate};

MCR² Manchester Robotics

{Learn, Create, Innovate};

Thank you