Purdue AMP

Getting Started

Essential links -

- Get to know about us better at https://engineering.purdue.edu/AMP/
- Learn ROS from https://www.robotigniteacademy.com/en/course/ros-in-5-days/details/
- Refer to ROS documentation at - http://wiki.ros.org/ROS/Tutorials
- Obtain the code you will be working with by cloning/downloading our GitHub repo from https://github.com/kstrubel/AMP

ROS

Learning ROS, if you aren't already familiar with it, will probably be your biggest challenge during the onboarding process. But don't be alarmed, we're here to get you started and will walk you through exactly what you need to go to get you up to speed. Following is a compilation of the most important parts of the official ROS tutorial that are essential to AMP –

1. **Installing and configuring ROS**
   Install the Kinetic version of ROS from here - http://wiki.ros.org/ROS/Installation
   If you are ever having problems finding or using your ROS packages make sure that you have your environment properly setup. A good way to check is to ensure that environment variables like ROS_ROOT and ROS_PACKAGE_PATH are set:

   ```
   $ printenv | grep ROS
   ```

   If they are not then you might need to 'source' some setup.*sh files. If you installed ROS Kinetic, that would be:

   ```
   $ source /opt/ros/kinetic/setup.bash
   ```

   Now, we create and build a catkin workspace:

   ```
   $ mkdir -p ~/catkin_ws/src
   $ cd ~/catkin_ws/
   $ catkin_make
   ```

2. **Navigating ROS**
   To navigate ROS, we will inspect the ros-tutorials package, which can be installed using:

```
$ sudo apt-get install ros-<distro>-ros-tutorials
```

Some Filesystem concepts are:

- Packages: Packages are the software organization unit of ROS code. Each package can contain libraries, executables, scripts, or other artifacts.
- Manifests (package.xml): A manifest is a description of a package. It serves to define dependencies between packages and to capture meta information about the package like version, maintainer, license, etc...

Rospack - allows you to get information about packages. Usage:

```
$ rospack find [package_name]
```

Roscd – allows you to change directory (similar to cd in bash). Usage:

```
$ roscd [locationname[/subdir]]
```

Rosls - allows you to ls directly in a package by name. Usage:

```
$ rosls [locationname[/subdir]]
```

3. **Creating ROS packages**

The simplest possible package might have a structure which looks like this:

```
• my_package/
•   CMakeLists.txt
•   package.xml
```

First change to the source space directory of the catkin workspace you created in the Creating a Workspace for catkin tutorial:

```
# You should have created this in the Creating a Workspace Tutorial
$ cd ~/catkin_ws/src
```

Now use the catkin_create_pkg script to create a new package called 'beginner_tutorials' which depends on std_msgs, roscpp, and rospy:

```
$ catkin_create_pkg beginner_tutorials std_msgs rospy roscpp
```

Now you need to build the packages in the catkin workspace:

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

To add the workspace to your ROS environment you need to source the generated setup file:

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

4. **Building a ROS Package**

You should already have a catkin workspace and a new catkin package called beginner_tutorials from the previous tutorial, Creating a Package. Go into the catkin workspace if you are not already there and look in the src folder:

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

We can now build that package using catkin_make:

```
$ catkin_make
```

You should see a lot of output from cmake and then make

5. **Understanding ROS Nodes**

We will be using a simulator for this tutorial. Install it by running:

```
$ sudo apt-get install ros-kinetic-ros-tutorials
```

Here are some Graph concepts that are essential to using ROS:

- Nodes: A node really isn't much more than an executable file within a ROS package. ROS nodes use a ROS client library to communicate with other nodes. Nodes can publish or subscribe to a Topic. Nodes can also provide or use a Service.
- Client Libraries: ROS client libraries allow nodes written in different programming languages to communicate: rospy (python) and roscpp (c++)
- Roscore: roscore is the first thing you should run when using ROS.Just run:

  ```
  $ roscore
  ```

- Using rosnode: Open up a new terminal to use rosnode and see what running roscore did. But do this without closing the previous terminal. Rosnode displays information about the ROS nodes that are currently running. The rosnode list command lists these active nodes:

  ```
  $ rosnode list
  ```

- The rosnode info command returns information about a specific node. It gives us info about what it published, its subscriptions, and its services.

```
$ rosnode info /[node_name]
```

- Using rosrun: rosrun allows you to use the package name to directly run a node within a package

```
$ rosrun [package_name] [node_name]
```

- Let's run the turtlebot simulator. Open a new terminal and run:

```
$ rosrun turtlesim turtlesim_node
```

- A window should open up with a turtlebot and you should be able to view the turtle. Additionally, if you open a new terminal and run rosnode list, you should be able to see:

```
/rosout
/turtlesim
```

- If you close the turtilesim window and want to run this with a different name, you can reassign the names in the command line by running:

```
$ rosrun turtlesim turtlesim_node __name:=my_turtle
```

- Now, if we run rosnode list again, we would see:

```
/my_turtle
/rosout
```

- To test the /my_turtle node, run ping, which is another rosnode command:

```
$ rosnode ping my_turtle
```

6. **Understanding ROS Topics**

First, make sure that roscore is running by opening up a new terminal and run:

```
$ roscore
```

Next, we run turtlesim in a new terminal

```
$ rosrun turtlesim turtlesim_node
```

To operate the turtle, open a new terminal and run

```
$ rosrun turtlesim turtle_teleop_key
```

Now you can use the arrow keys of the keyboard to drive the turtle around.

Behind the scenes, these nodes are communicating with each other via “topics”. The turtlesim_node and the turtle_teleop_key node are communicating with each other over a ROS Topic. turtle_teleop_key is publishing the key strokes on a topic, while turtlesim subscribes to the same topic to receive the key strokes.

rqt_graph creates a dynamic graph of what's going on in the system. rqt_graph is part of the rqt package. To install it, run –

```
$ sudo apt-get install ros-<distro>-rqt
$ sudo apt-get install ros-<distro>-rqt-common-plugins
```

To run rqt_graph, run the following command in a new terminal –

```
$ rosrun rqt_graph rqt_graph
```

The rostopic tool allows you to get information about ROS topics. You can use the help option to get the available sub-commands for rostopic –

```
$ rostopic -h
```

rostopic echo shows the data published on a topic.

```
rostopic echo [topic]
```

rostopic list returns a list of all topics currently subscribed to and published.
Let's figure out what argument the list sub-command needs. In a new terminal run:

```
$ rostopic list -h
```

Communication on topics happens by sending ROS messages between nodes. For the publisher (turtle_teleop_key) and subscriber (turtlesim_node) to communicate, the publisher and subscriber must send and receive the same type of message. This means that a topic type is defined by the message type published on it. The type of the message sent on a topic can be determined using rostopic type. rostopic type returns the message type of any topic being published –

```
rostopic type [topic]
```

rostopic pub publishes data on to a topic currently advertised –

```
rostopic pub [topic] [msg_type] [args]
```

rostopic hz reports the rate at which data is published –

```
rostopic hz [topic]
```

rqt_plot displays a scrolling time plot of the data published on topics. Here we'll use rqt_plot to plot the data being published on the /turtle1/pose topic. First, start rqt_plot by typing

```
$ rosrun rqt_plot rqt_plot
```

7. Understanding ROS Services and Parameters

Services are another way that nodes can communicate with each other. Services allow nodes to send a request and receive a response.

```
rosservice list         print information about active services
rosservice call         call the service with the provided args
rosservice type         print service type
rosservice find         find services by service type
rosservice uri          print service ROSRPC uri
```

rosparam allows you to store and manipulate data on the ROS Parameter Server. The Parameter Server can store integers, floats, boolean, dictionaries, and lists. rosparam uses the YAML markup language for syntax. In simple cases, YAML looks very natural: 1 is an integer, 1.0 is a float, one is a string, true is a boolean, [1, 2, 3] is a list of integers, and {a: b, c: d} is a dictionary. rosparam has many commands that can be used on parameters –

```
rosparam set            set parameter
rosparam get            get parameter
rosparam load           load parameters from file
rosparam dump           dump parameters to file
rosparam delete         delete parameter
rosparam list           list parameter names
```

8. **Using rqt_console and roslaunch**

To begin this tutorial, install the rqt and the turtlesim packages by running –

```
$ sudo apt-get install ros-kinetic-rqt ros-kinetic-rqt-common-plugins r
os-kinetic-turtlesim
```

rqt_console attaches to ROS's logging framework to display output from nodes. rqt_logger_level allows us to change the verbosity level (DEBUG, WARN, INFO, and ERROR) of nodes as they run. Before we start the turtlesim, in two new terminals start rqt_console and rqt_logger_level –

```
$ rosrun rqt_console rqt_console
```

```
$ rosrun rqt_logger_level rqt_logger_level
```

Two windows will pop up, both of which allow you set the severity level of events. Setting the logger level to “Warn” will display a “Warn” message when our turtle hits the wall. These logger levels differ in severity with “Fatal” being the most severe message and “Debug” being the least severe.

9. **Using rosed to edit files in ROS**

rosed is part of the rosbash suite. It allows you to directly edit a file within a package by using the package name rather than having to type the entire path to the package –

```
$ rosed [package_name] [filename]
```

This opens the file with the vim editor

10. **Creating a ROS msg and srv**

msg: msg files are simple text files that describe the fields of a ROS message. They are used to generate source code for messages in different languages.
srv: a srv file describes a service. It is composed of two parts: a request and a response.

msg files are stored in the msg directory of a package, and srv files are stored in the srv directory.

Let's define a new msg in the package that was created in the previous tutorial –

```
$ roscd beginner_tutorials
$ mkdir msg
$ echo "int64 num" > msg/Num.msg
```

The example .msg file above contains only 1 line.
We need to make sure that the msg files are turned into source code for C++, Python, and other languages. Open package.xml, and make sure these two lines are in it and uncommented –

```
  <build_depend>message_generation</build_depend>
  <exec_depend>message_runtime</exec_depend>
```

Let's make sure that ROS can see it using the rosmsg show command –

```
$ rosmsg show [message type]
```

For example, running "$ rosmsg show beginner_tutorials/Num" should display "int64 num"

Let's use the package we just created to create a srv –

```
$ roscd beginner_tutorials
$ mkdir srv
```

We need to make sure that the srv files are turned into source code for C++, Python, and other languages. Unless you have done so already, open package.xml, and make sure these two lines are in it and uncommented –

```
  <build_depend>message_generation</build_depend>
  <exec_depend>message_runtime</exec_depend>
```

Let's make sure that ROS can see it using the rossrv show command –

```
$ rossrv show <service type>
```

To make sure these work correctly, uncomment the following lines in CMakeLists.txt –

```
# generate_messages(
#   DEPENDENCIES
# #  std_msgs  # Or other packages containing msgs
# )
```

And make sure they look like the following –

```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

Now, make the package again –

```
# In your catkin workspace
$ roscd beginner_tutorials
$ cd ../..
$ catkin_make install
$ cd -
```

11. Writing a Simple Publisher and Subscriber

"Node" is the ROS term for an executable that is connected to the ROS network. Here we'll create the publisher ("talker") node which will continually broadcast a message.
To work on the tutorial, change the directory to the beginner_tutorials package –

```
$ roscd beginner_tutorials
```

First lets create a 'scripts' folder to store our Python scripts in –

```
$ mkdir scripts
$ cd scripts
```

Then download the example script talker.py to your new scripts directory and make it executable –

```
$ wget https://raw.github.com/ros/ros_tutorials/kinetic-devel/rospy_tut
orials/001_talker_listener/talker.py
$ chmod +x talker.py
```

You can view and edit the file with $ rosed beginner_tutorials talker.py

Every Python ROS Node will have this declaration at the top. The first line makes sure your script is executed as a Python script. You need to import rospy if you are writing a ROS Node. The

std_msgs.msg import is so that we can reuse the std_msgs/String message type (a simple string container) for publishing.

pub = rospy.Publisher("chatter", String, queue_size=10) declares that your node is publishing to the chatter topic using the message type String. String here is actually the class std_msgs.msg.String. The queue_size argument is New in ROS hydro and limits the number of queued messages if any subscriber is not receiving them fast enough.

The next line, rospy.init_node(NAME, ...), is very important as it tells rospy the name of your node -- until rospy has this information, it cannot start communicating with the ROS Master. In this case, your node will take on the name talker.

anonymous = True ensures that your node has a unique name by adding random numbers to the end of NAME.

rate = rospy.Rate(10) creates a Rate object rate. With its argument of 10, we should expect to go through the loop 10 times per second.

The while loop is a fairly standard rospy construct: checking the rospy.is_shutdown() flag and then doing work. You have to check is_shutdown() to check if your program should exit (e.g. if there is a Ctrl-C or otherwise). In this case, the "work" is a call to pub.publish(hello_str) that publishes a string to our chatter topic. The loop calls rate.sleep(), which sleeps just long enough to maintain the desired rate through the loop.

To write the subscriber node, download the listener.py file into your scripts directory –

```
$ roscd beginner_tutorials/scripts/

$ wget https://raw.github.com/ros/ros_tutorials/kinetic-devel/rospy_tutorials/001_talker_listener/listener.py
```

Don't forget to make the node executable using the chmod command –

```
$ chmod +x listener.py
```

To build your nodes, we use CMake as our build system. This is to make sure that the autogenerated Python code for messages and services is created. Go to your catkin workspace and run catkin_make –

```
$ cd ~/catkin_ws

$ catkin_make
```

12. Examining the Simple Publisher and Subscriber

Make sure that a roscore is up and running by typing –

```
$ roscore
```

If you are using catkin, make sure you have sourced your workspace's setup.sh file after calling catkin_make but before trying to use your applications –

```
# In your catkin workspace
$ cd ~/catkin_ws
$ source ./devel/setup.bash
```

Run the “talker” publisher –

```
$ rosrun beginner_tutorials talker.py
```

The publisher node is up and running. Now we need a subscriber to receive messages from the publisher.

Now, run the “listener” subscriber –

```
$ rosrun beginner_tutorials listener.py
```

When you are done, press Ctrl-C to terminate both the listener and the talker.

13. Writing a Simple Service and Client

Here we'll create the service ("add_two_ints_server") node which will receive two ints and return the sum. Change directory into the beginner_tutorials package, you created in the earlier tutorial, creating a package –

```
$ roscd beginner_tutorials
```

Create the scripts/add_two_ints_server.py file within the beginner_tutorials package and paste the following inside it –

```
1 #!/usr/bin/env python
2
3 from beginner_tutorials.srv import *
4 import rospy
5
6 def handle_add_two_ints(req):
7     print "Returning [%s + %s = %s]"%(req.a, req.b, (req.a + req.b))
```

```
    return AddTwoIntsResponse(req.a + req.b)

def add_two_ints_server():
    rospy.init_node('add_two_ints_server')
    s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
    print "Ready to add two ints."
    rospy.spin()

if __name__ == "__main__":
    add_two_ints_server()
```

Now, make the script executable –

```
chmod +x scripts/add_two_ints_server.py
```

Now, create the scripts/add_two_ints_client.py file within the beginner_tutorials package and paste the following inside it:

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

import sys
import rospy
from beginner_tutorials.srv import *

def add_two_ints_client(x, y):
    rospy.wait_for_service('add_two_ints')
    try:
        add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
        resp1 = add_two_ints(x, y)
        return resp1.sum
    except rospy.ServiceException, e:
        print "Service call failed: %s"%e

def usage():
    return "%s [x y]"%sys.argv[0]

if __name__ == "__main__":
    if len(sys.argv) == 3:
        x = int(sys.argv[1])
        y = int(sys.argv[2])
    else:
        print usage()
        sys.exit(1)
    print "Requesting %s+%s"%(x, y)
    print "%s + %s = %s"%(x, y, add_two_ints_client(x, y))
```

Now, make the script executable –

```
$ chmod +x scripts/add_two_ints_client.py
```

We use CMake as our build system. This is to make sure that the autogenerated Python code for messages and services is created. Go to your catkin workspace and run catkin_make –

```
# In your catkin workspace
$ cd ~/catkin_ws
$ catkin_make
```

14. Examining the Simple Service and Client

Start by running the service –

```
$ rosrun beginner_tutorials add_two_ints_server.py
```

You should see the following –

```
Ready to add two ints.
```

Now let's run the client with the necessary arguments –

```
$ rosrun beginner_tutorials add_two_ints_client.py 1 3
```

You should see the following –

```
Requesting 1+3
1 + 3 = 4
```

15. Recording and playing back data

Finally, we will work on how to record topic data from a running ROS system. The topic data will be accumulated in a bag file. First, execute the following commands in separate terminals –

```
roscore
```

```
rosrun turtlesim turtlesim_node
```

```
rosrun turtlesim turtle_teleop_key
```

This will start two nodes - the turtlesim visualizer and a node that allows for the keyboard control of turtlesim using the arrows keys on the keyboard.

First let's examine the full list of topics that are currently being published in the running system. To do this, open a new terminal and execute the command –

```
rostopic list -v
```

We now will record the published data. Open a new terminal window. In this window run the following commands –

```
mkdir ~/bagfiles
cd ~/bagfiles
rosbag record -a
```

Now that we've recorded a bag file using rosbag record we can examine it and play it back using the commands rosbag info and rosbag play. First we are going to see what's recorded in the bag file. We can do the info command -- this command checks the contents of the bag file without playing it back. Execute the following command from the bagfiles directory:

```
rosbag info <your bagfile>
```

This tells us topic names and types as well as the number (count) of each message topic contained in the bag file. We can see that of the topics being advertised that we saw in the rostopic output, four of the five were actually published over our recording interval. As we ran rosbag record with the -a flag it recorded all messages published by all nodes.

Now, we want to replay the bag file to reproduce behavior in the running system. In a terminal window run the following command in the directory where you took the original bag file –

```
rosbag play <your bagfile>
```

A final option that may be of interest is the -r option, which allows you to change the rate of publishing by a specified factor. Executing the following command will allow you to see the turtle execute a slightly different trajectory - this is the trajectory that would have resulted had you issued your keyboard commands twice as fast –

```
rosbag play -r 2 <your bagfile>
```

Finally, The rosbag record command supports logging only particular topics to a bag file, allowing users to only record the topics of interest to them. If any turtlesim nodes are running exit them and relaunch the keyboard teleop launch file –

```
rosrun turtlesim turtlesim_node
rosrun turtlesim turtle_teleop_key
```

In your bagfiles directory, run the following command –

```
rosbag record -O subset /turtle1/cmd_vel /turtle1/pose
```

The -O argument tells rosbag record to log to a file named subset.bag, and the topic arguments cause rosbag record to only subscribe to these two topics. Move the turtle around for several seconds using the keyboard arrow commands, and then Ctrl-C the rosbag record.