# EE106A: Lab 2 - Writing Publisher/Subscriber Nodes in ROS*

Fall 2018

---

## Goals

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

- Write ROS nodes in Python that both publish and subscribe to topics
- Define custom ROS message types to exchange data between nodes
- Create and build a new package with dependencies, source code, and message definitions
- Write a new node that interfaces with existing ROS code

---

*A quick note:* Most of Labs 1 and 2 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



# 1 Introduction

In Lab 1, you were introduced to the concept of the ROS computation graph. The graph is populated with *nodes*, which are executables that perform some internal data processing and communicate with other nodes by *publishing* and *subscribing* to *topics*, or by calling *services* offered by other nodes.

In this lab, you will explore how to write nodes that publish and subscribe to topics. ROS provides library code that takes care of most of the details of transmitting data via topics and services, which makes writing your own nodes quick and easy.

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

# 2 Examine a publisher/subscriber pair

Often the quickest way to learn new programming concepts is to look at working example code, so let's take a look at a publisher/subscriber pair that's already been written for you.

If you haven't already, download the `lab2_resources.zip` archive from the bCourses page, and extract the contents into the `~/ros_workspaces` directory you created in the previous lab. This task can be accomplished by running "`unzip lab2_resources.zip`" from within "`~/ros_workspaces`". You will notice that you now have an unbuilt workspace named "`lab2`". Build this workspace using "`catkin_make`". Recall that any packages you wish to run must be under one of the directories on the `ROS_PACKAGE_PATH`. Source the appropriate "`setup.bash`" file so that ROS will be able to locate the packages in the `lab2` workspace. Verify that ROS can find the newly unzipped package using "`rospack find chatter`".

Examine the files in the `/src` directory of the `chatter` package, `example_pub.py` and `example_sub.py`. Both are 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. In a new terminal window start the ROS master with the "`roscore`" command, then, in the original terminal, try executing "`rosrun chatter example_pub.py`", which 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. What happens if you start multiple instances of the publisher or subscriber in different terminal windows?

# 3 Write a publisher/subscriber pair

## 3.1 What you'll be creating

Now you're ready to write your own publisher/subscriber pair using the example code as a template. Your new publisher and subscriber should do the following:

### Publisher

1. Prompt the user to enter a line of text (you might find the Python function `raw_input()` helpful)

   ```
   Please enter a line of text and press <Enter>:
   ```

2. Generate a message containing the user's text and a timestamp of when the message was entered (you might find the function `rospy.get_time()` useful)

3. Publish the message on the `/user_messages` topic

4. Prompt the user for input repeatedly until the node is killed with `Ctrl+C`

### Subscriber

1. Subscribe to the `/user_messages` topic and wait to receive messages

2. When a message is received, print it to the command line using the format

   ```
   Message: <message>, Sent at: <timestamp>, Received at: <timestamp>
   ```

   (Note that the final `<timestamp>` is NOT part of the sent message; your new message type should contain only a single message and timestamp. Where does it come from?)

3. Wait for more messages until the node is killed with `Ctrl+C`

## 3.2 Steps to follow

To do this, you'll need to complete the following steps:

1. Create a new package (let's call it `my_chatter`) with the appropriate dependencies
2. Define a new message type that can hold both the user input (a string), and the timestamp (a number), and save this in the `msg` folder of the new package (discussed below)
3. Place the Python code for your two new nodes in the `src` directory of the package (if you create the Python file from scratch, you will need to make the file executable by running "`chmod +x your_file.py`")
4. Build the new package
5. Run and test both nodes

It might be interesting to see if you can detect any discrepancy between when the messages are created in the publisher and when they are received by the subscriber; this is why we ask you to print both!

### 3.2.1 Defining a new message

The sample publisher/subscriber from the previous section uses the primitive message type `string`, found in the `std_msgs` package. However, we need a message type that can hold both a string and a numeric (`float`) value, so we'll have to define our own.

A ROS message definition is a simple text file of the 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.

Create a new message description file called `TimestampString.msg`, and add the data types and names for our new message type. Save this file in the `/msg` subfolder of the `my_chatter` package. (You may need to create this directory.)

Now we need to tell `catkin_make` that we have a new message type that needs to be built. Do this by uncommenting (remove the `<!-- -->`) the following two lines in `my_chatter/package.xml`:

```
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
```

Next, update the following functions in `my_chatter/CMakeLists.txt` so they read exactly as follows, uncommenting and adding information as necessary:

```
find_package(catkin REQUIRED COMPONENTS rospy std_msgs message_generation)

add_message_files(FILES TimestampString.msg)

generate_messages(DEPENDENCIES std_msgs)

catkin_package(CATKIN_DEPENDS rospy std_msgs message_runtime)
```

At this point, you can build the new message type using `catkin_make`. You can verify that this worked by confirming the existence of the file `~/ros_workspaces/lab2/devel/lib/python2.7/dist-packages/my_chatter/msg/_TimestampString.py`. (This is the how ROS implements your high-level message descriptions as Python classes.) Inspect this file if you are curious, but *do not modify it*. You can confirm the contents of your new message type by running "`rosmsg show TimestampString`".

To use the new message, you need to add a corresponding `import` statement to any Python programs that use it:

```
from my_chatter.msg import TimestampString
```

Do this for the publisher and subscriber that you are writing.

---

## Checkpoint 1

Get a TA to check your work (you may continue if a TA is unavailable). At this point you should be able to:

- Explain all the contents of your `lab2` workspace
- Discuss the new message type you created for the `user_messages` topic
- Demonstrate that your package builds successfully
- Demonstrate the functionality of your new nodes using `TimestampString`

---

# 4 Write a controller for turtlesim

For the last part of this lab, let's write a new controller for the turtlesim node you used in the lab last week. This node will replace `turtle_teleop_key`. Since the `turtlesim` node is the subscriber in this example, you'll only need to write a single publisher node. Create a new package `lab2_turtlesim` (that depends on `turtlesim` and other appropriate packages) to hold your node.

Your node should do the following:

- 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). The Python package `sys` will help you get command line arguments.

- Publish velocity control messages on the appropriate topic (`rostopic list` could be useful) 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()` instead.)

When you think you have your node working, open a turtlesim window and spawn multiple turtles in it. Then see if you can open multiple instances of your new turtle controller node, each linked to a different turtle. What happens if you start multiple instances of the node all controlling the same turtle?

---

### Checkpoint 2

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

- Explain all the contents of your `lab2_turtlesim` package
- Show that your new package builds successfully
- Demonstrate the functionality of your new turtle controller node by controlling two turtles with different inputs

---