

# Lab 2: Writing Publisher and Subscriber Nodes in ROS \*

EECS/ME/BIOE C106A/206A Fall 2021

---

## 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
  - Write ROS nodes in Python to setup a service
  - Define custom ROS service type
  - Create and build new packages with dependencies, source code, message, and service definitions
  - Write nodes that interfaces with existing ROS code
- 

If you get stuck at any point in the lab you may submit a help request during your lab section at [tinyurl.com/106alab](https://tinyurl.com/106alab).

*Note:* Much of Labs 1 and 2 is borrowed from [the official ROS tutorials](#). We picked out the material you will find most useful in this class, but feel free to explore other resources if you are interested in learning more.

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Examine a publisher/subscriber pair</b>	<b>2</b>
<b>3</b>	<b>Write a publisher/subscriber pair</b>	<b>2</b>
3.1	What you'll be creating . . . . .	2
3.2	Steps to follow . . . . .	3
3.2.1	Defining a new message . . . . .	3
<b>4</b>	<b>Write a controller for turtlesim</b>	<b>6</b>
<b>5</b>	<b>Control turtlesim via service</b>	<b>7</b>

---

\*Developed by Aaron Bestick and Austin Buchan, Fall 2014. Additional content developed by Fangyu Wu, Fall 2021.

# 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 as well as request and respond to a service. 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.

## 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.

Our starter code is on GitHub for you to clone so that you can easily access any updates we make to the starter code. It can be found at [https://github.com/ucb-ee106/lab2\\_starter.git](https://github.com/ucb-ee106/lab2_starter.git). You can clone it by running

```
git clone https://github.com/ucb-ee106/lab2_starter.git
```

and then move the `lab2` directory into your `ros_workspaces` directory. We also highly recommend you make a **private** GitHub repository for each of your labs just in case.

You will notice that you now have an unbuilt catkin workspace named “`lab2`”. After moving it to your `ros_workspaces` directory, 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 (run “`source devel/setup.bash`” from the root of your catkin workspace) 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

```
roslaunch 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. Remember to source every new terminal you open!

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()` useful)

```
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. If you have difficulty, refer to Lab 1.
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 timestamps!

### 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 (`float64`) 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. Recall that you can create a new file by using `nano` or `subl` and providing the file name in the terminal. 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>
<exec_depend>message_runtime</exec_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 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 `rosmmsg show TimestampString`. Keep in mind that `TimestampString` is a class, and the input message must be instantiated as an object of this class.

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

Submit a checkoff request at [tinyurl.com/106alab](https://tinyurl.com/106alab) for a staff member to come and check off your work. 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 next 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

```
roslaunch 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.)

*Some things to keep in mind:*

- Are you writing a publisher or a subscriber? The starter code has examples of both that you can copy and modify for your purpose.
- How can you find out which topic the node should use for publishing or subscribing control messages? What command can you run to visualize the computation graph? What topic did the `turtle_teleop_key` node use for this in Lab 1?
- What ROS utility function can be used to find out the kind of message type that should be sent across a certain topic?
- What kind of linear and angular velocity values should you send to the turtle to make it go forward, backward, turn left, or turn right?

When you think you have your node working, open a turtlesim window and spawn multiple turtles in it (if you forget how to do so, look into Lab 1). 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

Submit a checkoff request at [tinyurl.com/106alab](http://tinyurl.com/106alab) for a staff member to come and check off your work. At this point 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
-

## 5 Control turtlesim via service

Now that we have learned how to use publishers and subscribers to control turtles in turtlesim, we will explore an alternative communication pattern: servers/requests/services. Unlike the publisher/subscriber/topics pattern, where publishers and subscribers are running asynchronously, a service's servers and clients operate like regular function calls in programming. A typical service call consists of the following steps:

1. A client requests a service from a server and *waits* for a response.
2. The server receives the request, fulfills the service call, and returns a response to the client.
3. The client receives the response and proceeds to next steps.

An important distinction between service calls and subscriber callbacks in a node is that service calls execute in *sequence* whereas subscriber callbacks execute in *parallel*.

To define a service in ROS in Python, we will need to define three things: a service type, a server node, and a client node. For example, take a look at the package `turtle_patrol` in `lab2`. You will notice that the package contains the following content:

```
CMakeLists.txt
package.xml
src
  patrol_client.py
  patrol_server.py
srv
  Patrol.srv
```

Examine the `Patrol.srv` file, which should look like this:

```
float32 vel
float32 omega
---
geometry_msgs/Twist cmd
```

The file defines a service type with a request and a response that are separated by a `---` line. The request consists of two `float32` messages: one for the translational velocity of the turtle and one for the angular velocity. The response consists of the `geometry_msgs/Twist` message.

The `patrol_server.py` file defines a node that provides the service. Examine the content of this script and make sure you understand how the node is syntactically constructed. The `patrol_client.py` file defines a node that uses the service. While you won't be editing this file in this lab, you can take a look at how this file works as an example of calling services from a node.

Next, let us run the service as follows:

- Close all previous ROS nodes
- Open a new turtlesim node

```
roslaunch turtlesim turtlesim_node
```

- Run the server node

```
roslaunch turtle_patrol patrol_server.py
```

- Run the client node that calls the service

```
roslaunch turtle_patrol patrol_client.py
```

After successfully invoking the service, you will find that turtle1 starts to patrol in a circle!

You can also conveniently call the service from the command line with the command

```
rosservice call /turtle1/patrol [vel] [omega]
```

where you can replace [vel] and [omega] with the parameters you would like to use. Try calling it a few times with different parameters to see how its behavior changes.

Finally, let us modify the service to control multiple turtles to patrol simultaneously in circles with initial poses specified in respective service calls. For this to work, you will need to modify the service type and server node. (*Hints:* To specify which turtle to patrol, you may pass an additional command line argument to the server node and use that argument to publish command messages to appropriate topics. To set initial patrol poses, you may add three arguments in the service request type to set initial  $x$  coordinate,  $y$  coordinate, and orientation  $\theta$ .) Once you think you have a working implementation, spawn a few more turtles in turtlesim and use several `rosservice` calls to make multiple turtles patrol in circles. For example, your service call may look like the following:

```
rosservice call /[turtle_name]/patrol [vel] [omega] [x] [y] [theta]
```

*Some things to keep in mind:*

- How did you write the Python code to use command line arguments to specify which turtle to control in the previous checkpoint? You should code it in a similar way when modifying the server.
- How did you write your custom message type in the first checkpoint? The service file should look somewhat similar to a message file, just with 2 parts. You will just need to add some additional information to the request part.
- How many server nodes do you need to run when making multiple turtles patrol? Do you need to run one server node per turtle? How will you tell ROS which turtle's patrol service to call?

Have fun building an army of patrolling turtles!

Next week, we will start using the robots! Please read the [Robot Usage Guide](#) and complete the Robot Usage Quiz on Gradescope.

---

## Checkpoint 3

Submit a checkoff request at [tinyurl.com/106alab](https://tinyurl.com/106alab) for a staff member to come and check off your work. At this point you should be able to:

- Explain how service calls get invoked differently from subscriber callbacks in a ROS node.
  - Show that your modified service can make multiple turtles patrol in circles from different initial poses.
  - Show that you have passed all of the questions on the Robot Usage Quiz on Gradescope
-