Introduction to ROS

Introducing

What is ROS?

- An **open-source, meta-operating system for robots**
- Provides **services like** hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management
- Also provides **tools and libraries for obtaining, building, writing, and running code across multiple computers**
- Unlike conventional operating systems, it can be **used for numerous combinations of hardware implementation**
- A robot software platform that **provides various development environments specialized for developing robot application programs**

ROS is a Meta-Operating System

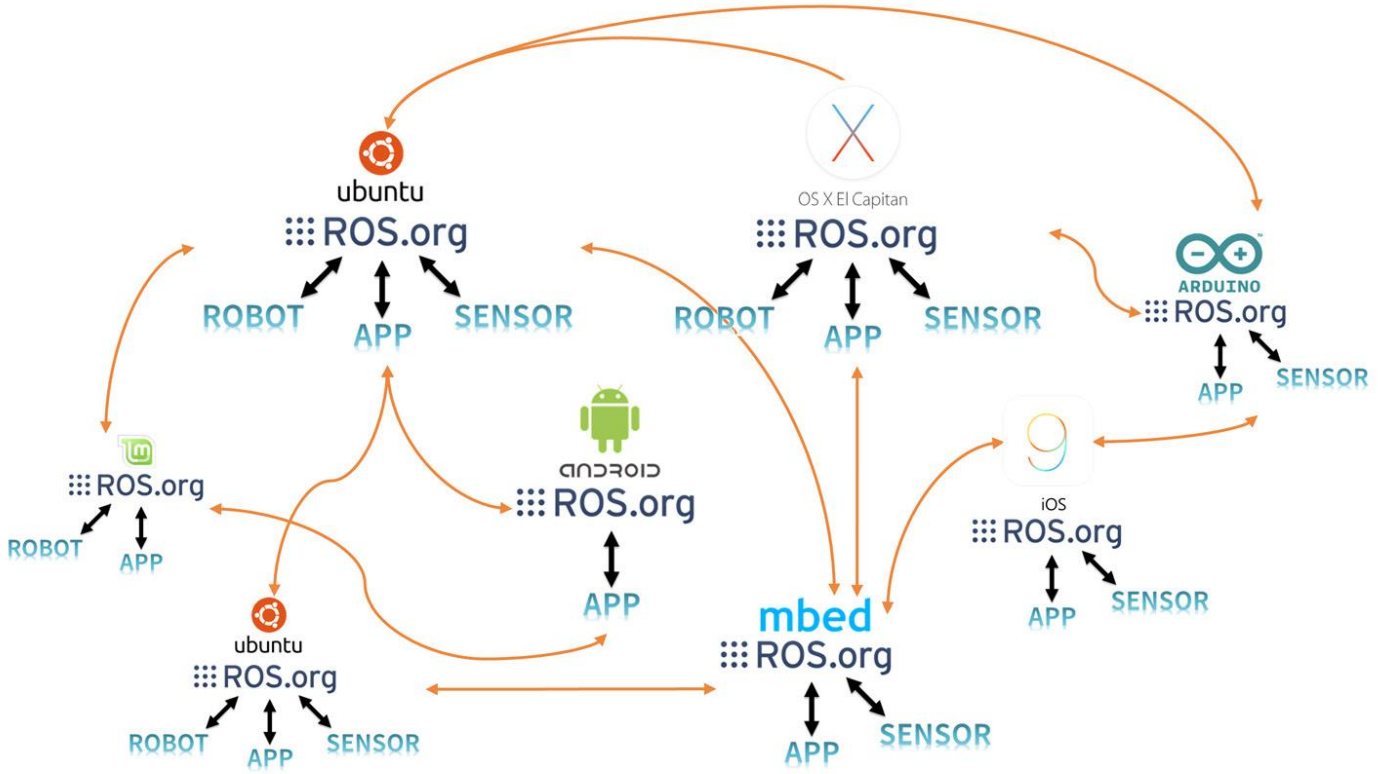
What is a Meta-Operating System?

- Operating Systems (OS) for general purpose computers include Windows, Linux and Mac. For smartphones, there are Android, iOS, etc
- ROS is **NOT a conventional OS;** to be accurate, **ROS is a meta-operating system that runs on existing OS** (confusion with conventional OS)
- Formally defined as "***a system that performs processes such as scheduling, loading, monitoring and error handling by utilizing virtualization layer between applications and distributed computing resources***"

ROS is a Meta-Operating System

ROS is **a supporting system** for **controlling a robot and sensor with a hardware abstraction** and for **developing robot application based on existing conventional OSs**

ROS is a Meta-Operating System



ROS **data communication is supported not only by one OS, but also by multiple operating systems, hardware, and programs**, making it highly suitable for robot development where various hardware are combined

ROS Components

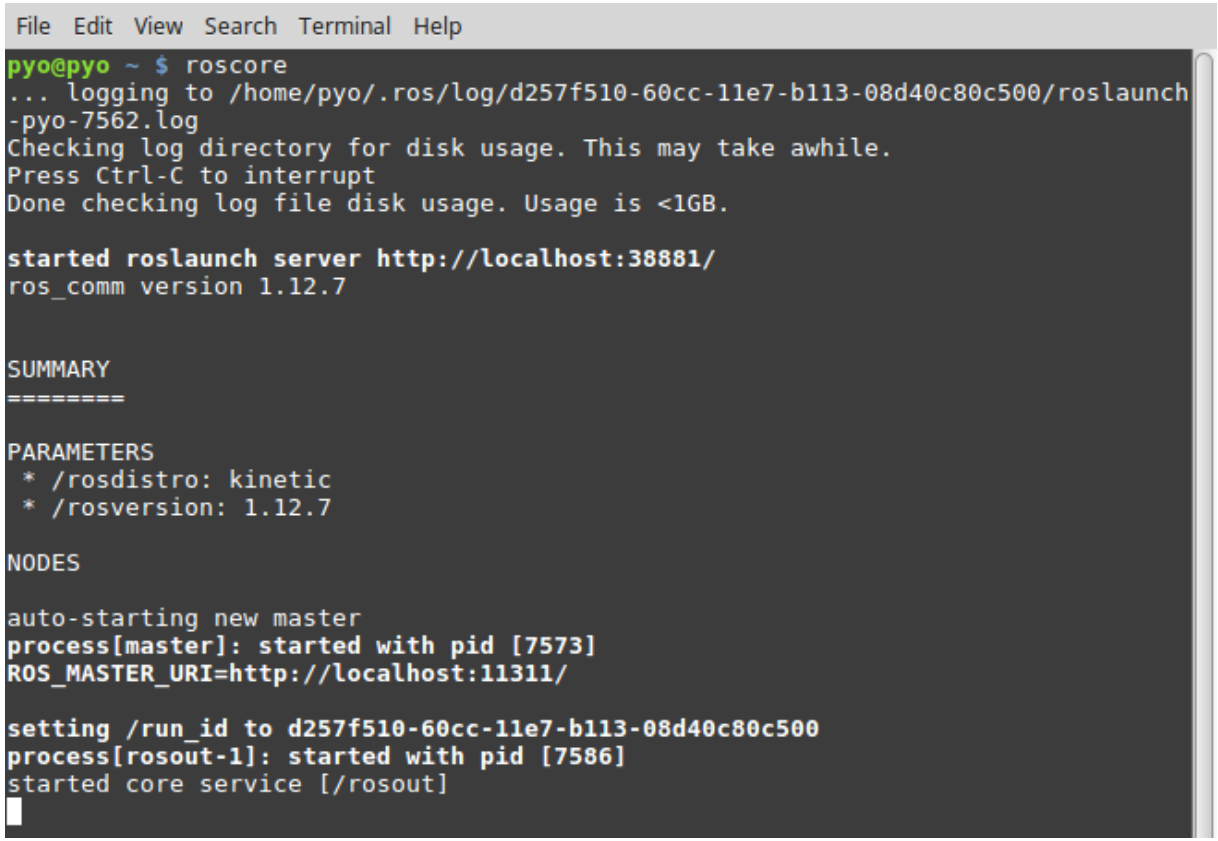
ROS Operation Test

To test if ROS is installed properly and works correctly (Let's Try!):

1. Open a new terminal window (*Ctrl + Alt + t*) and **enter `$ roscore`**
2. **Run turtlesim_node** in the turtlesim package
3. **Run turtle_teleop_key** in the turtlesim package
4. **Run rqt_graph** in the rqt_graph package
5. **Close the node** (Ctrl+c)

*Installation of ROS will NOT be covered in this course; if you are interested, refer to workshop materials

Run `roscore`



Screen showing roscore running

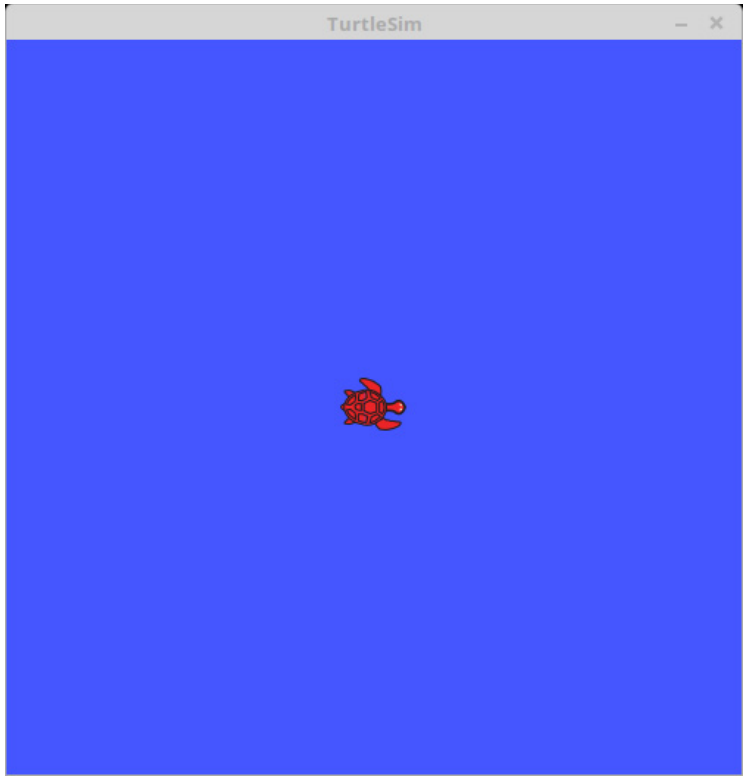
*roscore is the command that runs the ROS master (aka server)

Run turtlesim_node

Open a new terminal window and enter:

```
$ rosrun turtlesim turtlesim_node
```

A window will pop out with a turtle in the middle:



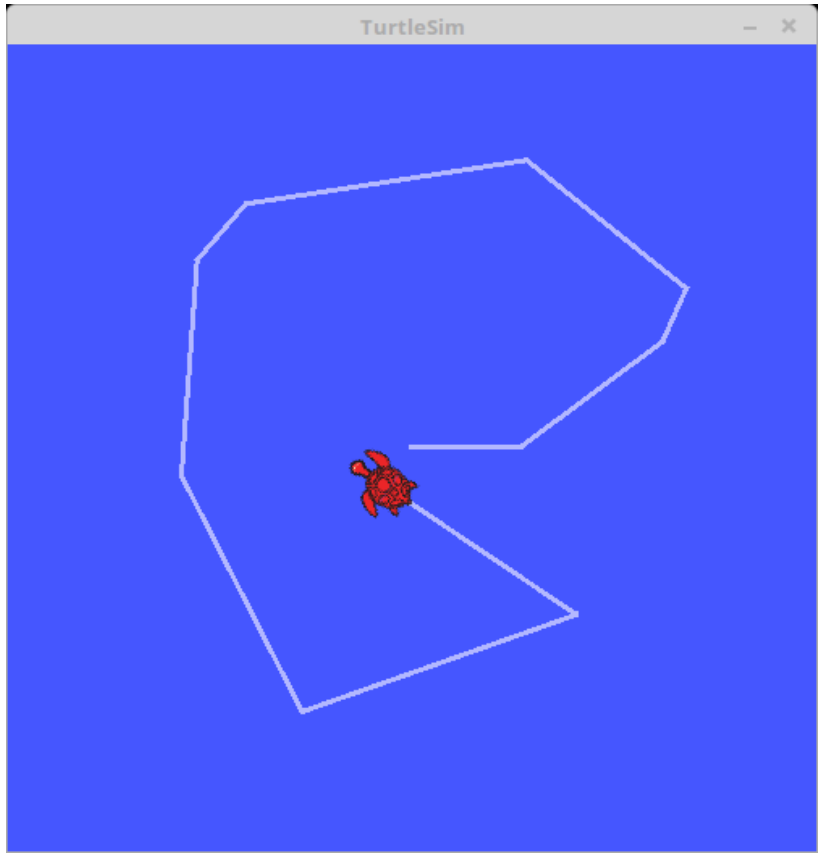
*rosrun is the basic execution command of ROS; used to run a single node in the package

Run turtle_teleop_key

Open a new terminal window and enter:

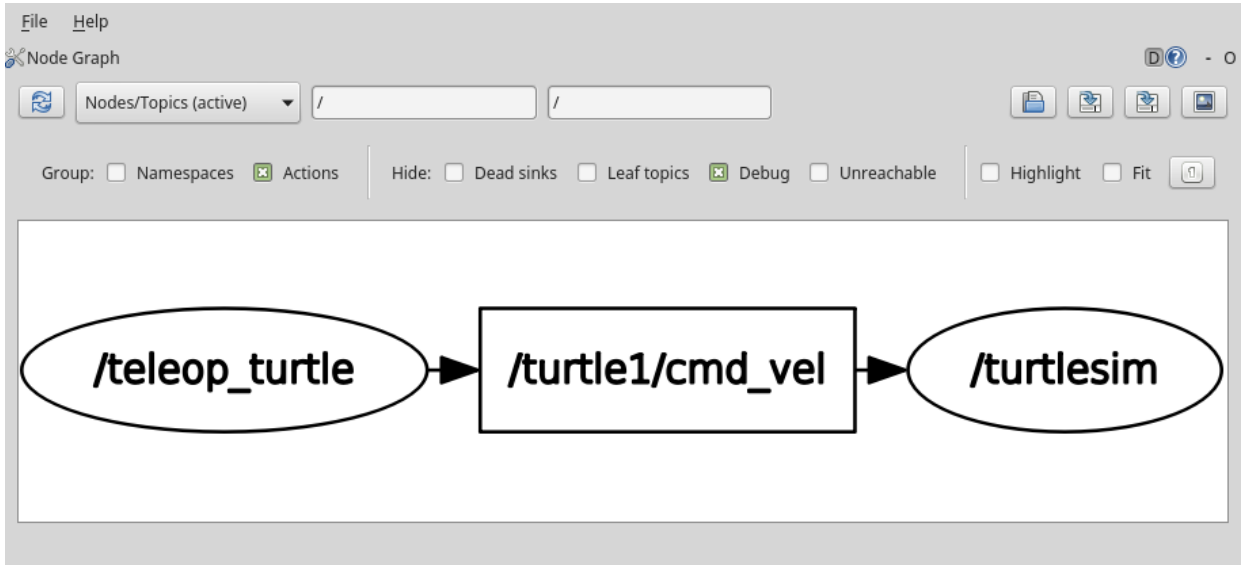
```
$ rosrun turtlesim turtle_teleop_key
```

You can use any of arrow keys on the keyboard (←, → , ↑, ↓) to move the turtle



Run rqt_graph

Open a new terminal window and enter: `$ rqt_graph`



After done, at each terminal, enter Ctrl+c to close all the programs

- The rqt_graph node shows **information about the currently running nodes in a GUI form**;
- **Circle represents a node** (program) and **square represents a topic** (message communication)

Download & Install Robotics ROS Package

Steps:

1. Change to the source space directory of the catkin workspace:

```
$ cd ~/catkin_ws/src
```

2. Git Clone the Robotics package:

```
$ git clone https://github.com/nicholashojunhui/robotics.git
```

3. Build the packages in the catkin workspace:

```
$ cd ~/catkin_ws && catkin_make
```

4. Go to *catkin_ws/src/robotics/src* and make all python files executable

Creating a new ROS Package (Optional)

Steps:

1. Change to the source space directory of the catkin workspace:

```
$ cd ~/catkin_ws/src
```

2. Use the catkin_create_pkg script to create a new package called '***new_package***' which depends on std_msgs, roscpp, and rospy:

```
$ catkin_create_pkg new_package std_msgs rospy roscpp
```

3. Build the packages in the catkin workspace:

```
$ cd ~/catkin_ws
$ catkin_make
```

4. Add the workspace to your ROS environment by sourcing the generated setup file:

```
$ . ~/catkin_ws/devel/setup.bash
```

Definitions:

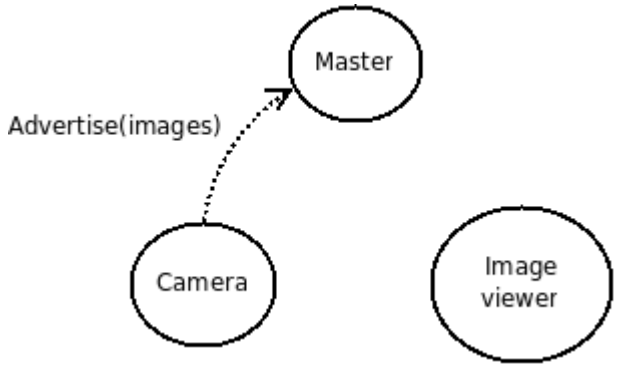
- A **ROS Topic** is a channel for communicating Messages among the Nodes
- A ROS Topic has a fixed Message type
- A Node **publishes** on a Topic to broadcast Messages
- A Node **subscribes** to a Topic to get Messages from other Nodes
- *Many-to-many* communication
  - One Node can publish/subscribe multiple Topics
  - One Topic can be published/subscribed by different Nodes
- **ROS Services** are another way that nodes can communicate with each other
- ROS Services allow nodes to send a request and receive a response

ROS Master

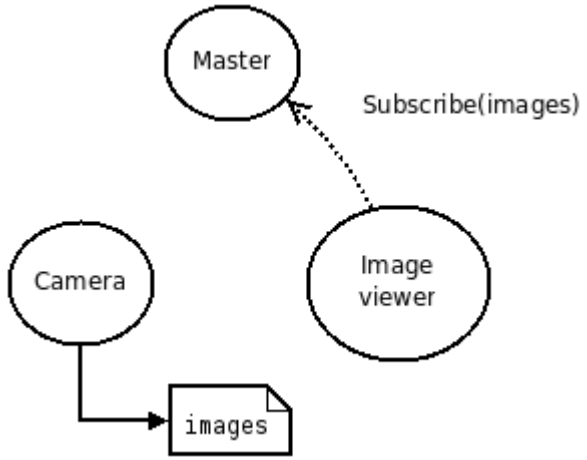
- The ROS Master **provides naming and registration services to the rest of the nodes** in the ROS system
- It **tracks publishers and subscribers to topics as well as services**
- Its role is to **enable individual ROS nodes to locate one another**
- Once these nodes have located each other via the ROS Master, they communicate with each other peer-to-peer
- Run using the **roscore command**, which loads the ROS Master

ROS Master (Example)

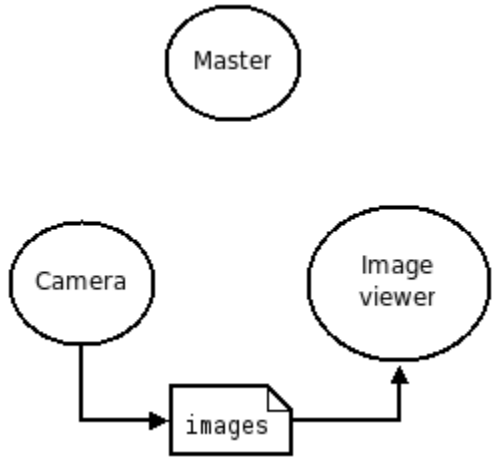
Let's say we have two Nodes: a Camera node and an Image_viewer node



The Camera will notify the master that it wants to publish images on the topic "images"

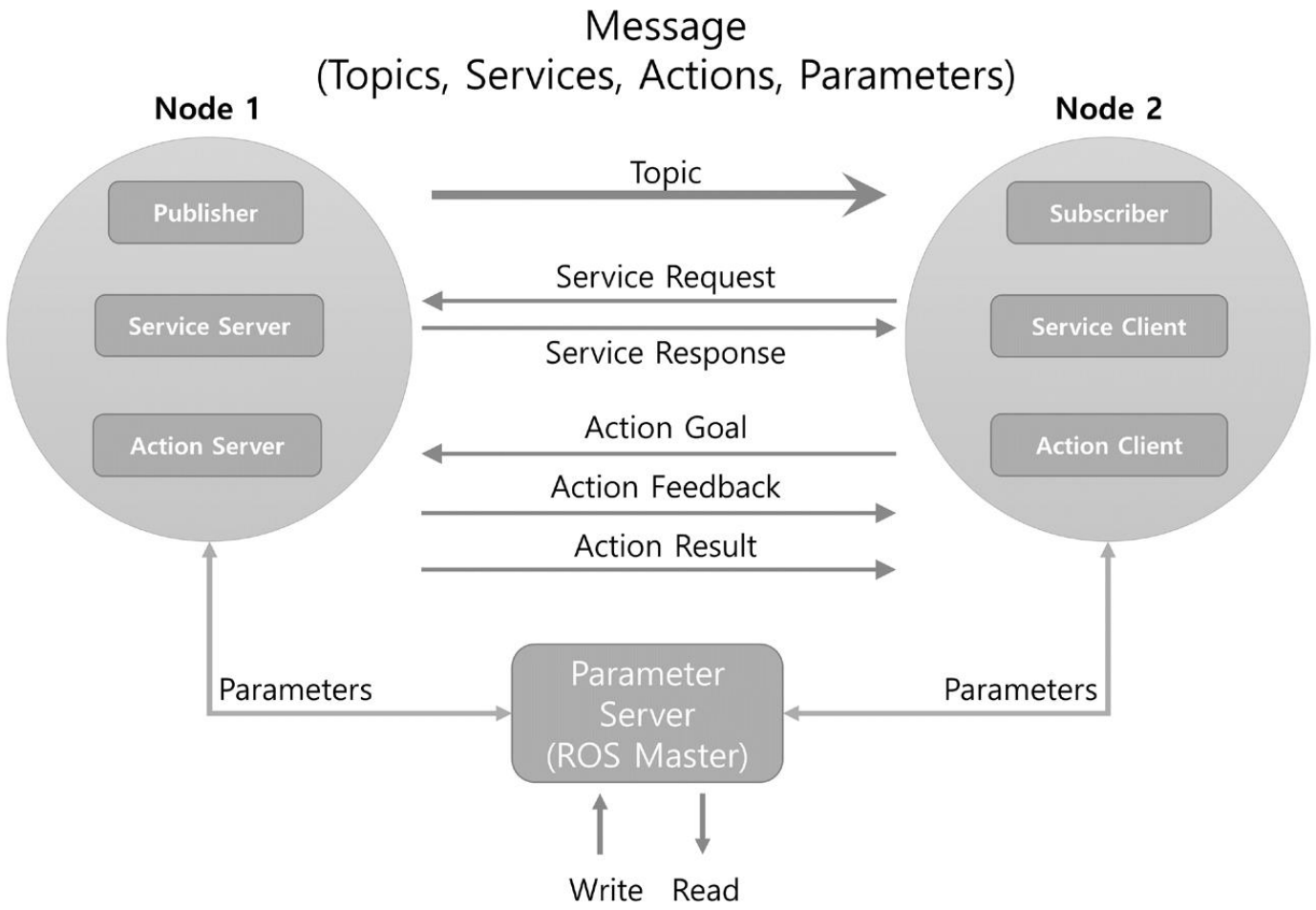


- The Camera will publish images to the "images" topic, but nobody is subscribing to that topic yet so no data is actually sent
- Same time, Image_viewer will subscribe to the topic "images" to see if there is any images there

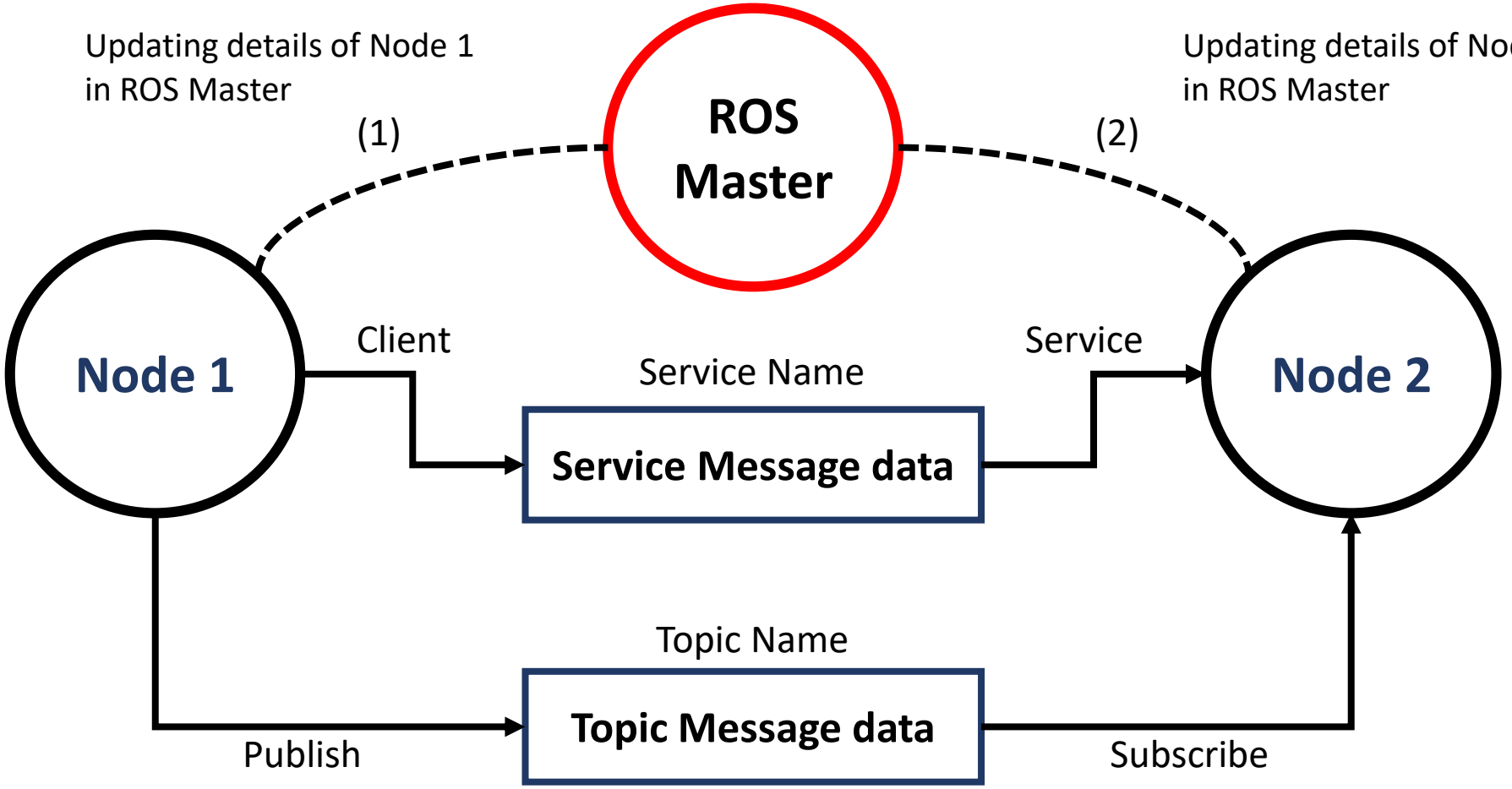


Once the topic "images" has both a publisher and a subscriber, the master node notifies Camera and Image_viewer about each others existence so that they can start transferring images to one another

ROS Communication Block Diagram:



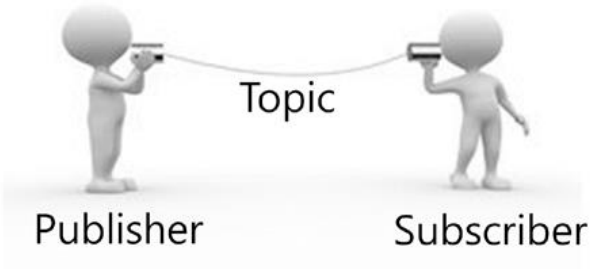
ROS Communication Block Diagram (simplified):

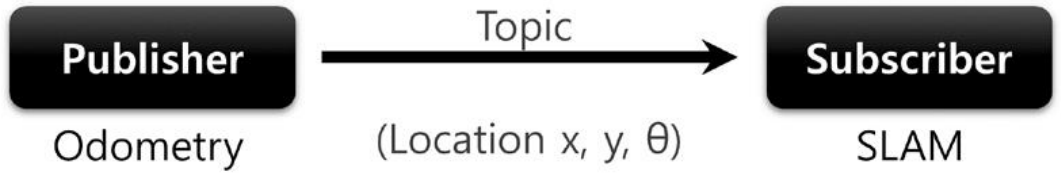


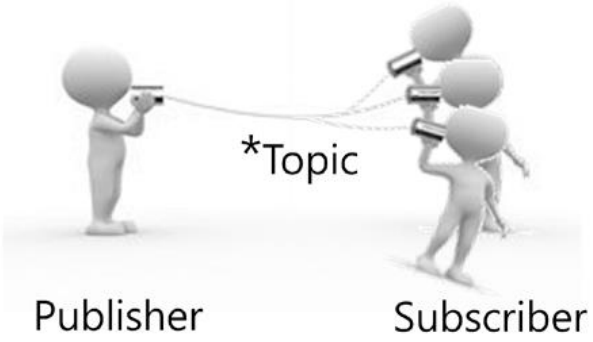
Comparison of Topic, Service and Action

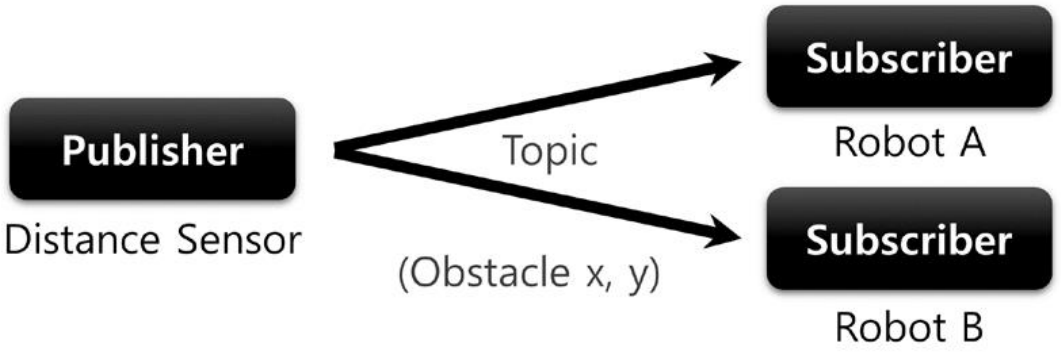
Type	Features		Description
Topic	Asynchronous	Unidirectional	Used when exchanging data continuously
Service	Synchronous	Bi-directional	Used when request processing requests and responds current states
Action	Asynchronous	Bi-directional	Used when it is difficult to use the service due to long response times after the request or when an intermediate feedback value is needed

Topic Message Communication



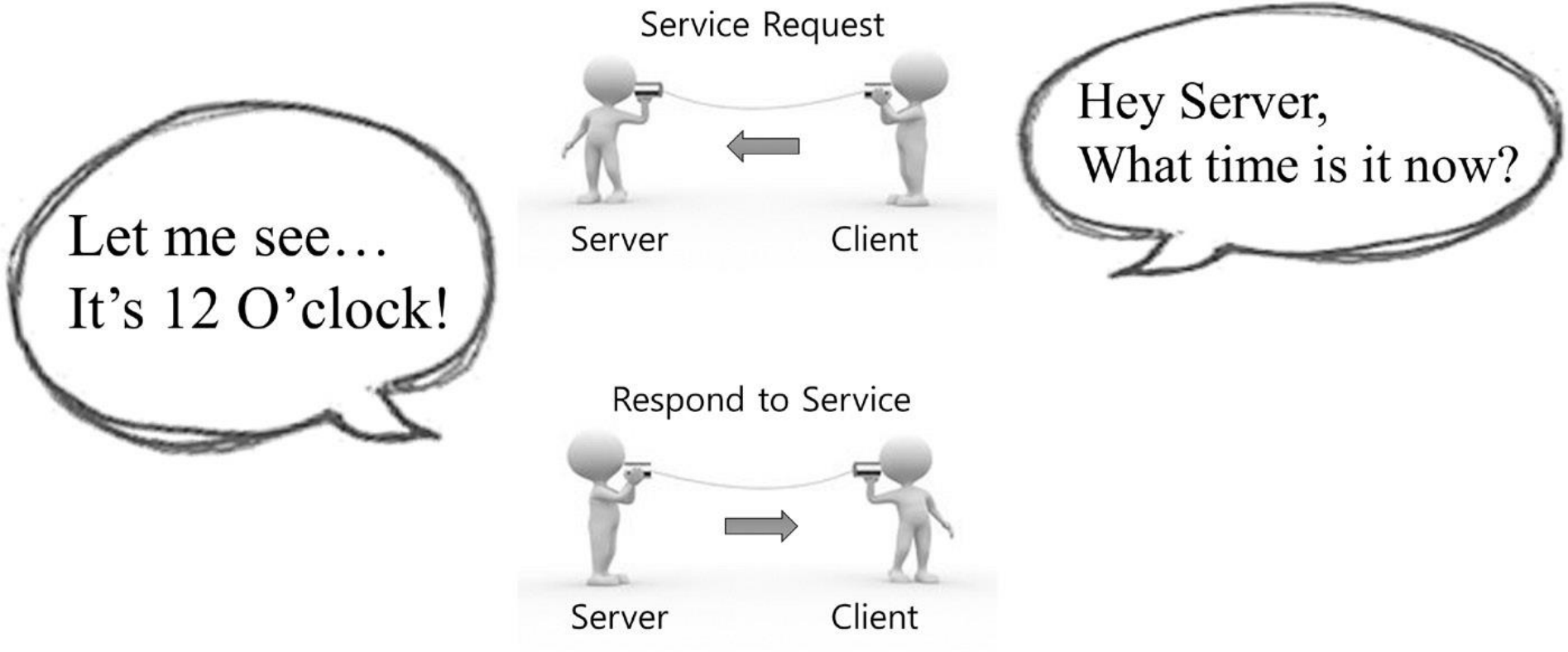




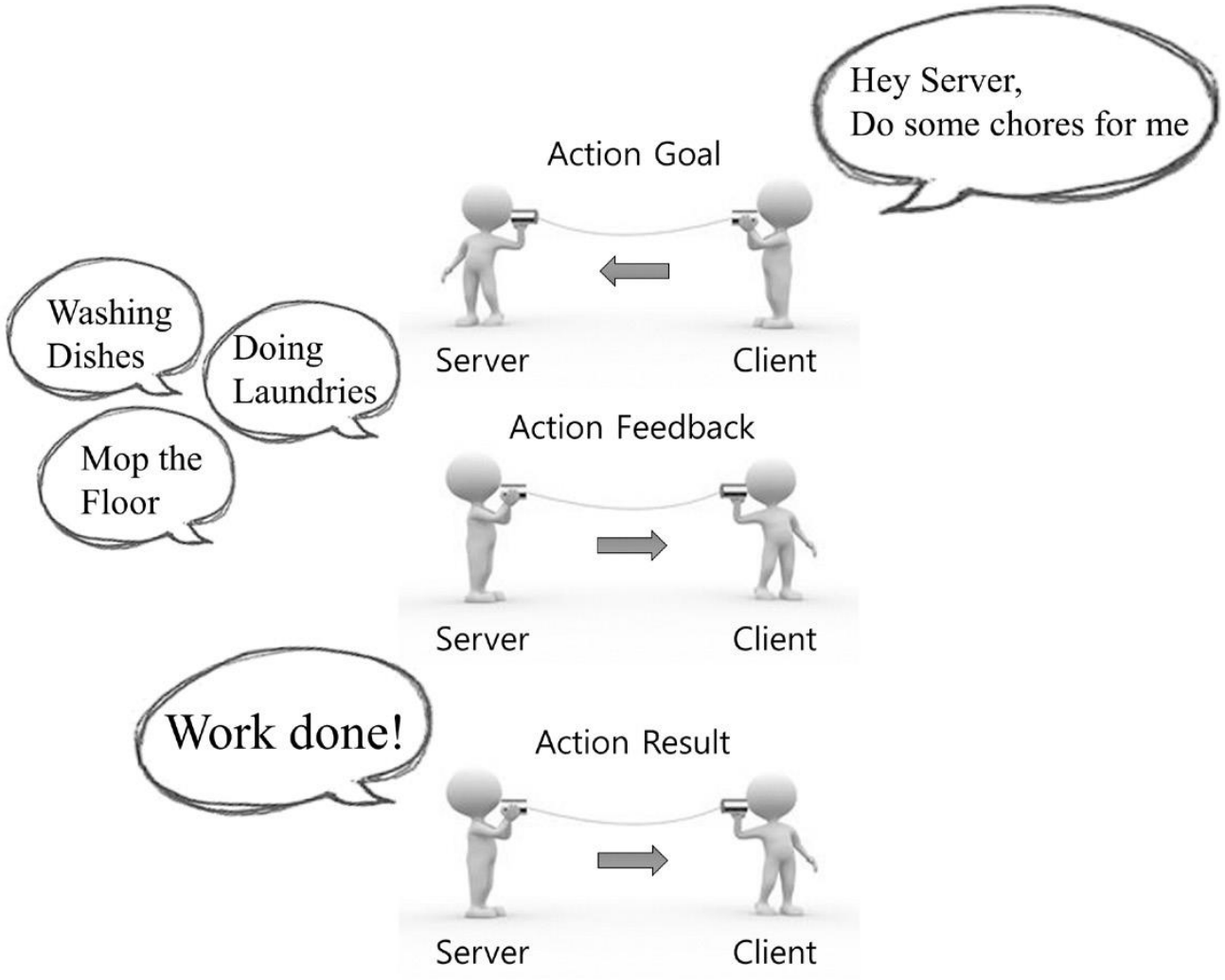


*Topic not only allows 1:1 Publisher and Subscriber communication, but also supports 1:N, N:1 and N:N depending on the purpose.

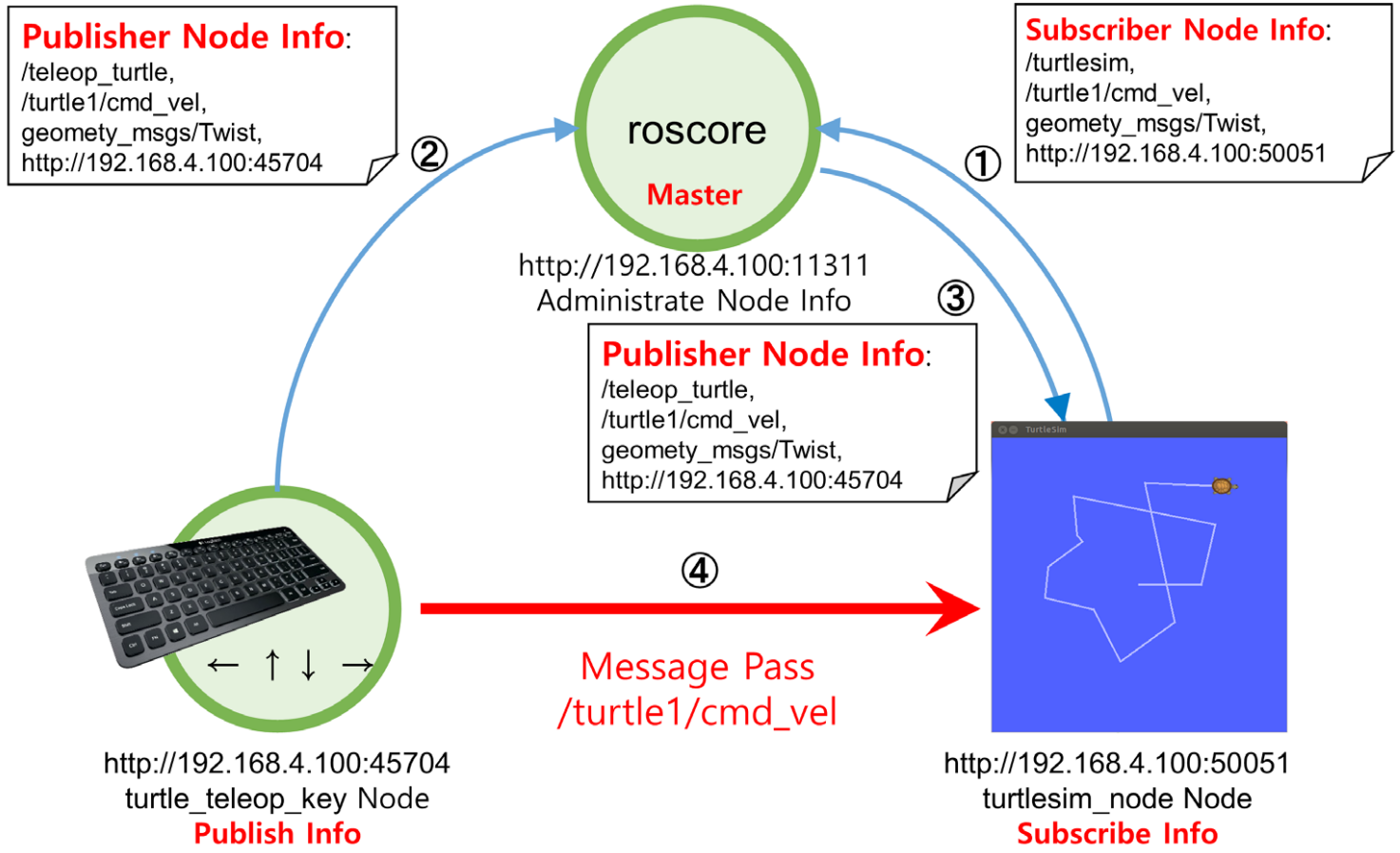
Service Message Communication



Action Message Communication



Example of Message Communication



Roslaunch vs Rosrun

- **Rosrun** is a command that runs only one node in the specified package

```
rosrun [PACKAGE_NAME] [NODE_NAME]
```

- **Roslaunch** is a command that executes more than one node in the specified package or sets execution options

```
roslaunch [PACKAGE_NAME] [launch_FILE_NAME]
```

ROS BASICS (General)

First, run roscore and the turtlesim_node

```
$ roscore
```

```
$ rosrun turtlesim turtlesim_node
```

We will open new terminal and try the following (one by one):

```
1.$ rostopic list
```

```
2.$ rostopic type /turtle1/cmd_vel
```

```
3.$ rosmsg show geometry_msgs/Twist
```

ROS BASICS (General)

Next, open new terminal and try the following to move the turtlebot with the inputs:

```
$ rostopic pub /turtle1/cmd_vel geometry_msgs/Twist "linear:
    x:1
    y:0
    z:0
angular:
    x:0
    y:0
    z:1.57"
```

***Note that you do not need to enter the complete command. Use the Tab key to autocomplete the command. Just type **rostopic pub /turtle1/cmd_vel**, and keep pressing the Tab key to autocomplete other fields. Don't forget to change the linear.x and angular.z values first before pressing the Enter key

ROS BASICS (Python)

Instead of using the above commands to move the turtlebot, we can write a Python code (i.e. move_turtle.py) to do the same movements; to run the Python code, follow the steps:

```
$ roscore
```

```
$ rosrun turtlesim turtlesim_node
```

```
$ rosrun robotics move_turtle.py 1.0 1.57
```

You will get the same output as the previous slide commands when you run this code. It creates a circle. You can change the input values to your liking

ROS BASICS (Python)

Instead of using only the input velocities to move the turtlebot, we can write a Python code (i.e. move_distance.py) based on stated position; to run the Python code, follow the steps:

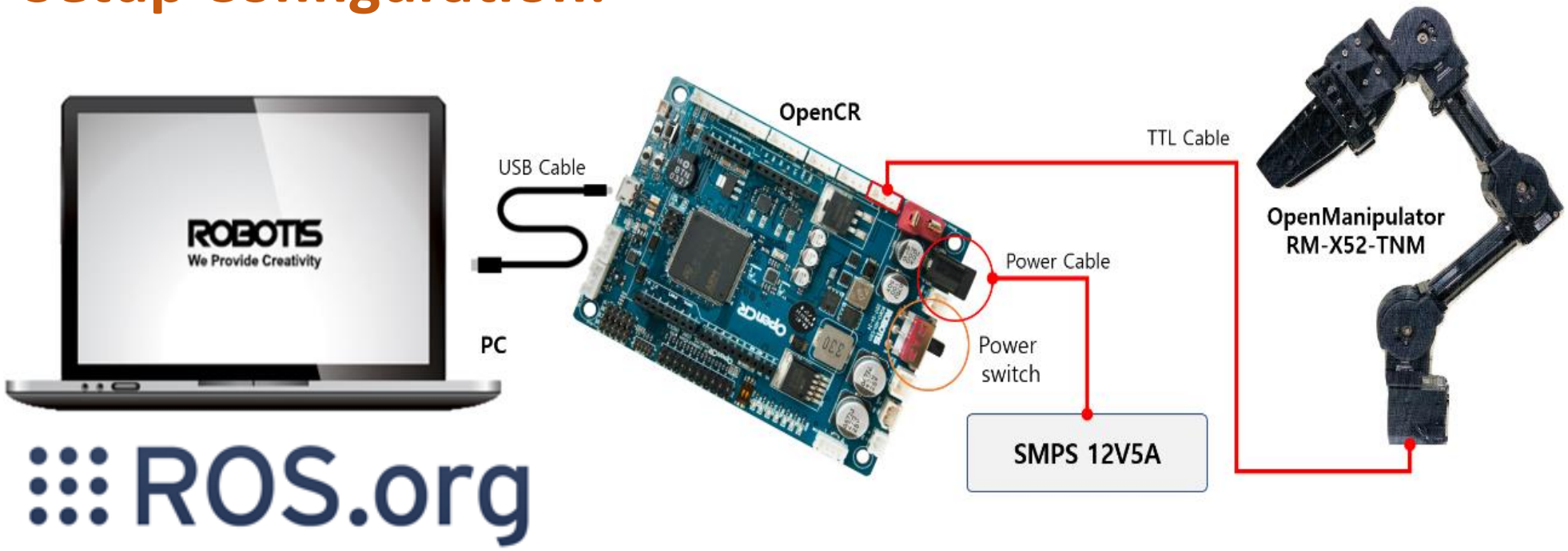
```
$ roscore
$ rosrun turtlesim turtlesim_node
$ rosrun robotics move_distance.py 1.0 0.0 8.0
```

This code makes the turtle moves forward until x = 8.0m at a velocity of 1m/s. You can change the input values to your liking. Note that the initial position of the turtle is [x=5.54, y=5.54, theta=0.0]

Using ROS to Control OM

Using ROS as the Robot Controller

Setup Configuration:

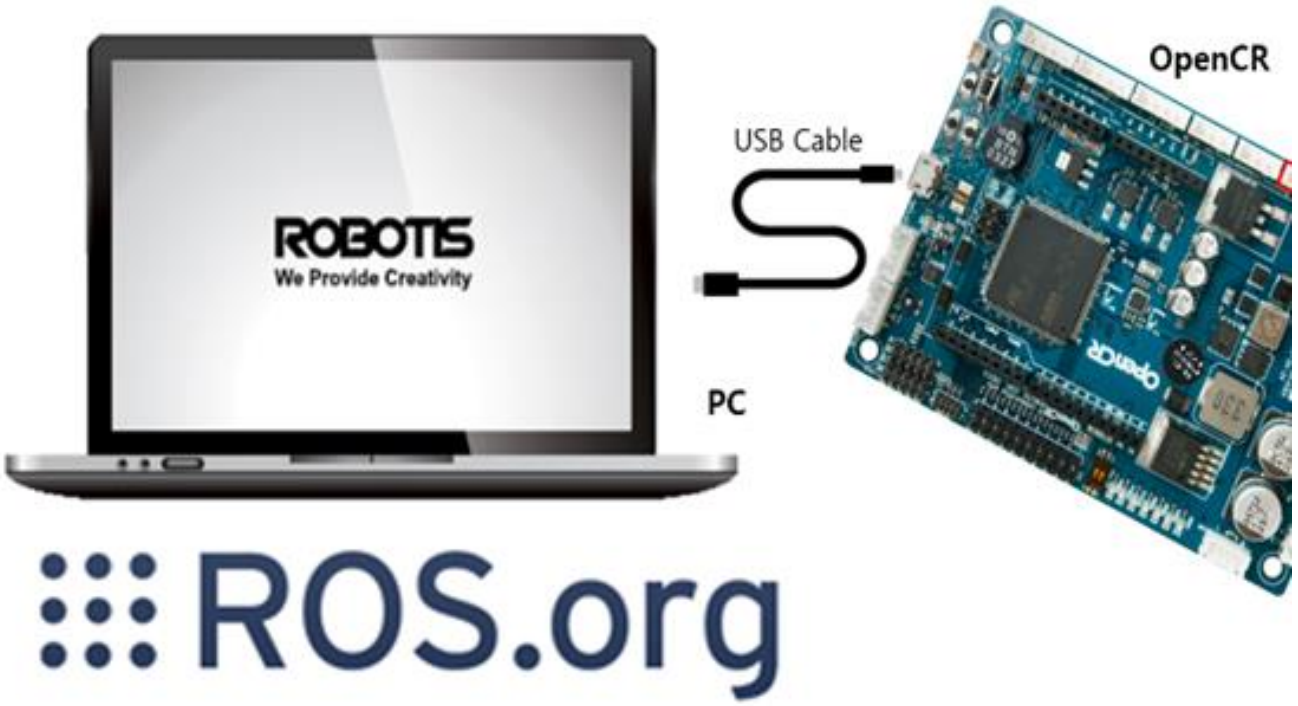


5 Steps to do before you can start using the OM Robotic Arm

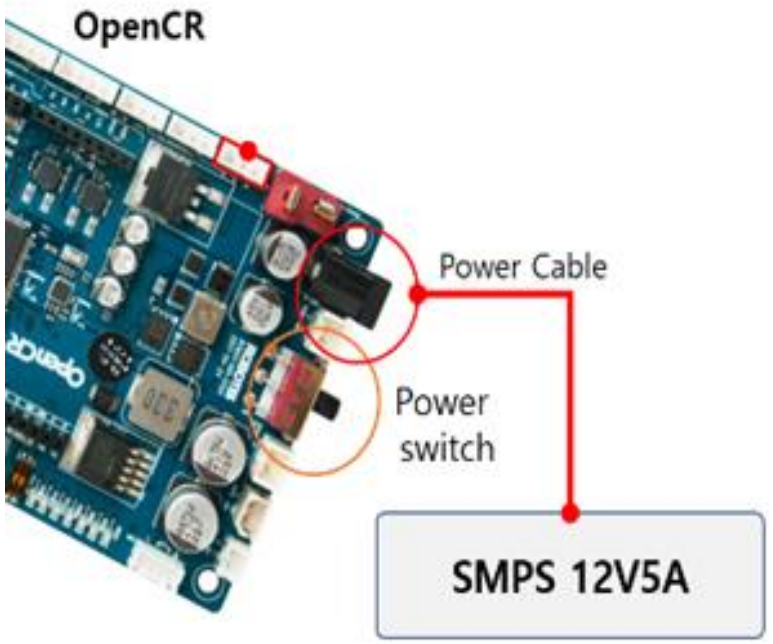
1. **Connect the board to your computer** using the **USB cable** (i.e. the USB extension cable)
2. **Connect the board and the power source** using the **power cable** (Do NOT switch on the power first)
3. **Open the Arduino IDE software and do the board setup** (only need to do this once)
4. **Launch the controller codes**
5. **You are ready to control the OM Robotic Arm using ROS software** (there are various methods to do so; we will go through 3 methods: manual, MoveIt! application, and python methods)

*Note that the motors and the board are already connected for you

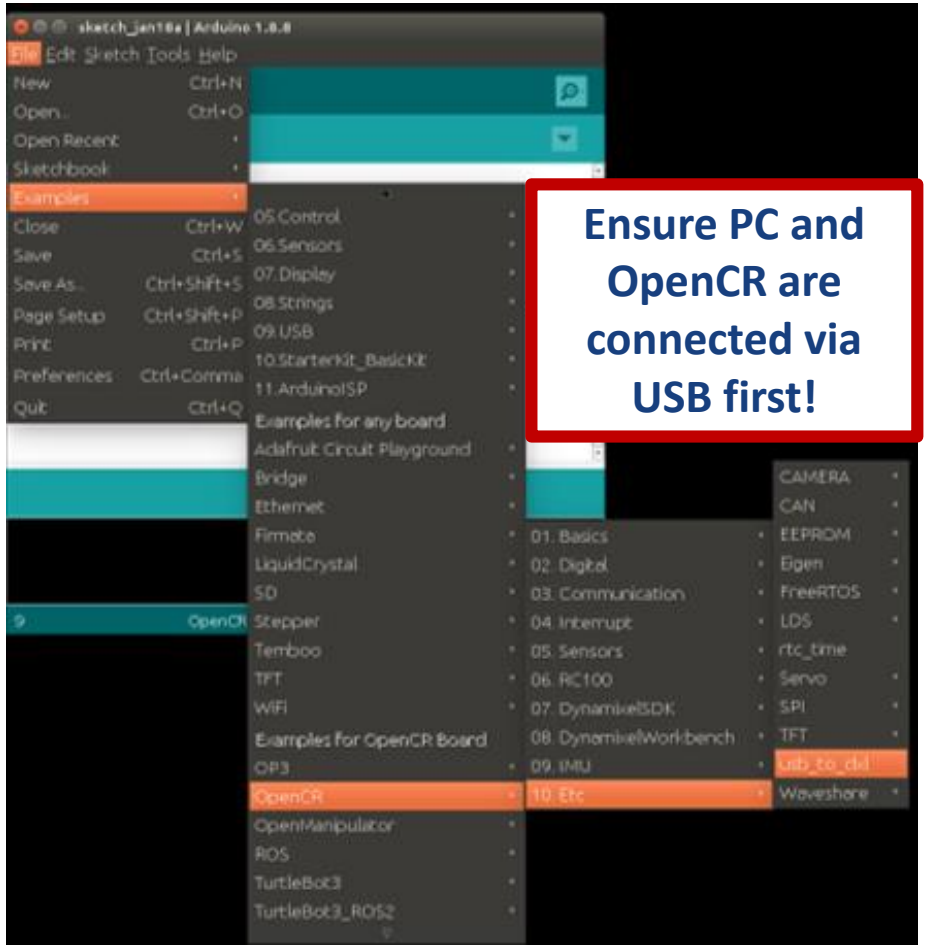
Step 1: Connect the board to your computer using the USB extension cable



Step 2: Connect the board and the power source using the power cable
(Do NOT switch on the power first)

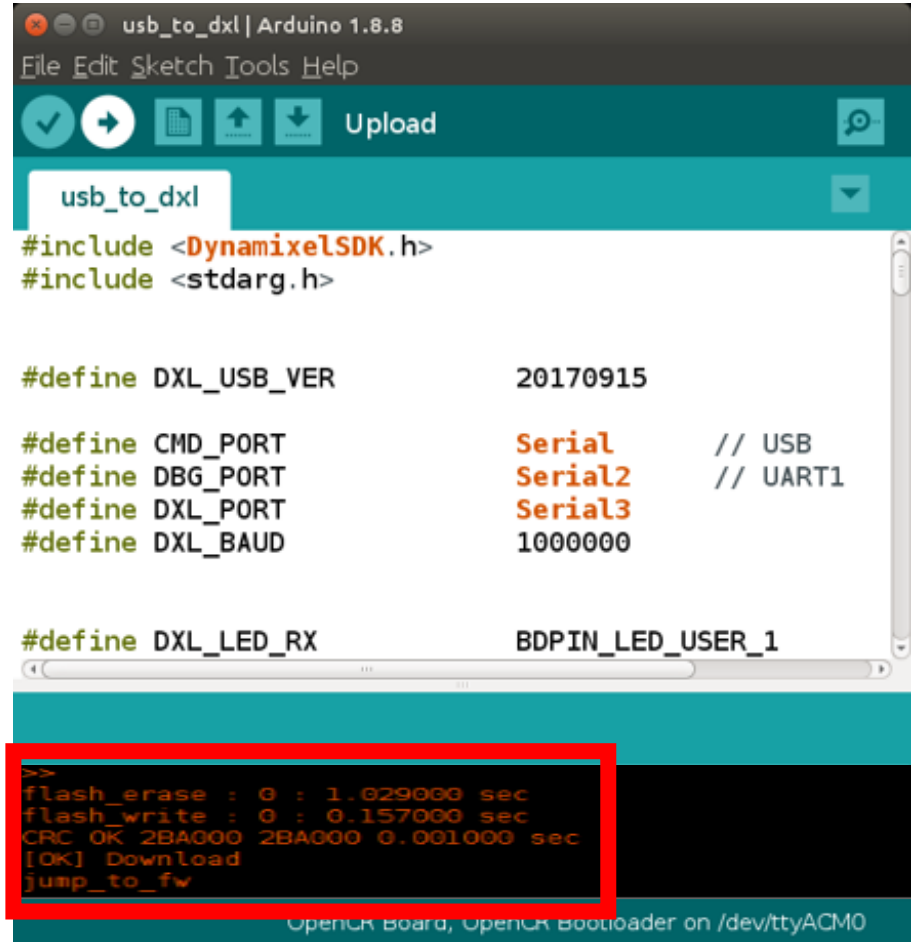


Step 3: OpenCR Setup (DONE!)



1. Open Arduino IDE software in your PC
2. Go to `Examples` → `OpenCR` → `10.Etc` → `usb_to_dxl` to open the `usb_to_dxl` example source code

Step 3: OpenCR Setup (DONE!)



3. Click Upload. If launched successfully, the terminal will appear as follows (left)
   - `jump_to_fw` will appear at the end of the line

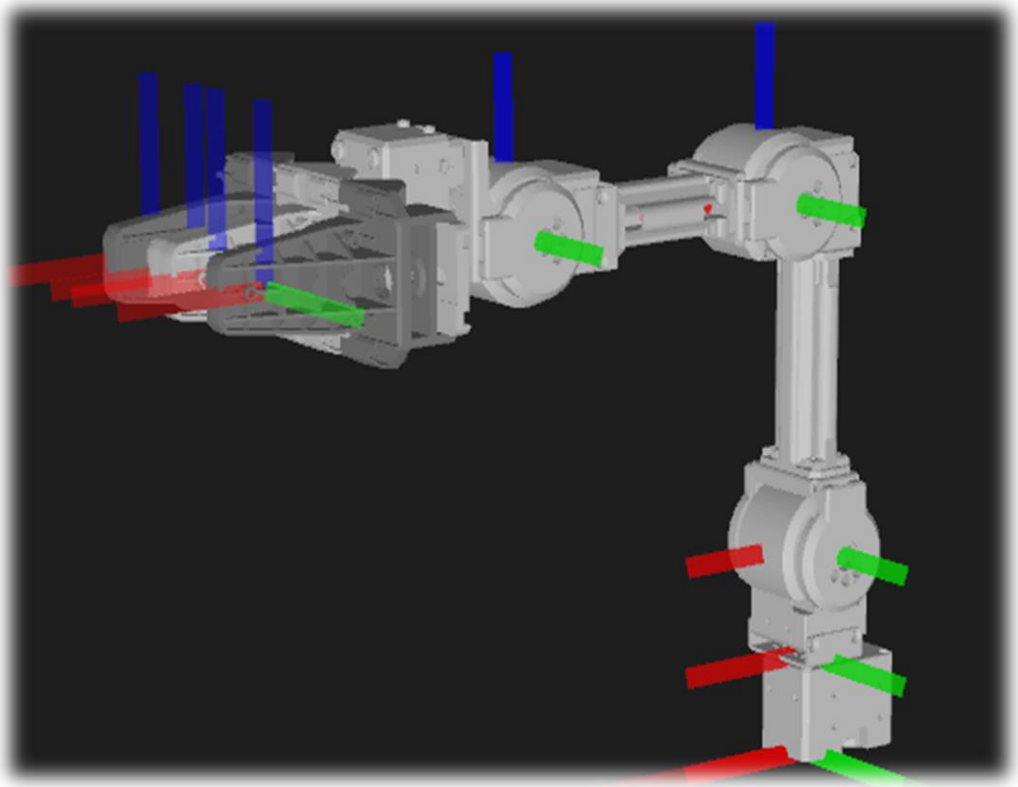
Step 4: Launch Controller using ROS

Important!!!

Before you launch the controller codes, manually lift the arm to a "L" position first!!!

With the end effector (i.e. the gripper) facing the front

(refer to right image)



Step 4: Launch Controller using ROS

Important!!!

Before you close the terminal with the launched controller codes OR before you switch off the power, manually support the arm first!!! And slowly put it down.

If not, the arm motors will be switch off, and it will fall and may injure someone or damage the robot/sensors

Step 4: Launch Controller using ROS

```
SUMMARY
========

PARAMETERS
 * /open_manipulator/control_period: 0.01
 * /open_manipulator/moveit_sample_duration: 0.05
 * /open_manipulator/planning_group_name: arm
 * /open_manipulator/using_moveit: False
 * /open_manipulator/using_platform: True
 * /rosdistro: kinetic
 * /rosversion: 1.12.14

NODES
  /
    open_manipulator (open_manipulator_controller/open_manipulator_controller)

ROS_MASTER_URI=http://localhost:11311

process[open_manipulator-1]: started with pid [23452]
Joint Dynamixel ID : 11, Model Name : XM430-W350
Joint Dynamixel ID : 12, Model Name : XM430-W350
Joint Dynamixel ID : 13, Model Name : XM430-W350
Joint Dynamixel ID : 14, Model Name : XM430-W350
Gripper Dynamixel ID : 15, Model Name :XM430-W350
[ INFO] [1544509070.096942788]: Succeeded to init /open_manipulator
```

*****Ensure the ARM is switched on first**

Type (in New Terminal):

- `$ roscore`
- `$ roslaunch open_manipulator_controller open_manipulator_controller.launch`
- If launched successfully, the terminal will appear as follows (left)
- You are ready to control your robotic arm

Step 4: Launch Controller using ROS

Note that if you can't launch your controller successfully, double check if the USB port in the launch file is defined correctly

- From Home, search and open `open_manipulator_controller.launch`
- Ensure that the following line is as such: `<arg name="dynamixel_usb_port" default="/dev/ttyACM0"/>`

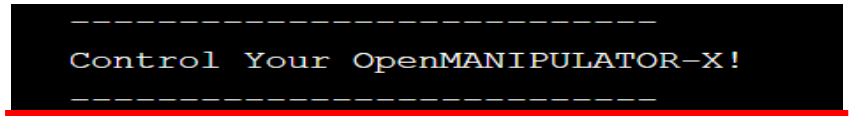
ROS BASICS (for OM)

We will open new terminal and try the following (one by one):

```
1.$ rostopic list
2.$ rosservice list
3.$ rostopic echo /joint_states -n1
4.$ rqt
```

Step 5: Control OM using ROS Software

(a) Manual Control



For physical OM (ROS Noetic), use Hotkeys instead:

g: gripper open

f: gripper close

1: initial pose

2: home pose

Do NOT use other keys!

***Ensure the ARM is switched on and the Controller is activated first**

Type (in New Terminal):

- `$ roslaunch open_manipulator_teleop open_manipulator_teleop_keyboard.launch`
- If the node is successfully launched, the following instruction will appear in the terminal window (left)
- Use your keyboard keys to control the arm; the arm will respond accordingly
- Close terminal when done

Step 5: Control OM using ROS Software

(b) Control via Python (rospy)

Important!!!

Before you close the terminal with the launched controller codes OR before you switch off the power, manually support the arm first!!! And slowly put it down.

If not, the arm motors will be switch off, and it will fall and may injure someone or damage the robot/sensors

Step 5: Control OM using ROS Software

(b) Control via Python (rospy)

Published Topic List : A list of topics that the open_manipulator_controller publishes.

- `/open_manipulator/states`
- `/open_manipulator/joint_states`
- `/open_manipulator/gripper/kinematics_pose`
- `/open_manipulator/*joint_name*_position/command`
- `/open_manipulator/rviz/moveit/update_start_state`

NOTE: These topics are messages for checking the status of the robot regardless of the robot's motion.

`/open_manipulator/joint_states` (sensor_msgs/JointState) is a message indicating the states of joints of OpenMANIPULATOR-X. **"name"** indicates joint component names. **"effort"** shows currents of the joint DYNAMIXEL. **"position"** and **"velocity"** indicates angles and angular velocities of joints.

`/open_manipulator/gripper/kinematics_pose` (open_manipulator_msgs/KinematicsPose) is a message indicating pose (position and orientation) in task space. **"position"** indicates the x, y and z values of the center of the end-effector (tool). **"Orientation"** indicates the direction of the end-effector (tool) as quaternion.

`/open_manipulator/states` (open_manipulator_msgs/OpenManipulatorState) is a message indicating the status of OpenMANIPULATOR. **"open_manipulator_actuator_state"** indicates whether actuators (DYNAMIXEL) are enabled ("ACTUATOR_ENABLE") or disabled ("ACTUATOR_DISABLE"). **"open_manipulator_moving_state"** indicates whether OpenMANIPULATOR-X is moving along the trajectory ("IS_MOVING") or stopped ("STOPPED").

`/open_manipulator/*joint_name*_position/command` (std_msgs/Float64) are the messages to publish goal position of each joint to gazebo simulation node. `*joint_name*` shows the name of each joint. The messages will only be published if you run the controller package with the `use_platform` parameter set to `false`.

`/rviz/moveit/update_start_state` (std_msgs/Empty) is a message to update start state of moveit! trajectory. This message will only be published if you run the controller package with the `use_moveit` parameter set to `true`.

Step 5: Control OM using ROS Software

(b) Control via Python (rospy)

NOTE: These services are messages to operate OpenMANIPULATOR-X or to change the status of DYNAMIXEL of OpenMANIPULATOR.

Service Server List : A list of service servers that open_manipulator_controller has.

- `/open_manipulator/goal_joint_space_path` (open_manipulator_msgs/SetJointPosition)
  The user can use this service to create a trajectory in the joint space. The user inputs the angle of the target joint and the total time of the trajectory.
- `/open_manipulator/goal_joint_space_path_to_kinematics_pose` (open_manipulator_msgs/SetKinematicsPose)
  The user can use this service to create a trajectory in the joint space. The user inputs the kinematics pose of the OpenMANIPULATOR-X end-effector(tool) in the task space and the total time of the trajectory.
- `/open_manipulator/goal_joint_space_path_to_kinematics_position` (open_manipulator_msgs/SetKinematicsPose)
  The user can use this service to create a trajectory in the joint space. The user inputs the kinematics pose(position only) of the OpenMANIPULATOR-X end-effector(tool) in the task space and the total time of the trajectory.
- `/open_manipulator/goal_joint_space_path_to_kinematics_orientation` (open_manipulator_msgs/SetKinematicsPose)
  The user can use this service to create a trajectory in the joint space. The user inputs the kinematics pose(orientation only) of the OpenMANIPULATOR-X end-effector(tool) in the task space and the total time of the trajectory.
- `/open_manipulator/goal_task_space_path` (open_manipulator_msgs/SetKinematicsPose)
  The user can use this service to create a trajectory in the task space. The user inputs the kinematics pose of the OpenMANIPULATOR-X end-effector(tool) in the task space and the total time of the trajectory.

Step 5: Control OM using ROS Software

(b) Control via Python (rospy)

NOTE: These services are messages to operate OpenMANIPULATOR-X or to change the status of DYNAMIXEL of OpenMANIPULATOR.

Service Server List : A list of service servers that open_manipulator_controller has.

- `/open_manipulator/goal_task_space_path_position_only` (open_manipulator_msgs/SetKinematicsPose)
  The user can use this service to create a trajectory in the task space. The user inputs the kinematics pose(position only) of the OpenMANIPULATOR-X end-effector(tool) in the task space and the total time of the trajectory.
- `/open_manipulator/goal_task_space_path_orientation_only` (open_manipulator_msgs/SetKinematicsPose)
  The user can use this service to create a trajectory in the task space. The user inputs the kinematics pose(orientation only) of the OpenMANIPULATOR-X end-effector(tool) in the task space and the total time of the trajectory.
- `/open_manipulator/goal_joint_space_path_from_present` (open_manipulator_msgs/SetJointPosition)
  The user can use this service to create a trajectory from present joint angle in the joint space. The user inputs the angle of the target joint to be changed and the total time of the trajectory.
- `/open_manipulator/goal_task_space_path_from_present` (open_manipulator_msgs/SetKinematicsPose)
  The user can use this service to create a trajectory from present kinematics pose in the task space. The user inputs the kinematics pose to be changed of the OpenMANIPULATOR-X end-effector(tool) in the task space and the total time of the trajectory.
- `/open_manipulator/goal_task_space_path_from_present_position_only` (open_manipulator_msgs/SetKinematicsPose)
  The user can use this service to create a trajectory from present kinematics pose in the task space. The user inputs the kinematics pose(position only) of the OpenMANIPULATOR-X end-effector(tool) in the task space and the total time of the trajectory.
- `/open_manipulator/goal_task_space_path_from_present_orientation_only` (open_manipulator_msgs/SetKinematicsPose)
  The user can use this service to create a trajectory from present kinematics pose in the task space. The user inputs the kinematics pose(orientation only) of the OpenMANIPULATOR-X end-effector(tool) in the task space and the total time of the trajectory.
- `/open_manipulator/goal_tool_control` (open_manipulator_msgs/SetJointPosition)
  The user can use this service to move the tool of OpenMANIPULATOR.
- `/open_manipulator/set_actuator_state` (open_manipulator_msgs/SetActuatorState)
  The user can use this service to control the state of actucators.
  If the user set true at set_actuator_state valuable, the actuator will be enabled.
  If the user set false at set_actuator_state valuable, the actuator will be disabled.
- `/open_manipulator/goal_drawing_trajectory` (open_manipulator_msgs/SetDrawingTrajectory)
  The user can use this service to create a drawing trajectory. The user can create the circle, the rhombus, the heart, and the straight line trajectory.

Step 5: Control OM using ROS Software

(b) Control via Python (rospy)

- **Ensure the ARM is switched on and the Controller is activated first**
- For this exercise, we will be using the python file: `control_om.py`
- First ensure your python files are in the *robotics* package folder in the *src* folder; you may create a folder named "*scripts*" to put all python files in it
- Open terminal and type:

```
$ rosrun robotics control_om.py
```

- The OM will move to a point determined by the joint angles. To alter these joint angles, open the python file, find the following line and change the angles in it; these angles are in radians:

```
joint_position.position =  [-0.5, 0, 0.5, -0.5]
```

- To alter the gripper angles, alter the following line (-0.01 for fully close and 0.01 for fully open):

```
gripper_position.position =  [0.01]
```

Step 5: Control OM using ROS Software

(b) Control via Python (rospy)

Note that if you cant execute the python file, you have to give execution permissions to it by typing (change directory first):

- `$ chmod +x name_of_the_file.py`

Alternatively, a faster way to do this is to follow the steps below:

1. Select all your python files and Right Click
2. Click “Properties” and go to “Permissions” tab
3. Click the box beside “Allow executing file as program”; this box should have a tick inside

Step 5: Control OM using ROS Software

(b) Control via Python (rospy)

Note if you cant execute the python file, you have to give execution permissions to it by typing:

```
$ chmod +x name_of_the_file.py
```

control_om.py

```
#!/usr/bin/env python
# works for actual OM ONLY!
# does not work for actual OM_with_TB3

import rospy                                        #import the python library for ROS
from open_manipulator_msgs.msg import JointPosition     #import JointPosition message from the open_manipulato
from open_manipulator_msgs.srv import SetJointPosition
from sensor_msgs.msg import JointState
import math
import time

def callback(msg):                                  #define a function called 'callback' that receives a parameter
        print msg.name
        print msg.position

def talker():
        rospy.init_node('OM_publisher') #Initiate a Node ca
        set_joint_position = rospy.ServiceProxy('/open_manipulator/goal_joint_space_path', SetJointPosition)
        set_gripper_position = rospy.ServiceProxy('/open_manipulator/goal_tool_control', SetJointPosition)

        while not rospy.is_shutdown():
                joint_position = JointPosition()
                joint_position.joint_name = ['joint1','joint2','joint3','joint4']
                joint_position.position =  [-0.5, 0, 0.5, -0.5]          # in radians
                resp1 = set_joint_position('planning_group',joint_position, 3)
                gripper_position = JointPosition()
                gripper_position.joint_name = ['gripper']
                gripper_position.position =  [0.01]     # -0.01 for fully close and 0.01 for fully open
                respg2 = set_gripper_position('planning_group',gripper_position, 3)

                sub_joint_state = rospy.Subscriber('/open_manipulator/joint_states', JointState, callback)

if __name__== '__main__':
        try:
                talker()
        except rospy.ROSInterruptException:
                pass
```

Service Server

Subscriber

Try Out: Pick up and Place the item

In your respective groups, **run a given ROS node [i.e. `Project2.py`] that can control the OM Robotic Arm (that is placed on the Turtlebot3 base) to successfully pick up a given item from a fixed ground location and place it on another ground location.**

Try Out: Pick up and Place the box

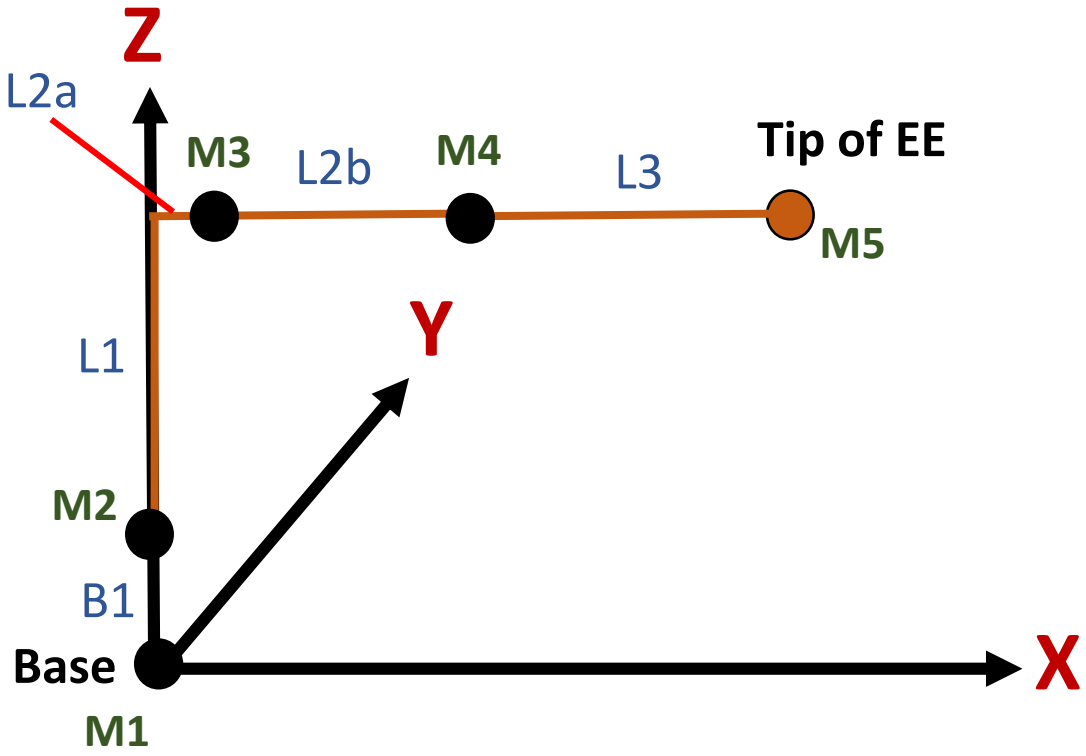


*Figure is not drawn to scale

Physical Specifications of robotic arm and other details

L1	12.8 cm
L2a	2.4 cm
L2b	12.4 cm
L3	12.6 cm
B1	7.7 cm
dist	30.12 cm
Initial Position (IP) of item (x-y-z)	[0, 30.12, 5]
Final Position (FP) of item (x-y-z)	[0, -30.12, 5]

Try Out: Pick up and Place the box

To control the OM,

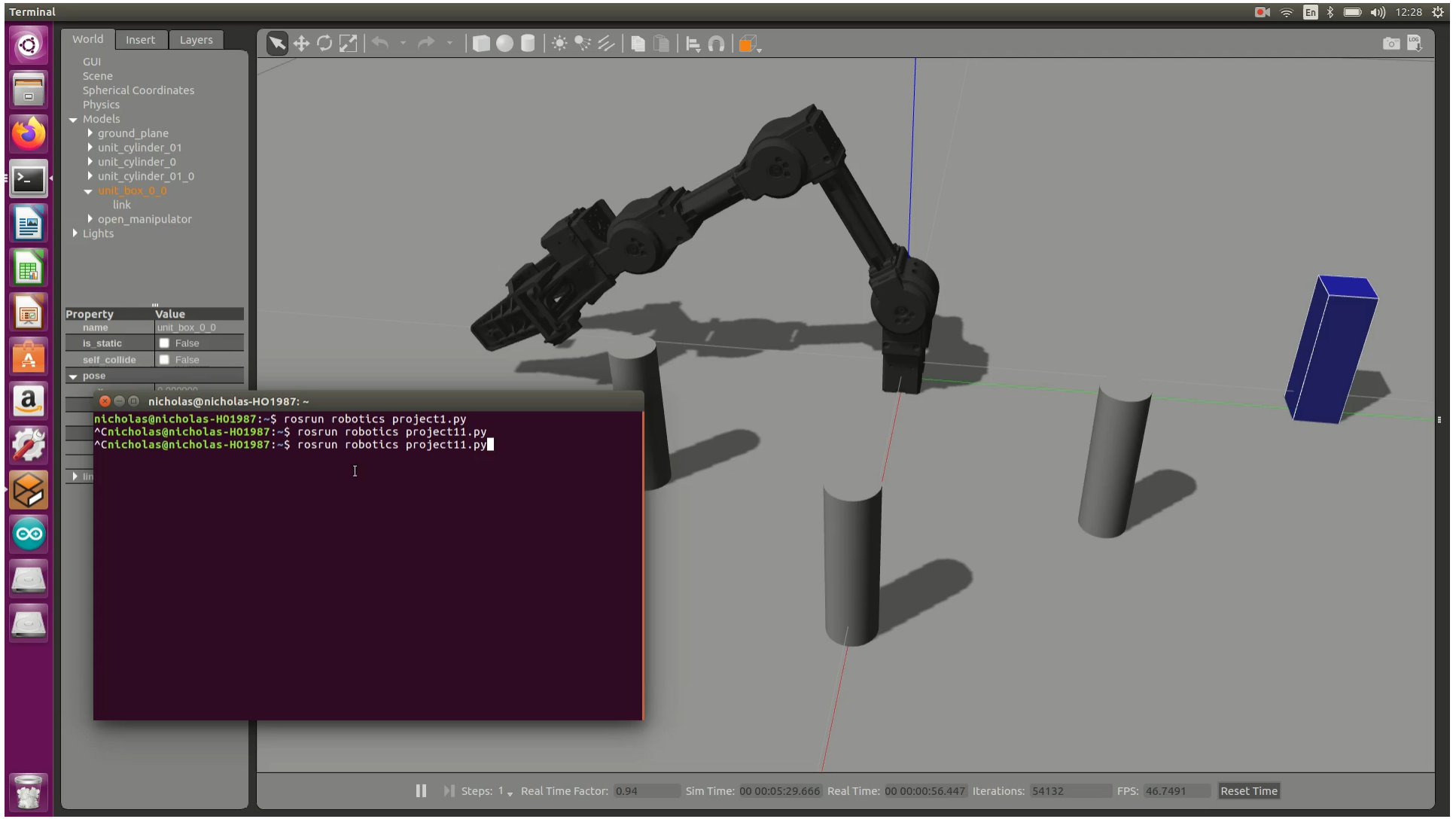
1. Run `$ roscore` on one terminal
2. Prepare the setup for the OM: Connect the wires, switch on the power, and manually bring up the OM
3. On another terminal, activate the controller; this will lock the motors

```
$ roslaunch open_manipulator_controller
open_manipulator_controller.launch
```

4. ROSRUN the given code (found in the autonomous package)

```
$ rosrun autonomous Project2.py
```

Try Out (in Simulation)



Thank you!

Questions?