[ELE00118M] MSc Practical Robotics (PRAR)

[ELE00103M] Year 4 Robotics module

Lab Session 7 Handout

Lab 7: Robot Control with ROS

Lecturer: Mark A Post (mark.post@york.ac.uk)

Technician: Mike Angus

1. Aims and Objectives

In this lab session you will be experimenting with Robot Operating System (ROS) robotics middleware and learning how to write networked publishers and subscribers in the Python language within the ROS programming methodology. ROS is not the only robotics middleware out there, but it is the most widely developed and used in the open-source community, and will be of great use in future projects.

2. Learning outcomes

- An understanding of the ROS architecture and methodology for system design
- The ability to start, use, and analyze the components and messages within ROS
- Experience programming ROS packages using the Python language and Catkin

3. Software and hardware

You'll be using the Raspberry Pi with Linux and ROS installed. You can also use a Lab PC if you are in the lab, but it must be booted into Linux – if it is currently running Windows, reboot it now and choose Linux from the menu when it starts up. It is best to use an Ubuntu Linux host PC – either with a natively running Linux OS or Ubuntu running on a Virtualbox VM, so that you can use "`ssh -X`" to connect to the Raspberry Pi and have applications open a window on your host PC. Note that if you wish you can install ROS to Mac OSX and potentially to Windows using Windows Subsystem for Linux but these have not been tested for your labs so we suggest Pi-only work-arounds.

4. Pre-Lab Preparation

- Review the material from class on ROS and other middlewares and their components and operation

5. Tasks

TASK 1: GETTING STARTED

First of all, you need to set up the command-line environment for ROS if it has not been set up already. Open up a terminal window and type

```
source /opt/ros/noetic/setup.bash
```

This will add the ROS tools to your PATH environment variable and allow you to use the various ROS commands. To make this persistent in your user home directory (if it is not there already – check by typing "`cat .bashrc`"), you can add the line to your .bashrc file with the following line (make sure two ">>" characters are used!)

```
echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc
```

Now, in your terminal window type

```
rosdep update
```

You will only have to do this once, because the results of rosdep update are cached in your home directory, which is stored on the network. Note that you need an Internet connection to do this - Now type

```
roscore &
```

to start the core ROS service and the /rosout node. You will leave this program running for the whole of the lab session without intervention, so if you are using a terminal window feel free to minimise it. If you are using the serial terminal or ssh, the "&" afterwards tells it to run in the background so you can do other things in the same terminal. you will need to type "`fg`" to bring it back to the foreground, after which you can terminate it with *Ctrl-C*.

For the remainder of the lab session you will be opening a large number of terminal windows. This is because each ROS node runs as a separate program, and each one will be launched from a separate terminal window. You may need to arrange the windows carefully on the screen so that you can see everything you need to see.

TASK 2: RUNNING THE EXAMPLE SYSTEM

In this section you will be using an example system provided in the 'tutorials' package of ROS. It's a simulation of a turtle, a wheeled robot with a pen that can draw lines when

it moves around. You'll use it to see an example of topics, publishers and subscribers, and to investigate some of the tools that allow running ROS systems to be investigated.

The turtle simulator consists of one node that runs the simulation itself, and another that listens for key presses and publishes data on a topic that provides commands to the turtle. The simulator node is subscribed to this topic, and performs the requested actions.

Turtlesim usually opens a window to show the turtle graphically. However, if you do not have your Raspberry Pi hooked up to a monitor, are not using a Linux OS, and are not running Linux on Virtualbox on your host PC, you will not be able to see a window as you need a compatible X window system running such as Xorg. The work-around for this is to export the following environment variable into your shell:

```
export QT_QPA_PLATFORM='offscreen'
```

To run the example system:

1. Open a terminal window to the Pi with SSH and type

```
rosrun turtlesim turtlesim_node
```

2. Open another new terminal window (or use the serial port console) and type

```
rosrun turtlesim turtle_teleop_key
```

to start the /turtlesim node. If you have an X server running this also opens a new window, showing the turtle in the middle of a solid background.

3. Make sure you can see the simulator window – move some windows around if you need to. Now click on the terminal window running the */turtle_teleop_key* node so that it has the focus, meaning that keystrokes typed on the keyboard are sent to this window. Press the arrow keys on the keyboard and watch the turtle move around. If you do not see the window, run the following line and see the turtle coordinates changing:

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

TASK 3: EXAMINING THE SYSTEM

Now the turtle simulator is running, you can use a variety of tools to investigate its structure and inspect the messages passing from node to node. Open new terminal windows or SSH terminals as you need to see more.

1. First, open another terminal window and try

```
rosnode list
```

to see a list of the nodes in the system. You should see the /turtlesim and /teleop_turtle nodes that you have created, plus the /rosout logging node that is created when you start roscore.

2. For more detailed information on any given node, try e.g.

```
rosnode info /teleop_turtle
```

which will list the topics on which the node publishes, the topics to which it is subscribed, and some other information about the node.

3. Now try

```
rostopic list
```

to see a list of all the topics that are present in the system. You should see the /rosout topic, as well as one called /rosout_agg that is used by the /rosout node (see wiki.ros.org/rosout for more information), and three topics within the /turtle1 namespace.

4. It is also possible to eavesdrop on a topic to see what data is being published. Try typing

```
rostopic echo /turtle1/cmd_vel
```

and then making the turtle move using the /turtle_teleop node while watching the output. Observe how the commanded turtle velocity from the /teleop_turtle node changes as you press the arrow keys, and how the /turtlesim node responds.

5.Note how the /rosout node and channel are missing, as well as any of the topics that don’t have subscribers. To show the /rosout node and topic, uncheck the box marked ‘Debug’. You should see the /rosout node and topic appear, along with a node called /rqt_gui_py_node_xxxxx

where xxxxx is a randomly assigned number. This last node is the node created by rqt_graph for the purposes of obtaining system information.

7.To show the topics that have no subscribers, first use the drop-down box at the top left to switch the view from ‘Nodes only’ to ‘Nodes/Topics (all)’. The display will change a little, so each topic is presented exactly once (as a rectangle) and the arrows join nodes to topics and topics to nodes. Now uncheck the ‘Dead sinks’ and ‘Leaf topics’ boxes, and see how the remaining topics appear.

8.One of the unsubscribed topics is /turtle1/color_sensor. Try echoing this to see what data it produces, by opening a new terminal window and typing

```
rostopic echo /turtle1/color_sensor
```

before moving the turtle around. See how the output of the /turtle1/color_sensor topic changes as the turtle crosses lines it's previously drawn.

TASK 4: CREATING A PACKAGE

In this section you will create your own package containing two nodes. One node will publish data on a topic, and the other node will subscribe to the topic to receive the data. Both of these nodes will be written in the Python language.

The topic will be called /data and will carry strings as messages. The nodes will be called /sender and /receiver. The /sender node will publish strings to the topic, and /receiver will receive them via a subscription.

This example is loosely based on one from the ROS wiki at wiki.ros.org. Before you start, cancel (Ctrl-C) all the processes in the open terminal windows, except roscore. Close all of the unused terminal windows.

To create a new package, we will use the catkin tool. The steps to use this are the same every time you want to create a new package, so you might find these instructions useful during your group project too.

1. To create and initialise a workspace for catkin:

    a. Open a terminal window, and type

    ```
    cd
    ```

    to make sure you are in your home directory.

    b. Create the necessary directories using

    ```
    mkdir -p catkin_ws/src
    ```

    ```
    cd catkin_ws
    ```

    c. Initialise the workspace by typing

    ```
    catkin_make
    ```

2. To initialize the package code, create a bare package for your work:

a. Change to the catkin source directory, by typing

```
cd src
```

b. Type

```
catkin_create_pkg lab_exercises std_msgs rospy
```

to create a package called ‘lab_exercises’ with the ‘std_msgs’ and ‘rospy’ packages as dependencies. ‘std_msgs’ is the package that provides the standard ROS message types for topics, and ‘rospy’ provides support for writing packages using Python.

Customise your package metadata. Note that after the last step catkin responded with a message inviting you to “adjust the values in package.xml”. We’re going to do that now.

a. Type

```
nano lab_exercises/package.xml
```

to edit the file.

b. Change the maintainer email address to your own.

c. Notice how you can change the package description and the licence (useful if you’re posting code online).

d. Scroll down to the bottom of the file and find the lines starting with <buildtool_depend>. These indicate that the ‘catkin’ and ‘rospy’ packages are required when building this and running this package.

e. Change back to the catkin_ws directory with “`cd ..`” and rebuild the workspace, including your new package, with

```
catkin_make
```

The package has now been created, but it doesn’t do anything yet because we haven’t written any actual code.

TASK 5: EXAMINING THE PACKAGE

Before commands like rosrun are able to find our new package, we need to ‘set up’ the package. This involves running a script, and you will need to do this every time you open a new terminal window because the variables it creates exist only within a session.

To set up any packages in the catkin workspace, first make sure you're in the home directory (type cd to be sure) and then type

```
source catkin_ws/devel/setup.bash
```

TASK 6: WRITING SOME CODE

Now that the new package has been set up, you can use the roscd command to change to the package directory without having to remember exactly where it is:

```
roscd lab_exercises
```

You will need to create a directory inside the package to hold your python scripts, so type

```
mkdir scripts
```

```
cd scripts
```

to create and enter the directory.

TASK 7: THE SENDER

To create the sender, type

```
nano sender.py
```

and then add the following code. There's quite a bit of typing to do, but if I gave you the code as a file you'd probably never read it! At least this way you have to read it as you type it out! The first line looks strange but is important: it ensures that the Python interpreter is used to run this file.

```
#!/usr/bin/env python

# Imports
import rospy
import std_msgs.msg

# Sender function
def sender():

      # Declare a topic called /data with String type
      pub = rospy.Publisher('data', std_msgs.msg.String, queue_size=10)

      # Initialise the node
```

```
      rospy.init_node('sender')

      # Loop once per second and publish a message to the topic
      rate = rospy.Rate(1)
      while not rospy.is_shutdown():
            message = "The time is %s" % rospy.get_time()
            rospy.loginfo("Sending: " + message)
            pub.publish(message)
            rate.sleep()

# Calls the sender() function when the file is run
if __name__ == '__main__':
      try:
            sender()
      except rospy.ROSInterruptException:
            pass
```

TASK 8: THE RECEIVER

For the receiver, edit receiver.py and add this code:

```
#!/usr/bin/env python

# Imports
import rospy
from std_msgs.msg import String

# This function will be called whenever data is received
def callback(data):
      rospy.loginfo("Received: " + data.data)

# Receiver function
def receiver():
      # Initialise the node
      rospy.init_node('receiver')
      # Subscribe to the /data topic
      rospy.Subscriber('data', String, callback)
      # Repeatedly wait for messages
      rospy.spin()

# Calls the receiver() function when the file is run
if __name__ == '__main__':
      try:
            receiver()
      except rospy.ROSInterruptException:
            pass
```

TASK 9: RUNNING THE CODE

To make the code executable, type

```
chmod +x *.py
```

which makes all the python files in this directory executable.

Now you should be able to type

```
rosrun lab_exercises sender.py
```

to start the sender.

Start the receiver by opening a new terminal window, typing

```
source catkin_ws/devel/setup.bash
```

to set up your package, and then typing

```
rosrun lab_exercises receiver.py
```

TASK 10: NAME CLASHES

Try starting a second copy of the sender or receiver. What happens?

ROS will not allow two nodes to exist with the same name. If you wanted to retain control over the nodenames but allow multiple copies of the same node, you could change your python code to allow the node name to be passed in as an argument. If you're not bothered, then there's an 'anonymous' option to rospy.init_node() that can be used, for example

```
rospy.init_node('sender', anonymous=True)
```

which will add a random number to the end of the node name to make it unique. Try editing sender.py to make the node anonymous, and then creating two senders and one receiver. Does it work? What does rosnode_list show for this new system?

If you decide to use ROS in your final group project, you will find you need to learn more than this script has taught you. For this, two important resources are:

- The ROS wiki at wiki.ros.org
- 'A Gentle Introduction to ROS' by Jason M. O'Kane, which has code examples only in C++ but is a good introduction to several important ROS concepts.

Combined with the wiki above, you should find it easy enough to convert the examples to Python.