EE106B: Lab 0 - Review of the Robot Operating System (ROS) *

Spring 2018

Goals

This lab draws from various parts of labs 1 through 4 from EECS 106A/206A. For a deeper look at the material, please refer to these documents.

By the end of this lab you should be able to:

- Set up a new ROS environment, including creating a new workspace and creating a package with the appropriate dependencies specified
- Use the `catkin` tool to build the packages contained in a ROS workspace
- Run nodes using `rosrun`
- Use ROS's built-in tools to examine the topics, services and messages used by a given node
- Write a new node that interfaces with existing ROS code

Contents

*Developed by Valmik Prabhu, Spring 2018. Compiled heavily from labs written by Aaron Bestick and Austin Buchan, Fall 2014.

1 Initial configuration

The lab machines you're using already have ROS and the Baxter robot SDK installed, but you'll need to perform a few user-specific configuration tasks the first time you log in with your class account.

First create a folder called `ros_workspaces` by typing

```
mkdir ros_workspaces
```

This folder will be used to hold the workspaces for each lab.

Now open the `.bashrc` file, located in your home directory (denoted " ~ "), in a text editor. (If you don't have a preferred editor, we recommend Sublime Text, which is preinstalled on lab computers and can be accessed using `subl <filename>`. In this case, you'd type `subl ~/.bashrc`.) Then add the following three new lines to the end of the file:

```
source /opt/ros/indigo/setup.bash
source /scratch/shared/baxter_ws/devel/setup.bash
export ROS_HOSTNAME=$(hostname --short).local
```

Save and close the file when you're done editing, then execute the command "`source .bashrc`" to update your environment with the new settings. Alternately, you can quit terminal completely and re-open it. The `.bashrc` file is automatically sourced each time you open a new terminal window, but is only updated when the command line is not running.

The first two additonal lines tell Ubuntu to run two ROS-specific configuration scripts every time you open a new terminal window. These scripts set several environment variables that tell the system where the ROS installation and the Baxter packages are located. When you create your own workspaces, you will need to run a workspace specific `setup.bash` file to ensure that your packages are located on the ROS path.

The third line edits the `$ROS_HOSTNAME` environment variable, which tells your code that your local computer is hosting ROS.

2 Creating ROS Workspaces and Packages

You're now ready to create your own ROS package. To do this, we also need to create a catkin workspace. Since all ROS code must be contained within a package in a workspace, this is something you'll do frequently.

2.1 Creating a workspace

A workspace is a collection of packages that are built together. ROS uses the `catkin` tool to build all code in a workspace, and do some bookkeeping to easily run code in packages. Each time you start a new project (i.e. lab or final project) you will want to create and initialize a new catkin workspace.

For this lab, begin by creating a directory for the workspace. Create the directory `lab0_ws` in your home `ros_workspaces` folder. The directory "`ros_workspaces`" will eventually conatin several lab-specific workspaces (named `lab1`, `lab2`, etc.) Next, create a folder `src` in your new workspace directory (`lab0_ws`). From the new `src` folder, run:

```
catkin_init_workspace
```

It should create a single file called `CMakeLists.txt`

After you fill `/src` with packages, you can build them by running "`catkin_make`" from the workspace directory (`lab0_ws` in this case). Try running this command now, just to make sure the build system works. You should notice two new directories alongside `src`: `build` and `devel`. ROS uses these directories to store information related to building your packages (in `build`) as well as automatically generated files, like binary executables and header files (in `devel`).

2.2 Creating a New Package

Let's create a new package. From the `src` directory, run

```
catkin_create_pkg lab0_turtlesim rospy roscpp std_msgs geometry_msgs turtlesim
```

Our package is called `lab0_turtlesim`, and we add `rospy`, `roscpp`, `std_msgs`, `geometry_msgs`, and `turtlesim` as dependencies. `rospy` and `roscpp` allow ROS to interface with code in Python and C++, `std_msgs` and `geometry_msgs` are both message libraries. Messages are data structures that allow ROS nodes to communicate. You'll learn more about them in Section 3.

2.3 Building a package

Now imagine you've added all your resources to the new package. The last step before you can use the package with ROS is to *build* it. This is accomplished with `catkin_make`. Run the command again from the `lab0_ws` directory.

```
catkin_make
```

`catkin_make` builds all the packages and their dependencies in the correct order. If everything worked, `catkin_make` should print a bunch of configuration and build information for your new package "`lab0_turtlesim`" with no errors. You should also notice that the `devel` directory contains a script called "`setup.bash`." "Sourcing" this script will prepare your ROS environment for using the packages contained in this workspace (among other functions, it adds "`~/ros_workspaces/lab0/src`" to the `$ROS_PACKAGE_PATH`). Run the commands

```
echo $ROS_PACKAGE_PATH
source devel/setup.bash
echo $ROS_PACKAGE_PATH
```

and note the difference between the output of the first and second `echo`.

Note: *Any time that you want to use a non-built-in package, such as one that you have created, you will need to source the* `devel/setup.bash` *file for that package's workspace.*

2.4 File System Tools

When working with ROS, you will invariably be working with many packages stored in many places. ROS provides a collection of tools to navigate. Some of the most useful are `rospack`, `rosls`, and `roscd`. Type

```
rospack find baxter_examples
```

This should return the directory at which the `baxter_examples` package is located. What do you think the others do? Note that these commands only work on packages in the `$ROS_PACKAGE_PATH`, so make sure to source the relevant workspace before using these commands.

3 Understanding ROS nodes

We're now ready to test out some actual software running on ROS. First, a quick review of some computation graph concepts:

- *Node*: an executable that uses ROS to communicate with other nodes
- *Message*: a ROS datatype used to exchange data between nodes
- *Topic*: nodes can *publish* messages to a topic as well as *subscribe* to a topic to receive messages

Now let's test out some built-in examples of ROS nodes.

3.1 Running roscore

First, run the command

```
roscore
```

This starts a server that all other ROS nodes use to communicate. Leave `roscore` running and open a second terminal window (`Ctrl+Shift+T` or `Ctrl+Shift+N`).

As with packages, ROS provides a collection of tools we can use to get information about the nodes and topics that make up the current computation graph. Try running

```
rosnode list
```

This tells us that the only node currently running is `/rosout`, which listens for debugging and error messages published by other nodes and logs them to a file. We can get more information on the `/rosout` node by running

```
rosnode info /rosout
```

whose output shows that `/rosout` publishes the `/rosout_agg` topic, subscribes to the `/rosout` topic, and offers the `/set_logger_level` and `/get_loggers` services.

The `/rosout` node isn't very exciting. Let's look at some other built-in ROS nodes that have more interesting behavior.

3.2 Running turtlesim

To start additional nodes, we use the `rosrun` command. The syntax is

```
rosrun [package_name] [executable_name]
```

The ROS equivalent of a "hello world" program is turtlesim. To run turtlesim, we first want to start the `turtlesim_node` executable, which is located in the `turtlesim` package, so we open a new terminal window and run

```
rosrun turtlesim turtlesim_node
```

A turtlesim window should appear. Repeat the two `rosnode` commands from above and compare the results. You should see a new node called `/turtlesim` that publishes and subscribes to a number of additional topics.

4 Understanding ROS topics

Now we're ready to make our turtle do something. Leave the `roscore` and `turtlesim_node` windows open from the previous section. In a yet another new terminal window, use `rosrun` to start the `turtle_teleop_key` executable in the `turtlesim` package:

```
rosrun turtlesim turtle_teleop_key
```

You should now be able to drive your turtle around the screen with the arrow keys.

4.1 Using rqt_graph

Let's take a closer look at what's going on here. We'll use a tool called `rqt_graph` to visulize the current computation graph. Open a new terminal window and run

```
rosrun rqt_graph rqt_graph
```

This should produce an illustration like Figure 1. In this example, the `teleop_turtle` node is capturing your keystrokes and publishing them as control messages on the `/turtle1/cmd_vel` topic. The `/turtlesim` node then subscibes to this same topic to receive the control messages.

Figure 1: Output of rqt_plot when running turtlesim.

4.2 Using rostopic

Let's take a closer look at the `/turtle1/cmd_vel` topic. We can use the `rostopic` tool. First, let's look at individual messages that `/teleop_turtle` is publishing to the topic. We will use "`rostopic echo`" to echo those messages. Open a new terminal window and run

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

Now move the turtle with the arrow keys and observe the messages published on the topic. Return to your `rqt_graph` window, and click the refresh button (blue circle arrow icon in the top left corner). You should now see that a second node (the `rostopic` node) has subscribed to the `/turtle1/cmd_vel` topic, as shown in Figure 2.

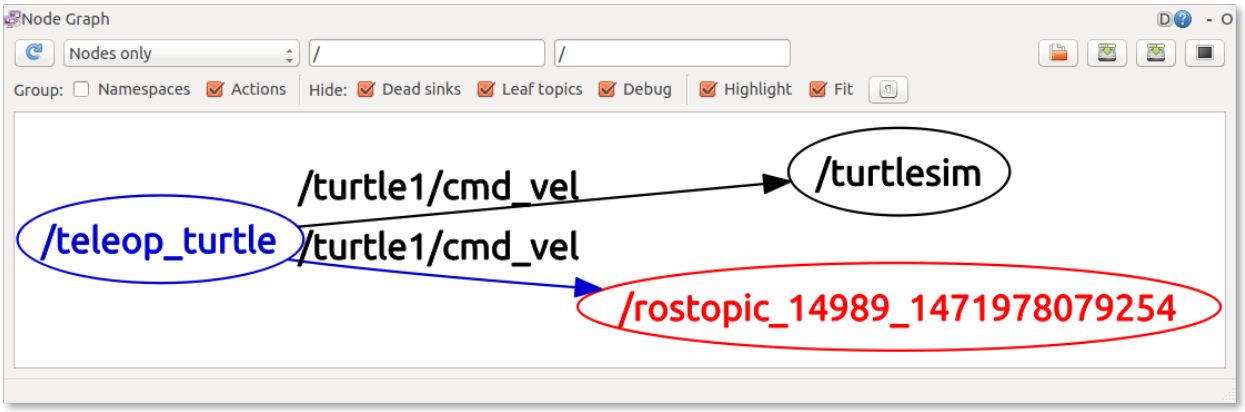

Figure 2: Output of rqt_graph when running turtlesim and viewing a topic using rostopic echo.

While `rqt_graph` only shows topics with at least one publisher and subscriber, we can view all the topics published or subscribed to by all nodes by running

```
rostopic list
```

For even more information, including the message type used for each topic, we can use the verbose option:

```
rostopic list -v
```

4.3 Examining ROS messages

Inter-node communication is done via messages, so understanding how to examine already-existing messages is an essential skill. Let's take a deep dive into the turtlesim command messages. Your `rostopic list` should produce the following output:

```
/rosout
/rosout_agg
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

We'll be looking at the `/turtle1/cmd_vel` topic. Type

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

As you can see, the message "`Type`" is `geometry_msgs/Twist`. Here `Twist` is the name of the message, and it's stored in the package `geometry_msgs`. ROS also has utility methods for messages, in addition to those for packages and topics. Let's use them to learn more about the `Twist` message. Type

```
rosmsg show geometry_msgs/Twist
```

Your output should be

```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

What do you think this means? Remember that a ROS message definition takes the following form:

```
<< data_type1 >>  << name_1 >>
<< data_type2 >>  << name_2 >>
<< data_type3 >>  << name_3 >>
...
```

(Don't include the << and >> in the message file.)

Each `data_type` is one of

- `int8, int16, int32, int64`
- `float32, float64`
- `string`
- other msg types specified as `package/MessageName`
- variable-length `array[]` and fixed-length `array[N]`

and each name identifies each of the data fields contained in the message.

Keep the turtlesim running for use in the next section.

5 Understanding ROS services

Services are another method nodes can use to pass data between each other. While topics are typically used to exchange a continuous stream of data, a service allows one node to *request* data from another node, and receive a *response*. Requests and responses are to services as messages are to topics: that is, they are containers of relevant information for their associated service or topic.

5.1 Using rosservice

The `rosservice` tool is analogous to `rostopic`, but for services rather than topics. We can call

```
rosservice list
```

to show all the services offered by currently running nodes.

We can also see what type of data is included in a request/response for a service. Check the service type for the `/clear` service by running

```
rosservice type /clear
```

This tells us that the service is of type `std_srvs/Empty`, which means that the service does not require any data as part of its request, and does not return any data in its response.

5.2 Calling services

Let's try calling the the `/clear` service. While this would usually be done programmatically from inside a node, we can do it manually using the `rosservice call` command. The syntax is

```
rosservice call [service] [arguments]
```

Because the `/clear` service does not take any input data, we can call it without arguments

```
rosservice call /clear
```

If we look back at the `turtlesim` window, we see that our call has cleared the background.

We can also call services that require arguments. Use `rosservice type` to find the datatype for the `/spawn` service. The query should return `turtlesim/Spawn`, which tells us that the service is of type `Spawn`, and that this service type is defined in the `turtlesim` package. Use `rospack find turtlesim` to get the location of the `turtlesim` package (hint: you could also use "`roscd`" to navigate to the `turtlesim` package), then open the `Spawn.srv` service definition, located in the package's `/srv` subfolder. The file should look like

```
float32 x
float32 y
float32 theta
string name
---
string name
```

This definition tells us that the `/spawn` service takes four arguments in its request: three decimal numbers giving the position and orientation of the new turtle, and a single string specifying the new turtle's name. The second portion of the definition tells us that the service returns one data item: a string with the new name we specified in the request.

Now let's call the `/spawn` service to create a new turtle, specifying the values for each of the four arguments, in order:

```
rosservice call /spawn 2.0 2.0 1.2 "newturtle"
```

The service call returns the name of the newly created turtle, and you should see the second turtle appear in the `turtlesim` window.

6 Understanding ROS Publishers and Subscribers

Navigate to the `Lab 0` folder in BCourses and download `Lab0_Resources.zip`. Inside, you will find a package called `chatter`. Move this to the `src` directory in your `lab0_ws` workspace, build it using `catkin_make`, and source the workspace.

Now, examine the files in the `src` directory inside `chatter`. `example_pub.py` and `example_sub.py` are both Python programs that run as nodes in the ROS graph. The `example_pub.py` program generates simple text messages and publishes them on the `/chatter_talk` topic, while the `example_sub.py` program subscribes to this same topic and prints the received messages to the terminal.

Close your turtlesim nodes from the previous section, but leave `roscore` running. In a new terminal, type

```
rosrun chatter example_pub.py
```

This should produce an error message. In order to run a Python script as an executable, the script needs to have the executable permission. To fix this, run the following command from the directory containing the example scripts:

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

Now, try running the example publisher and subscriber in different terminal windows and examine their behavior.

Study each of the files to understand how they function. Both are heavily commented.

7 Writing a controller for turtlesim

Let's replace `turtle_teleop_key` with a new controller, and learn how to interact with previously existing ROS code. Go back to `Lab0_Resources.zip` and put `controller.py` into the `src` folder inside the `lab0_turtlesim` package you created earlier.

We need `controller.py` to have the following functionality:

- Accept a command line argument specifying the name of the turtle it should control (e.g., running

  ```
  rosrun lab2_turtlesim turtle_controller.py turtle1
  ```

  will start a controller node that controls turtle1).

- Publish velocity control messages on the appropriate topic whenever the user presses certain keys on the keyboard, as in the original `turtle_teleop_key`. (It turns out that capturing individual keystrokes from the terminal is slightly complicated — it's a great bonus if you can figure it out, but feel free to use `raw_input()` and the WASD keys instead.)

Your first step is to figure out what topic to which to publish and which message type to use. Once you've figured that out, edit lines 15 and 29 accordingly. Then edit the "Your Code" section starting at line 38 to query the user for a command, process it, and set it as a variable of the correct message type.