Summer School in Robotics

ROS Network and BrickPi3 Robots

Andreas Persson, Pedro Zuidberg Dos Martires,
David Caceres Dominguez
andreas.persson@oru.se, pedro.zuidberg-dos-martires@oru.se,
david.caceres-dominguez@oru.se

June 15, 2022

1 ROS Network Configuration

By default, the ROS core system will be launched with the *ROS Master* running in the `localhost`, meaning that the *ROS network* will only run locally on each device. However, configuring the ROS network to run across multiple devices is, in fact, very easy. All you need to do is to change the *ROS Master environment variable* in the Terminal, which is done through the `export` command. For this second tutorial, we have already prepared the Raspberry Pi with the label *rp12* to be the *ROS Master*. To change your specific *RPi* device to use the same *ROS Master*, use the following `export` command:

```
$ export ROS_MASTER_URI=http://192.168.1.112:11311
```

This command specifies that the `ROS_MASTER_URI` will be RPi with hostname `rp12`, communicating through the standard ROS network port `11311`. It should be noted, however, that changes in environmental variables are specific to each Terminal, so you need to run the above command in each additional Terminal that you want to use for communication with other devices across the ROS network.

Running a ROS network across multiple devices also circumvents the need to run the `roscore` on every single device. The `roscore` program will instead be running solely on the *ROS Master device*, coordinating the communication for the whole ROS network (already running on *rp12*). If you encounter connectivity problems, this normally means that the `roscore` and the *ROS Master device* can not determine the IP address or hostname of your specific RPi device. This problem can most often be fixed by changing the environment variables `ROS_IP` and `ROS_HOSTNAME`, and explicitly specifying the device's IP address and hostname, e.g., for RPi labeled *rp1*:

```
$ export ROS_IP=192.168.1.101
```

1.1 Running ROS nodes across multiple devices

Through a few minor changes, we can use the same `talker.py` and `listener.py` nodes for communication across multiple devices. Since a `topic` is identified by its name, there can not be two topics with the same name within a ROS network. It is, therefore, convenient to use a *hierarchical naming structure* for topics in a ROS network. This can be done by including the device name in the topic name. For example, for Raspberry Pi labeled *rp1*, changing the name of the `topic` named `chatter` to `/rp1/chatter`.

Follow the instructions from yesterday's tutorial for how to remotely access the VSCode through code-serve, open the source file for the `talker.py` node (sould be found within the folder `~/catkin_ws/src/tutorial/script`), and change to a hierarchical naming structure for the publishing `topic` (named on line 13). For example, for RPi labeled *rp1*:

```
12 # Create a publisher that will publish messages on topic named 'chatter'
13 pub = rospy.Publisher('/rp1/chatter', String, queue_size=10)
```

Also, feel free to change the text message published by the `talker.py` node to something more personal (keep it friendly, though):

```
# Create and publish a String message
str = "hello world %s" % rospy.get_time()
```

Save the source file changes, move to a Terminal, *source* the workspace's `setup.bash` file, and use the `rosrun` command to run the `talker.py` node of the `tutorial` package:

```
$ cd ~/catkin_ws
$ source ./devel/setup.bash
$ rosrun tutorial talker.py
```

Notice, we didn't start the ROS core system by the `roscore` *command.* Instead, we use the ROS core system of the *ROS Master device*. In another Terminal, check which topics that are available in the running ROS system by invoking the command:

```
$ rostopic list
```

Choose a topic from the list, open the source file for the `listener.py` node (within the folder `~/catkin_ws/src/tutorial/script`), and change the name for the subscribing `topic` accordingly (named on line 18):

```
12 # Initialize a subscriber that will receive and handle message
13 # though a given callback function
14 rospy.Subscriber("chatter", String, callback)
```

Save the source file changes, *source* the workspace's `setup.bash` file, and `rosrun` the `listener.py` node to see what text message your fellow summer school colleague has actually sent on the chosen topic.

2 BrickPi3 Robots

For this summer school, we have upgraded our old *LEGO Mindstorms EV3* robots and replaced each standard "EV3 Brick" with a combination of a BrickPi3 connected to a Raspberry Pi 4. Through the connectors of the BrickPi3 board, we can directly interface LEGO EV3 (and NXT) sensors and motors with the Raspberry Pi. Like the traditional EV3 Brick, the BrickPi3 board has four connectors for sensors (labeled `S1`-`S4`) and four for motors (labeled `MA`-`MD`). For this tutorial, we will begin by connecting a LEGO EV3 Touch sensor to BrickPi3 connector labeled `S1`. To get started with reading and publishing sensor values through a ROS node, we have prepared a small example:

```python
#!/usr/bin/env python
import rospy                        # Import the ROS Python library
from std_msgs.msg import Bool       # Import Bool message type from standard messages
import brickpi3                     # Import the BrickPi3 drivers

'''A class for handling sensor(s).'''
class Sensor:
    def __init__(self): # Class constructor

        # Create a publisher
        self.pub = rospy.Publisher('/touch/reading', Bool, queue_size=10)

        # Create BrickPi3 instance
        self.BP = brickpi3.BrickPi3()

        # Configure for a touch sensor on connector S1
        self.BP.set_sensor_type(self.BP.PORT_1, self.BP.SENSOR_TYPE.TOUCH)

    # Method for reading and publishing sensor values
    def read(self):
        try:
            value = self.BP.get_sensor(self.BP.PORT_1)
            self.pub.publish(value)
        except brickpi3.SensorError:
            pass

    # Method for "unconfigure" all sensors and motors
    def reset(self):
        self.BP.reset_all()

# Main function
if __name__ == '__main__':

    # Init the connection with the ROS system
    rospy.init_node('sensor', anonymous=True)

    # Create Sensor instance
    s = Sensor()
    try:
```

```python

        # Start the ROS main loop, running with a frequency of 10Hz
        rate = rospy.Rate(10)
        while not rospy.is_shutdown():
            s.read()     # Call sensor read method
            rate.sleep() # Call sleep to maintain the desired rate

    except rospy.ROSInterruptException:
        s.reset() # Call sensor reset method
```

This code example can also be found as file `sensor.py` under directory `tutorials` of the *web page* for the summer school: http://github.com/pedrozudo/astxoru-roboticssummerschool

Create a new package, called `robot`, in the same `catkin_ws` workspace. This package should have the same dependencies as the `tutorial` package, and should include a `scripts` folder for Python source files:

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg robot rospy std_msgs
$ cd ~/catkin_ws/src/robot
$ mkdir scripts
$ cd scripts
```

Add the `sensor.py` source file to the `scripts` folder of the newly created `robot` package, and make sure that the source file is directly executable:

```
$ chmod +x sensor.py
```

Edit the file `CMakeLists.txt` file found in the root folder of the `robot` package (i.e, `~/catkin_ws/src/robot`), and add the following lines:

```
catkin_install_python(PROGRAMS scripts/sensor.py
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

Last but not least, compile the package, *source* the workspace's `setup.bash` file, and `rosrun` the `sensor.py` node:

```
$ cd ~/catkin_ws
$ catkin_make
$ source ./devel/setup.bash
$ rosrun robot sensor.py
```

Through the use of another Terminal, you should now be able to see the sensor readings from the EV3 Touch Sensor being published on the ROS network by command:

```
$ rostopic echo /touch/reading
```

From hereon, take inspiration from the many examples found in the DexterInd/BrickPi3 GitHub repository. *For example, try to replace the Touch Sensor with a Color Sensor (and update the source file accordingly), add another sensor, or write a source file for receiving commands and controlling a motor (it's only the number of connectors that's your limitation!).*