Lab Sheet Mobile Robot Chapter 2

Robot Operating Systems

ROS (Robot Operating System) is an open-source framework that provides a collection of software libraries and tools to help developers build and control robotic systems. It offers a flexible and distributed architecture for creating modular and reusable robotic applications. ROS facilitates communication between different components of a robot, such as sensors, actuators, and algorithms, through a publish-subscribe messaging system. It enables developers to write robot-specific code in various programming languages and provides a wide range of libraries and packages for tasks like perception, navigation, and manipulation. With its extensive community support and resources, ROS has become a popular platform for research, development, and deployment of robots in various domains.

Objective

The objective of this lab is to provide students with a comprehensive introduction to ROS (Robot Operating System). Students will learn the basics of ROS, its key concepts, and how to work with ROS packages, nodes, and topics. By the end of this lab, students should be able to understand the fundamental concepts of ROS and be prepared to build and run their own ROS applications.

Brief

Importance and applications of ROS in robotics and automation

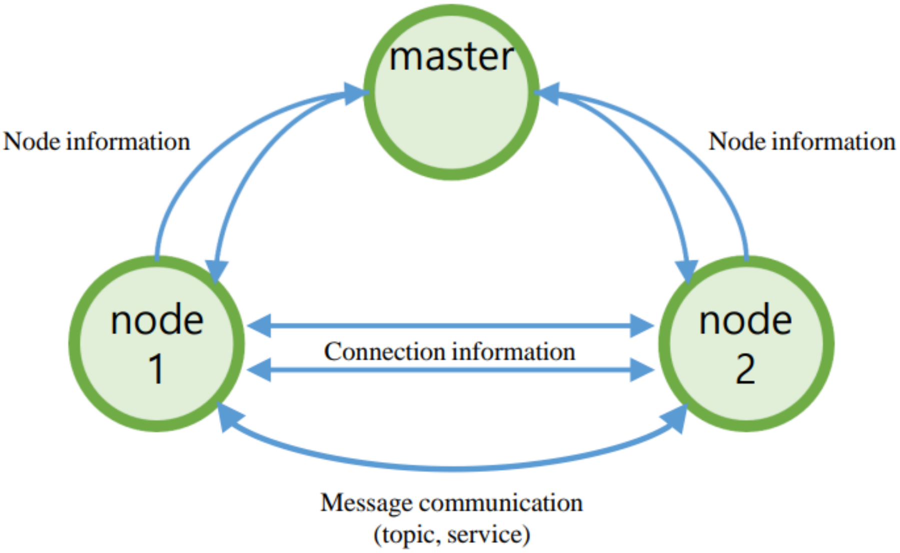
1. A standardized framework that promotes code *reuse*, *collaboration*, and interoperability among different robotic systems.
2. It simplifies the development process by providing a set of tools, *libraries*, and pre-built packages for common robotic tasks.
3. ROS fosters a modular and distributed architecture, allowing for *scalability* and flexibility in building complex robotic systems.
4. ROS has a large and active community, providing support, documentation, and a repository of resources for learning and problem-solving.

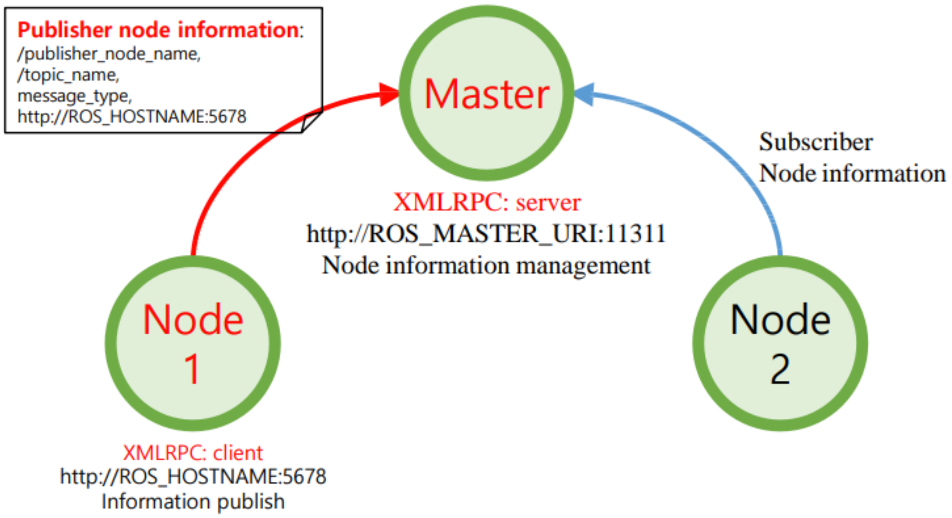
ROS Nodes: Understanding the Concept of Nodes in ROS

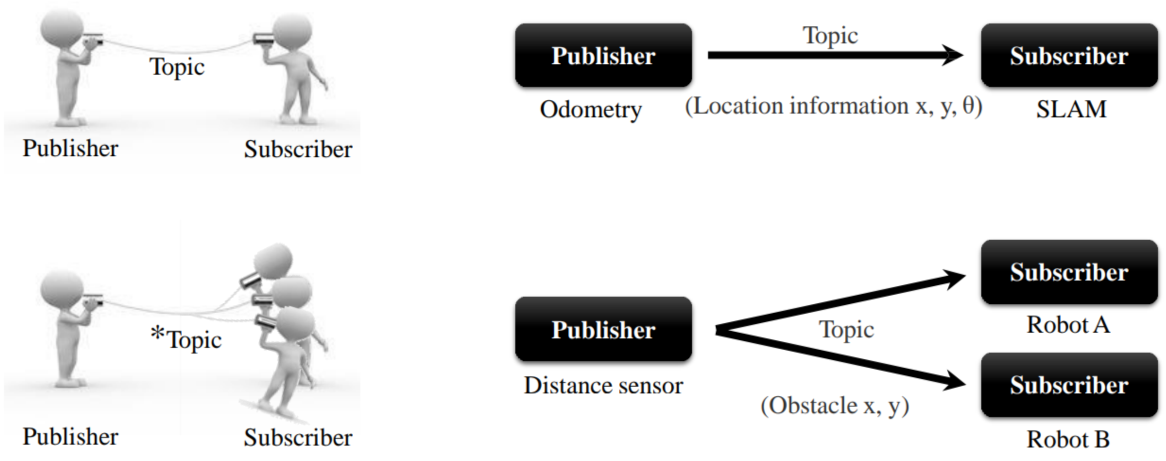
A ROS node can be thought of as a self-contained computational entity that performs a specific task within a robotic system. It represents a process running on a computer or embedded device, executing a piece of code to carry out a particular functionality

Node Communication in ROS

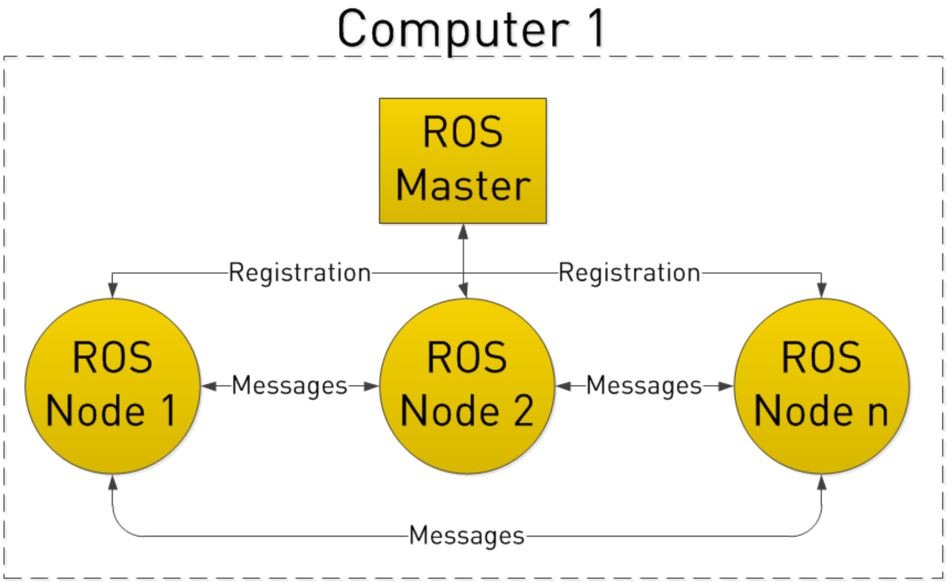
Communication between nodes in ROS is achieved through the use of topics. A topic is a named bus to which nodes can publish messages or subscribe to receive messages. It follows a publish-subscribe pattern, where publishers and subscribers are decoupled from each other. When a node publishes a message on a topic, any other node that has subscribed to that topic receives the message and can process it accordingly.







Example of node communication in ROS using a scenario of a robot performing object detection and localization



Nodes

1. **Camera Node**: The camera node captures video frames from a camera sensor attached to the robot. It publishes the image frames on a topic called **/camera/image**. The camera node continuously captures and publishes these frames at a specific rate.
2. **Object Detection Node**: Another node in the system is responsible for object detection. It subscribes to the **/camera/image** topic to receive the image frames published by the camera node. Using computer vision algorithms, this node analyzes the received frames, detects objects, and publishes the detected object information on a new topic called **/object_detection**.
3. **Path Planning Node**: In a separate node, the path planning algorithm plans the robot's path based on the detected objects. It subscribes to the **/object_detection** topic to receive the object information published by the object detection node. Based on this information, the path planning algorithm calculates an optimal path to navigate around the detected objects and publishes the computed path on a topic called **/path**.

Lab 1: ROS Node Communications

The objective of this lab is to introduce students to the concept of node communication in ROS using the TurtleSim simulator. Students will gain an understanding of how nodes can exchange information and coordinate their actions within a robotic system. By the end of this lab, students should be able to create and interact with multiple nodes, publish and subscribe to topics, and visualize the communication using TurtleSim.

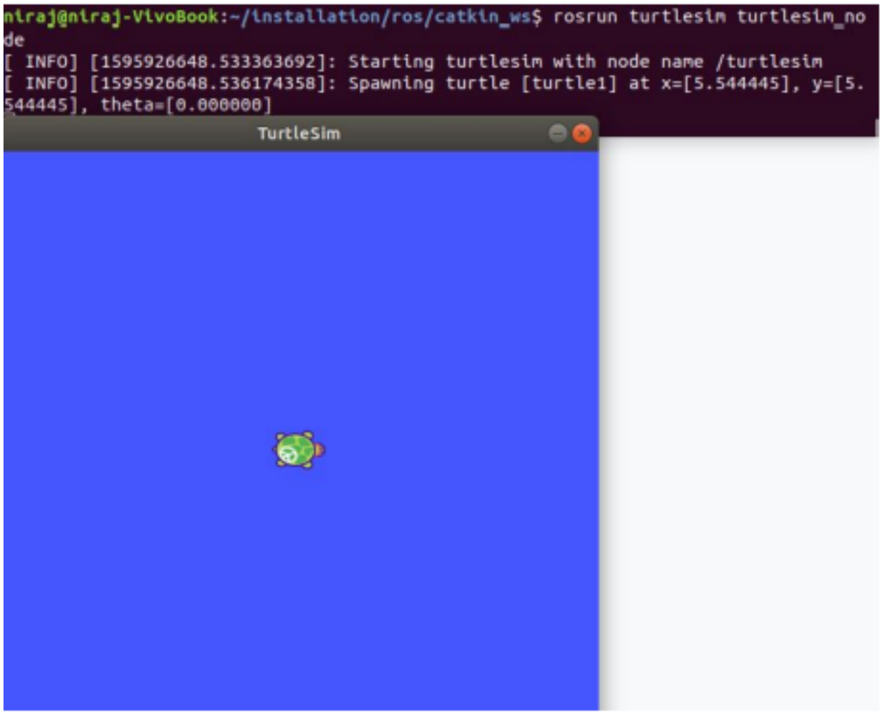
Lab 1 Practice on understanding ROS Node

1. Open a terminal and run the following command to launch ROS Master:

```
roscore
```

2. Open a new terminal and run the following command to launch the TurtleSim simulator:

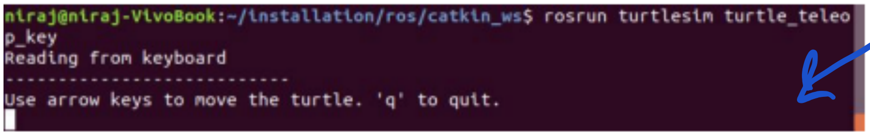
```
rosrun turtlesim turtlesim_node
```



```
niraj@niraj-VivoBook:~/installation/ros/catkin_ws$ rosrun turtlesim turtlesim_no
de
[ INFO] [1595926648.533363692]: Starting turtlesim with node name /turtlesim
[ INFO] [1595926648.536174358]: Spawning turtle [turtle1] at x=[5.544445], y=[5.
544445], theta=[0.000000]
```

3. Open a new terminal and run the following command to launch the virtual joystick (teleopkey):

```
rosrun turtlesim turtle_teleop_key
```



```
niraj@niraj-VivoBook:~/installation/ros/catkin_ws$ rosrun turtlesim turtle_teleo
p_key
Reading from keyboard
----------------------------
Use arrow keys to move the turtle. 'q' to quit.
```

Click ↑ ← → ↓

4. Open a new terminal and run the following command to start the a graphical tool in ROS that provides a visual representation of the nodes and their communication relationships:

```
rosrun rqt_graph rqt_graph
```

5. Open a new terminal and run the following command to start the command rostopic to list the currently active topic:

```
rostopic list
```

6. Run the following command to list the active nodes:

```
rosnode list
```

7. Run the following command to see the message type of topic /turtle1/cmd_vel:

```
rostopic info /turtle1/cmd_vel
```

   The output should display geometry_msgs/Twist which is the message type.

8. Run the following command to see the variable structure of message type:

```
rosmsg show geometry_msgs/Twist
```

   The output should display the variable name of message type.

9. Run the following command to see current value of the topic:

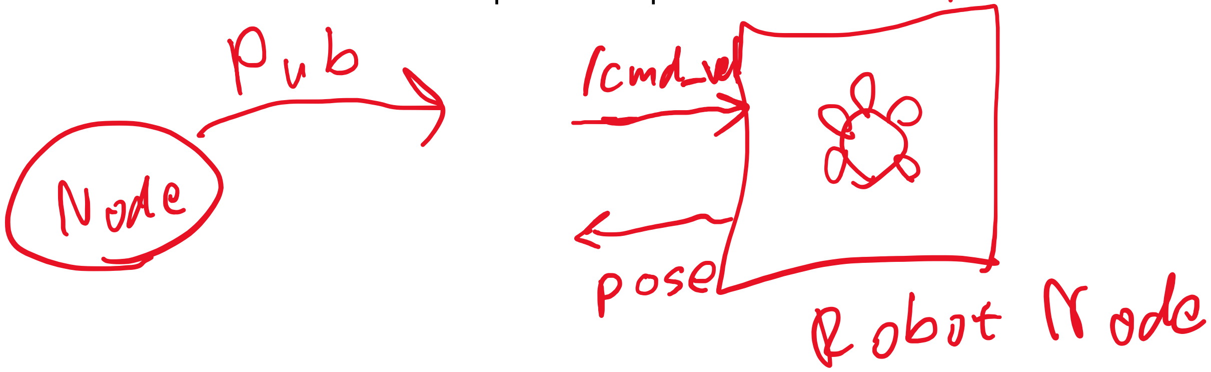
```
rostopic echo /turtle1/pose
```

   The output should display the current value of the topic, so this command can be stopped by ctrl+c.

Lab 1 Assignments

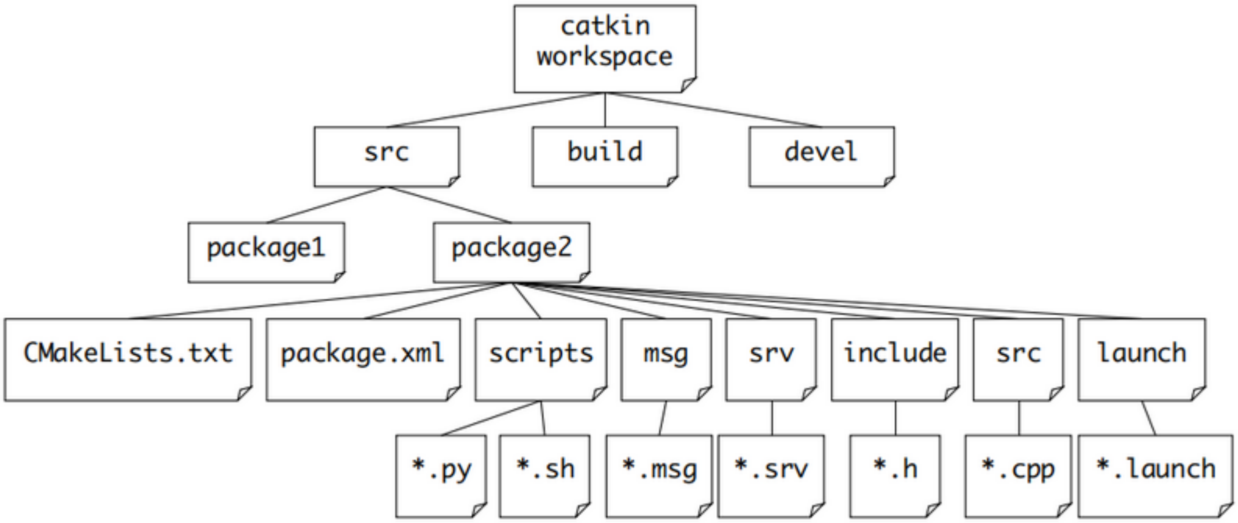
Please answer the following questions (By typing on the paper and upload to Classroom

1. Which nodes are displayed in the rqt_graph tool?
2. Which nodes are publishing on specific topics, and which nodes are subscribing to them?
3. What happens to the node graph when the virtual joystick (teleopkey) node is stopped?
4. What is the message type of topic /turtle1/pose? → rosinfo
5. What is the variable structure of topic /turtle1/pose? → rosmsg



Lab 2: Make your package

You should have gained a basic understanding of the ROS workspace and package concepts. From the diagram below, the root of ROS workspace is the catkin_ws. This folder will contain all of the robot packge.



ROS Workspace

A ROS workspace is a directory where you organize and develop your ROS packages. It serves as the main working directory for ROS development. The workspace provides a structured environment for creating, building, and managing multiple packages for your robotic projects.

The structure of a typical ROS workspace consists of three main folders:

1. **src:** The "src" folder (short for source) is where you place your ROS ***packages***. Each package resides in its own subdirectory within the "src" folder. Packages contain the source code, configuration files, and any additional resources required for a specific functionality or module.
2. **build:** The "build" folder is automatically generated by the build system (e.g., Catkin) during the compilation process. It contains the intermediate build files, object files, and executables that are created when building your ROS packages.
3. **devel:** The "devel" folder is also created by the build system. It serves as the development space where the built executables and libraries are located. The "devel" folder contains the setup files that need to be sourced to use the packages within the workspace.

ROS Package

A ROS package is a fundamental unit of software organization in ROS. It provides a modular and self-contained structure for managing and distributing ROS-related functionality. A package can contain one or more nodes, libraries, configuration files, launch files, and any additional resources needed for a specific purpose or task.

To create a new ROS package and build the workspace, follow these steps:

1. Open a terminal and navigate to src folder under the workspace.
   ```
   cd ~/catkin_ws/src
   ```

2. Run the following command to create a new package:
   ```
   catkin_create_pkg lab2 std_msgs rospy roscpp
   ```

3. Run the following command to compile the workspace:
   To compile the workspace, student should go to workspace directory.
   ```
   cd ~/catkin_ws
   ```

   Then, compile
   ```
   catkin_make
   ```

4. To update the current environment with new package, run the following command.
   ```
   source ~/catkin_ws/devel/setup.bash
   ```

ROS Run

After the package has been created successfully, the python script can be created to control our robot. And, it can be execute with command rosrun.

To create the python script, follow these steps:

1. The python script should be created under the package which keep in folder scripts. The first command is the changing directory to the package lab2 using roscd command.
   ```
   roscd lab2
   ```

2. Create the folder scripts (If it is not existed)
   ```
   mkdir scripts
   ```

3. Go inside the script folder
   ```
   cd scripts
   ```

4. Create a file name pub.py
   ```
   touch pub.py
   ```

5. Change mode (As I told you from 1st chapter)

```
chmod +x pub.py
```

6. You can write your code, but to use Python under the ROS environment, it requires importing rospy and adding a few lines of code. Please modify the code in the #TODO section.

```
#!/usr/bin/python3
import rospy

def main():
    rospy.init_node("lab2")
    rate = rospy.Rate(100)
    # Code to publish

    while not rospy.is_shutdown():
        # TODO
        rate.sleep()
if __name__ == '__main__':
    try:
        main()
    except rospy.ROSInterruptException:
        pass
```

7. The sample code to publish data into topic /turtle1/cmd_vel.

```
#!/usr/bin/python3
import rospy

from geometry_msgs.msg import Twist
vel = Twist()


def main():
    rospy.init_node("lab2")
    rate = rospy.Rate(100)

    # Code to publish
    pub = rospy.Publisher('/turtle1/cmd_vel', Twist,queue_size=10)

    while not rospy.is_shutdown():
        # TODO
        vel.linear.x = 1
        pub.publish(vel)
        rate.sleep()

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

8. The sample code to subscribe data from topic /turtle1/pose.

```
#!/usr/bin/python3
import rospy

from geometry_msgs.msg import Twist
from turtlesim.msg import Pose
vel = Twist()
pos = Pose()

def callback(data):
    """Active when recevie info from topic"""
    # using global keyword to
    # edit variable outside function
    global pos
    pos = data


def main():
    rospy.init_node("lab2")  → Init node
    rate = rospy.Rate(100)   → Freq

    # Code to publish
    pub = rospy.Publisher('/turtle1/cmd_vel', Twist,queue_size=10)  ✓
    sub = rospy.Subscriber('/turtle1/pose', Pose, callback)
    while not rospy.is_shutdown():
        # TODO
        print("Turtle is running at", pos.x, pos.y)   Current position
        vel.linear.x = 1   1 m/s  0 m/s  -1 m/s
        pub.publish(vel)
        rate.sleep()

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

Add code

Lab 2 Assignments

Assignment 1: Student must complete the create a new package process that require student to do the screenshot to prove the completion and submit.

Assignment 2: Students must complete the new Python script (Using source code from *step no. 8*) that requires self-evaluation by executing the rosrun lab2 pub.py If the execution has no error, students do the screenshot and submit.

Python

Assignment 3: Turtle needs to be stopped at position x = 10. Students modify the source code by creating the condition (if-else) to complete this task and upload the screenshot of completion.
Hint: The variable pos.x which represents the TurtleSim position can be used to create the condition to make it stop by changing the velocity command by vel.linear.x = 0.