EE106A: Lab 1 - Introduction to Robot Operating System (ROS) *

Fall 2017

Goals

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 and services used by a given node

---

A quick note: Most of this lab is borrowed from the official ROS tutorials at `http://www.ros.org/wiki/ROS/Tutorials`. We've tried to pick out the material you'll find most useful later in the semester, but feel free to explore the other tutorials too if you're interested in learning more.

Contents

*Developed by Aaron Bestick and Austin Buchan, Fall 2014.

- A motorized gripper on the end of the arm
- A stationary camera that observes the robot's workspace
- An infrared distance sensor next to the gripper on the manipulator arm

To pick up an object on a table, the robot might first use the camera to measure the position of the object, then command the arm to move toward the object's position. As the arm nears the object, the robot could use the IR distance sensor to detect when the object is properly positioned in the gripper, at which point it will command the gripper to close around the object. Given this sequence of tasks, how should we structure the robot's control software?

A useful abstraction for many robotic systems is to divide the control software into various low-level, independent control loops that each manage a single task on the robot, then couple these low level loops together with higher-level logic. In our example system above, we might divide the control software into:

- Two control loops (one for each joint) that, given a position or velocity command, control the power applied to the joint motor based on position sensor measurements at the joint
- Another control loop that receives commands to open or close the gripper, then switches the gripper motor on and off while controlling the power applied to it to avoid crushing objects
- A sensing loop that reads individual images from the camera at 30 Hz
- A sensing loop that reads the output of the IR distance sensor at 100 Hz
- A single high-level module that performs the supervisory control of the whole system

Given this structure for the robot's software, the control flow for the pick-and-place task could be the following: The high-level supervisor queries the camera sensing loop for a single image. It uses a vision algorithm to compute the location of the object to grasp, then computes the joint angles necessary to move the manipulator arm to this location and sends position commands to each of the joint control loops telling them to move to this position. As the arm nears the target, the supervisor queries the IR sensor loop for the distance to the object at a rate of 5 Hz and issues several more fine motion commands to the joint control loops to position the arm correctly. Finally, the supervisor signals the gripper control loop to close the gripper.

An important feature of this design is that the supervisor need not know the implementation details of any of the low-level control loops. It interacts with each only through simple control messages. This encapsulation of functionality within each individual control loop makes the system modular, and makes it easier to reuse the same code across many robotic platforms.

The ROS computation graph lets us build this style of software easily. In ROS, each individual control loop is a *node* within the computation graph. A node is simply an executable file that performs some task. Nodes exchange control messages, sensor readings, and other data by publishing or subscribing to *topics* or by sending requests to *services* offered by other nodes (these concepts will be discussed in detail later in the lab).

Nodes can be written in a variety of languages, including Python and C++, and ROS transparently handles the details of converting between different datatypes, exchanging messages between nodes, etc.

1.2 File system

As you might imagine, large software systems written using this model can become quite complex (nodes written in different languages, nodes that depend on third-party libraries and drivers, nodes that depend on other nodes, etc.). To help with this situation, ROS provides a system for organizing your ROS code into logical units and managing dependencies between these units. Specifically, ROS code is contained in *packages*. ROS provides a collection of tools to manage packages that will be discussed in more detail in the following sections.

2 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.

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
```

Note: If working on a VM, remove the "export ROS_HOSTNAME" line from the .bashrc

Save and close the file when you're done editing, then execute the command "`source .bashrc`" to update your environment with the new settings.

The first additonal line tells Ubuntu to run a ROS-specific configuration script every time you open a new terminal window. This script sets several environment variables that tell the system where the ROS installation is located.

The second line edits the `$ROS_PACKAGE_PATH` environment variable. This variable is particularly important, as it tells ROS which directories to search for software packages. Any code you want to run with ROS must be located beneath one of the directories specified in the list. By default, ROS's `setup.bash` file adds the directories for all of ROS's built-in packages to the package path. However, the Baxter SDK contains additional packages that we want to be able to run, so we must add its directory to the package path as well. The SDK is located in the `/scratch/shared` directory. 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 (we will discuss this in more detail in Section **??**).

3 Navigating the ROS file system

The basic unit of software organization in ROS is the *package*. A package can contain executables, source code, libraries, and other resources. A `package.xml` file is included in the root directory of each package. The `package.xml` contains metadata about the package contents, and most importantly, about which other packages this package depends on. Let's examine a package within the Baxter robot SDK as an example.

3.1 Anatomy of a package

Navigate to `/scratch/shared/baxter_ws/src/baxter_examples`. The `baxter_examples` package contains several example nodes which demonstrate the motion control features of Baxter. The folder contains several items:

- `\src` - source code for nodes
- `package.xml` - the package's configuration and dependencies
- `\launch` - launch files that start ROS and relevant packages all at once
- `\scripts` - another folder to store nodes

Other packages might contain some additional items:

- `\lib` - extra libraries used in the package
- `\msg` and `\srv` - message and service definitions which define the protocols nodes use to exchange data

Open `package.xml`. It should look something like this:

```
<?xml version="1.0"?>
<package>
  <name>baxter_examples</name>
  <version>1.2.0</version>
  <description>
    Example programs for Baxter SDK usage.
  </description>

  <maintainer email="rsdk.support@rethinkrobotics.com">
    Rethink Robotics Inc.
  </maintainer>
  <license>BSD</license>
  <url type="website">http://sdk.rethinkrobotics.com</url>
  <url type="repository">
    https://github.com/RethinkRobotics/baxter_examples
```

```
  </url>
  <url type="bugtracker">
    https://github.com/RethinkRobotics/baxter_examples/issues
  </url>
  <author>Rethink Robotics Inc.</author>

  <buildtool_depend>catkin</buildtool_depend>

  <build_depend>rospy</build_depend>
  <build_depend>xacro</build_depend>
  <build_depend>actionlib</build_depend>
  <build_depend>sensor_msgs</build_depend>
  <build_depend>control_msgs</build_depend>
  <build_depend>trajectory_msgs</build_depend>
  <build_depend>cv_bridge</build_depend>
  <build_depend>dynamic_reconfigure</build_depend>
  <build_depend>baxter_core_msgs</build_depend>
  <build_depend>baxter_interface</build_depend>

  <run_depend>rospy</run_depend>
  <run_depend>xacro</run_depend>
  <run_depend>actionlib</run_depend>
  <run_depend>sensor_msgs</run_depend>
  <run_depend>control_msgs</run_depend>
  <run_depend>trajectory_msgs</run_depend>
  <run_depend>cv_bridge</run_depend>
  <run_depend>dynamic_reconfigure</run_depend>
  <run_depend>baxter_core_msgs</run_depend>
  <run_depend>baxter_interface</run_depend>

</package>
```

Along with some metadata about the package, the `package.xml` specifies 11 packages on which `baxter_examples` depends. The `rospy` dependency is important - `rospy` is the ROS library that Python nodes use to communicate with other nodes in the computation graph. The corresponding library for C++ nodes is `roscpp`. The `build_depend` tags indicate packages used during the build phase. The `run_depend` tags indicate packages used during runtime.

3.2 File system tools

ROS provides a collection of tools to create, edit, and manage packages. One of the most useful is `rospack`, which returns information about a specific package. Try running the command

```
rospack find baxter_examples
```

which should return the same directory you looked at earlier.

Note: To get info on the options and functionality of many ROS command line utilities, run the utility plus "`help`" (e.g., just run "`rospack help`").

Next, let's test out a couple more convenient commands for working with packages. Run

```
rosls baxter_examples
```

and then

```
roscd baxter_examples
```

The function of these commands should become apparent quickly. Any ideas what they do?

4 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.

4.1 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 `ros_workspaces/lab1` in your home 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 (`lab1`). 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 (`lab1` 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`).

4.2 Creating a new package

You're now ready to create a package. From the `src` directory, run

```
catkin_create_pkg foo
```

Examine the contents of your newly created package, and open its `package.xml` file. By default, you will see that the only dependency created is for the catkin tool itself:

```
<buildtool_depend>catkin</buildtool_depend>
```

Next, we'll try the same command, but we'll specify a few dependencies for our new package. Return to the `src` directory and run the following command:

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

Examine the `package.xml` file for the new package and verify that the dependencies have been added. You're now ready to add source code, message and service definitions, and other resources to your project.

4.3 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 `lab1` 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 packages "`foo`" and "`bar`", 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/lab1/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.*

To summarize what we've done, here's what your directory structure should look like:

```
ros_workspaces
  lab1
    build
    devel
      setup.bash
    src
      CMakeLists.txt
      foo
        CMakeLists.txt
        package.xml
      bar
        CMakeLists.txt
        package.xml
        include
        src
```

Checkpoint 1

Get a TA to check your work (you can just do both checkpoints at the end of lab). You should be able to:

- Explain the contents of your `~/ros_workspaces` directory
- Demonstrate the use of the `catkin_make` command
- Explain the contents of a `package.xml` file
- Use ROS's utility functions to get data about packages

---

5 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.

5.1 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.

5.2 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.

6 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.

6.1 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 **??**. 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.

6.2 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 **??**.

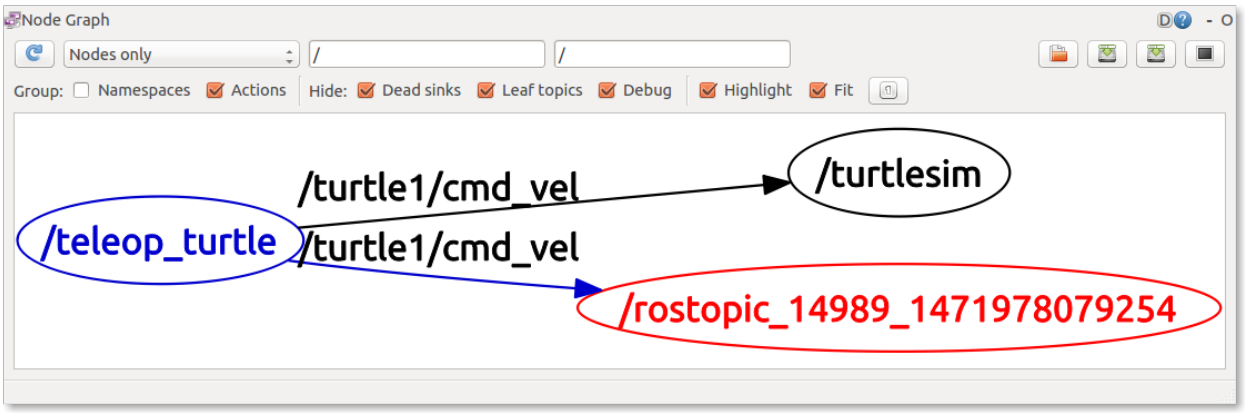

Figure 2: Output of rqt_graph when running turtlesim and viewing a topic using rostopic echo.

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
```

Keep the turtlesim running for use in the next section.

7 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.

7.1 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.

7.2 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.

Checkpoint 2

Get a TA to check your work. You should be able to:

- Explain what a *node*, *topic*, and *message* are
- Drive your turtle around the screen using arrow keys
- Use ROS's utility functions to view data on topics and messages

---