UNITED STATES MILITARY ACADEMY

LAB 2 REPORT - CONTROL OF THE TURTLESIM USING ROS

EE487: EMBEDDED SYSTEMS DEVELOPMENT
SECTION T2

COL KORPELA

BY

CDT NICHOLAS E. LIEBERS '23, C-2

WEST POINT, NY 10996

16 FEB 2022

LIEBERS.NICHOLAS.EDWARD.1557580448 Digitally signed by LIEBERS.NICHOLAS.EDWARD.1557580448 Date: 2022.02.16 03:24:24 -05'00' DOCUMENTATION IDENTIFIES ALL SOURCES USED AND ASSISTANCE RECEIVED IN COMPLETING THIS ASSIGNMENT.

____ NO SOURCES WERE USED OR ASSISTANCE RECEIVED IN COMPLETING THIS ASSIGNMENT.

SIGNATURE: LIEBERS.NICHOLAS.EDWARD.1557580448 Digitally signed by LIEBERS.NICHOLAS.EDWARD.1557580448 Date: 2022.02.16 03:24:40 -05'00'

Contents

1. Introduction

1.1. Purpose

The purpose of this lab is to introduce cadets to Robot Operating System (ROS) and use ROS to control a virtual robot (a turtle). Cadets must use this new knowledge of ROS to control/maneuver a virtual robot to draw the first letter in one's name. This purpose was accomplished, and the following learning objectives were met:

1.2. Learning Objectives

1. Configure nodes to exchange information via topics within the publisher/subscriber framework of ROS.

2. Write custom publisher/subscriber nodes in ROS to listen to a topic and perform computations when new information is available on that topic.

3. Learn how to interact and control a small, mobile robot such as the Turtlebot in a simulated environment (TurtleSim).

4. Leverage pre-existing hardware drivers and other nodes in the ROS repository and integrate them together with custom-written nodes as part of a robot application.

2. Theory

In order to understand how ROS functions, we must define what ROS Nodes, Topics, Messages, Publishers, and Subscribers. Additionally, we must explain how these different pieces of the ROS framework integrate into one another.

2.1. Why Use ROS?

ROS offers researchers the ability to use a test-proven framework (ROS) to allow their robot's different components to communicate with one another. The advantage of ROS is that it takes the focus off getting devices to communicate, and instead put the focus on programming the robot's tasks.

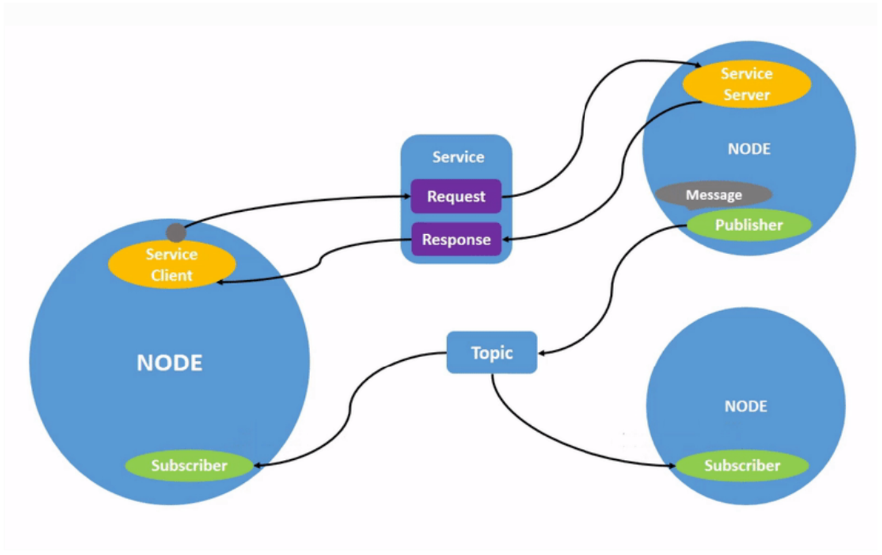


Figure 1: A simple diagram that outlines how Nodes, Topics, Messages, Publishers, and Subscribers relate to one another. The Publishers publish a message over a Topic, and the message is received/read by a subscriber [8].

2.2. ROS Nodes

A ROS Node processes, sends, and receives information. Usually, Nodes are created to perform one single task. For example, in a robot, one node would control the wheels, one node would control the robot's arm movement, and another node would control the robot's head movement. In summary, a node really should only be performing one single task, and this node will send and receive information related to performing its task [4]. As seen in Figure 1, the Nodes run publishing and subscribing services.

In Python, to initialize a Node in Python, the following lines are used:

```python
import rospy
rospy.init_node('some name', anonymous=True)
```

2.3. ROS Topics

A ROS Topic is a named bus or medium that messages can be sent through. Publishers publish to a Topic and a Subscriber subscribes to a topic. A topic offers a Publisher and Subscriber a shared "standard" for messages. For example, if a Node is subscribed to a location Topic, then that Node can expect to receive x and y coordinate data. Topics ensure that the data being received by a node matches what is expected, and this feature ensures that data can be processed correctly [7]. In Figure 1, notice how the Publishers and Subscribers are linked together by a common topic. This allows those Nodes to communicate over the assigned Topic.

2.4. ROS Messages

A ROS Message is just the message that holds the information being transmitted. Message format is dependent on what type of Topic the Message is being sent over. For example, a position Topic will support Messages that contain x and y coordinates. A Message that contains anything other than the Topic's defined Message format cannot be sent over that Topic. Notice in Figure 1, the Messages are being published by the Publisher, sent over a Topic, and then received by a Subscriber.

2.5. ROS Publisher

A ROS Node is able to publish to topics, and if a Node is publishing to a Topic, that Node is considered a Publisher. Publishing to a Topic requires a Node to send Messages over the specified Topic [5]. Notice in Figure 1 that each node can contain a Publisher that is sending Messages to Subscribers over a specified Topic. If a Publisher and Subscriber are not connected by the same Topic, the data stream will not be able to send data.

To initialize a Publisher in Python, use the following code:

```python
import rospy
rospy.init_node('some name', anonymous=True)
velocity_publisher = rospy.Publisher(topic, message_type, queue_size)
```

[1]

2.6. ROS Subscriber

A ROS Subscriber just listens for Messages on a Topic and receives them as the Messages come in. Any data or Message that is Published on a Topic will be heard/received by a Subscriber if and only if that Subscriber is Subscribed to the Topic and the Message is of the correct data type that the Subscriber is expecting [5]. Notice in Figure 1 that a Subscriber is connected to a Publisher that shares the same Topic. Also notice that there can be more than one Subscriber to a single Topic.

To initialize a Subscriber in Python, use the following code:

```python
import rospy
rospy.init_node('some name', anonymous=True)
self.pose_subscriber = rospy.Subscriber(topic, message_type,
    function_to_run_when_receive_update)
```

2.7. ROS Launch Files

When starting up a ROS session, it can be exceptionally helpful to use a ROS Launch File that will start up all of the ROS Nodes, Publishers, Subscribers, and other required services. Rather than using multiple terminals to accomplish this task, a Launch File does it all using one command [9].

The following is an example Launch File used in this lab:

```xml
<launch>
<node pkg="lab2liebers" type="lab1.py" name="turtlebot_controller"/>
<node pkg="lab2liebers" type="key.py" name="talker"/>
<node pkg="turtlesim" type="turtlesim_node" name="turtlesim"/>
</launch>
```

[9]

3. Experimental Setup and Procedure

3.1. Required Equipment

1. Computer running Linux.
2. ROS package running on Linux Machine.
3. Python

3.2. Software Setup

Since this Lab occurs solely on a computer, the physical setup only requires a person to have a computer running Linux or a VM running Linux.

3.2.1. ROS Setup

Using the roscreate command in a ROS workspace, a ROS package must be built[1]:

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

[2]

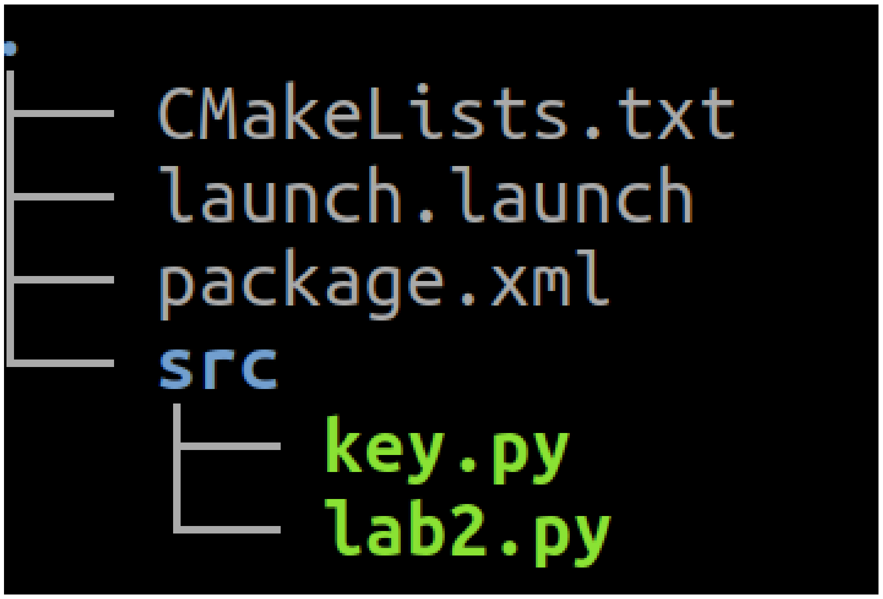
In the context of this Lab, the package was created using the following command:

```
# catkin_create_pkg lab2liebers turtlesim_cleaner geomtry_msgs rospy std_msgs
# catkin_make
```

[3]

[1]See http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment and http://wiki.ros.org/ROS/Tutorials/CreatingPackage to see a more detail explanation of how to set up the ROS environment.

After building the project, the sub directories are created such that the following file tree is created:



```
.
├── CMakeLists.txt
├── launch.launch
├── package.xml
└── src
    ├── key.py
    └── lab2.py
```

Figure 2: The directory and file structure for this project.

Running

```
# roslaunch lab2liebers launch.launch
```

will launch the all of the nodes and the Turtle simulation. The launch.launch file is located in Appendix A.

3.2.2. Node, Publisher, and Subscriber Setup

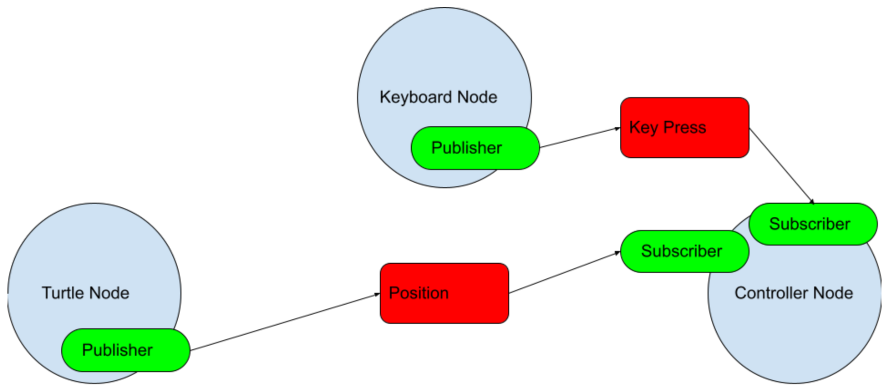


Figure 3: This diagram outlines this ROS project's structure of Nodes, Topics, Publishers, and Subscribers communicating. Blue circles are Nodes, the red boxes are Topics, and the green boxes are Publishers and Subscribers.

In Figure 3, the program's setup is outlined in a graphical format. The three different nodes are from their own Python files, and in order to launch them, the launch.launch file must start each one individually. Additionally, notice that some nodes have more than one subscriber. In Appendix B, a detailed implementation of our code in Python outlines how the code is setup and arranged.

3.2.3. Code Structure

The code is functioning in the following way:

1. Launch turtle and Nodes.
2. When the letter ”P” is pressed on the keyboard, send a message to the Controller Node.
3. When the Controller Node received the message that ”P” was pressed, draw the letter ”N” with the turtle.

In Figure 2, the ”key.py” contains the Publisher that sends information about the letter ”P” being pressed. The file ”lab2.py” contains all of the code to control the turtle and subscribe to all of the necessary topics.

3.2.4. Testing Procedures

In order to test our program’s functionality, a combination of trail & error, ”print line” debugging, and code tracing were used to test our code. In order to catch bugs, we only edited small parts of the code and tested those parts respectively.

4. Results and Discussion

The turtle was able to draw my the letter ”N” using a combination of Publishers and Subscribers. Since this lab is in code, these results are repeatable since there is no randomness associated with the code.

4.0.1. Errors and Issues

The biggest issue that was encountered in this project was working with the ROS projects and making sure the Linux environment was set up correctly to build the ROS project. Since ROS handles lots of the overhead processing required to send information between different nodes, if the environment is not set up correctly, errors will arise that are tricky to debug.

4.0.2. Solutions to Errors

Usually, to fix issues with the ROS projects, the solution was to delete the ROS folder and reinstall the project. Additionally, referencing the ROS documentation/tutorials usually showed another way to achieve implement the Python code.

4.0.3. Incremental Software Building Process

This project consists of three parts:

1. The Turtle.

2. The Turtle Controller.

3. The Keyboard-Press Publisher.

We built the Turtle Controller first. Using a tutorial from the ROS Wiki, we were able to quickly get the turtle to move around the screen [3]. This tutorial used a class-based implementation in Python, and this way allowed us to stay organized while creating functions. We edited the code to fit our implementation while also continuing to use parts that would help with our task. We went from making the turtle move in a straight line, to rotating, to moving and rotating, and finally to creating the letter ”N.”

After the turtle was working properly, we had to implement the a Publisher to publish whenever the letter "P" was pressed. When the letter "P" was pressed, the turtle would begin drawing. This task was easy since we already knew how to create Publishers and Subscribers.

By breaking the project up into two different Python files–and three parts–it allowed us to debug as we went without getting too overwhelmed.

5. Conclusion and Recommendations

In conclusion, this lab was a success. The purpose was accomplished: a virtual turtle was programmed using ROS to draw the letter ”N.” The most important takeaway from this lab is that ROS projects must be accomplished in the following order: build the ROS framework/test that ROS is running properly, and then spend time coding the implementation. During the lab, I started building the implementation without testing that ROS was functioning properly. This issue created troubles and made it difficult to test. Once the ROS project is working properly, it make engineering solutions simple because ROS is not that complicated, but it has a difficult learning curve. In the future, referencing this lab and its code will help to get the project moving faster.

6. References

[1] Code listing - Overleaf, Online LaTeX Editor.

[2] CreatingPackage - ROS Wiki.

[3] Go to Goal - ROS Wiki.

[4] Nodes - ROS Wiki.

[5] Publishers and Subscriber - ROS Wiki.

[6] Rotating Left and Right - ROS Wiki.

[7] Topics - ROS Wiki.

[8] Understanding ROS 2 Nodes, 2022.

[9] Jason M O'kane. A Gentle Introduction to ROS.

Appendix A - Launch File

0.1. launch.launch file

```xml
<launch>
<node pkg="lab2liebers" type="lab1.py" name="turtlebot_controller"/>
<node pkg="lab2liebers" type="key.py" name="talker"/>
<node pkg="turtlesim" type="turtlesim_node" name="turtlesim"/>
</launch>
```

Appendix B - Python Code

0.2. Turtle Controller

```python
import rospy
from std_msgs.msg import String
from geometry_msgs.msg import Twist
from turtlesim.msg import Pose
from math import pow, atan2, sqrt
import math

class KeyBoardSubscriber:
    def __init__(self):
  self.start = False
  # rospy.init_node('listener', anonymous=True)
  self.listenKey = rospy.Subscriber("chatter", String, self.callback)

    def callback(self, data):
  data = str(data)
  print(data)
  if data == "data: \"P\"":
    print(str(self.start))
    self.start = True
class TurtleBot:
    def __init__(self):
        # Creates a node with name 'turtlebot_controller' and make sure it is a
        # unique node (using anonymous=True).
        rospy.init_node('turtlebot_controller', anonymous=True)

        # A subscriber to the topic '/turtle1/pose'. self.update_pose is called
        # when a message of type Pose is received.
        self.pose_subscriber = rospy.Subscriber('/turtle1/pose',
                                                Pose, self.update_pose)

        self.pose = Pose()
        self.rate = rospy.Rate(10)

  self.vel_msg = Twist()

    def update_pose(self, data):
        self.pose = data
        self.pose.x = round(self.pose.x, 4)
        self.pose.y = round(self.pose.y, 4)


    def rotate(self, angle):
      # Setting the current time for distance calculus
      print("X location: {}, Y location: {}".format(self.pose.x, self.pose.y))
      self.vel_msg.angular.z = .25
      t0 = rospy.Time.now().to_sec()
      current_angle = 0
      relative_angle = angle*2*math.pi/360

      while(current_angle < relative_angle):
          self.velocity_publisher.publish(self.vel_msg)
          t1 = rospy.Time.now().to_sec()
          current_angle = self.vel_msg.angular.z*(t1-t0)

      self.vel_msg.angular.z = 0
      self.velocity_publisher.publish(self.vel_msg)

    def move(self, distance):
        # See Lab references for where this code implementation comes from
      print("X location: {}, Y location: {}".format(self.pose.x, self.pose.y))
      self.vel_msg.linear.x = 1.0
      t0 = rospy.Time.now().to_sec()
      current_location = 0
      while current_location < distance:
          self.velocity_publisher.publish(self.vel_msg)
          t1 = rospy.Time.now().to_sec()
          current_location = self.vel_msg.linear.x*(t1-t0)
        self.vel_msg.linear.x = 0
        self.velocity_publisher.publish(self.vel_msg)


if __name__ == '__main__':
    try:
        x = TurtleBot()
      key = KeyBoardSubscriber()
      while not key.start:
          pass
      x.rotate(90)
      x.move(3.0)
      x.rotate(225)
      x.move(3.5)
      x.rotate(135)
      x.move(3.0)
    except rospy.ROSInterruptException:
        pass
```

[3] [6]. Note, this code heavily references the example from the ROS Wiki. Some functions/variables are copied from this tutorial. Since I used the tutorial as a starting point and modified the code to fit my desired requirements, the code structure/layout heavily resembles the tutorial since the code originated from the tutorial.

0.3. Key-pressing Node

```
import rospy
from std_msgs.msg import String

def talker():
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(10)
    while True:
        string = raw_input("\n")
  if string == "P":
      # Publish that the string is now a P
      rospy.loginfo(string)
      pub.publish(string)
      break

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

[3]