Lab 2 - Writing Publisher/Subscriber Nodes in ROS*

EECS/ME/BIOE C106A/206A Fall 2023

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
- Understand the use and basic implementation of a service
- Write a new node that interfaces with some existing ROS code

---

Note: Much of this lab 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

1 Introduction

In Lab 1, you were introduced to the concept of the ROS computation graph. The graph is populated with:

- **nodes**: an executable representing an individual ROS software process, typically performing some internal data processing and communicating with other nodes
- **topics**: nodes can *publish* or *subscribe* to a topic to receive messages
- **services**: nodes can request services offered by other nodes to receive a response

*Rewritten in Fall 2023 by Shrey Aeron. Developed by Aaron Bestick and Austin Buchan, Fall 2014.

In Lab 2, you will explore how to write your own ROS nodes and enable them to communicate with each other by sending data through topics and services. Luckily, this process is fairly straightforward, as ROS provides a simple framework to construct these pipelines.

Before every lab, you should pull the updated starter code. In terminal, navigate to the `~/ros_workspaces` directory and ensure that you have pushed all Lab 1 changes to Github (as outlined in the Github section of Lab 1). Then, run

```
git pull starter main
```

to pull the lab 2 starter code.

2 Publisher/Subscriber Pair Example

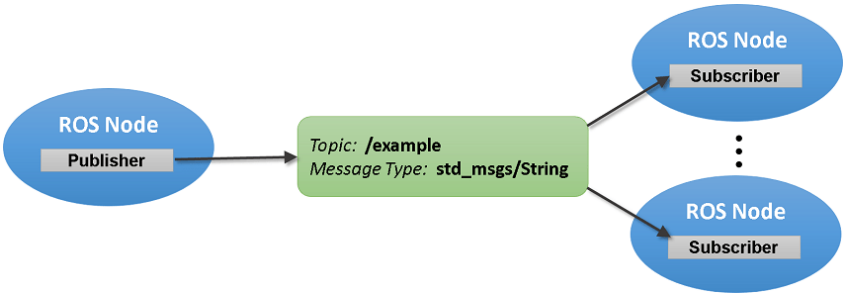

Figure 1: Message-based publishers and subscribers. Source: MathWorks

Your `ros_workspaces` folder should contain the new (unbuilt) workspace `lab2`, containing two packages: `chatter` and `turtle_patrol`. Let's examine the pre-written publisher/subscriber pair in `chatter` to better understand how they work together.

1. Build the `lab2` workspace using `catkin_make`.
2. Source the appropriate `setup.bash` file ("`source devel/setup.bash`") so that ROS will be able to locate the `lab2` packages.
3. Verify that ROS can find the newly unzipped package by running `rospack find chatter`.

The `/src` directory of the `chatter` package should contain two files: `example_pub.py` and `example_sub.py`, which are Python programs that run a ROS node. The `example_pub.py` program generates text messages and publishes them on the `/chatter_talk` topic, while the `example_sub.py` program subscribes to this topic and prints the received messages to the terminal.

1. In a new terminal window, start the ROS master with the `roscore` command. Then, in the original terminal, try running the publisher by executing

   ```
   rosrun chatter example_pub.py
   ```

   If you get an error message, it may be because the .py script needs to have executable permission (Check out this link for more linux file permissions info). To fix this, run the following command from the directory containing the example scripts:

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

2. Now that the publisher is running, run the subscriber by opening a new terminal window and following a similar process to run `example_sub.py`. Examine the behavior of the publisher and subscriber.
3. Look over each of the files to understand how they work. Take advantage of the provided comments to understand their contents. *What happens if you start multiple instances of the same publisher or subscriber file in different terminal windows?*

3 Writing a publisher/subscriber pair

Note: Please read through all of Section 3 before beginning coding!

Some general tips:

- You can create a new file from the terminal by using `subl <filename>`.
- Make sure to source the workspace `setup.bash` file before running any programs

3.1 What you'll be creating

Now you're ready to write your own publisher/subscriber pair! 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 `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. Continue prompting the user for input until the node is killed

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: The received `<timestamp>` is NOT part of the received message, which should contain only a single message and timestamp. Where does it come from?

3. Wait for more messages until the node is killed

3.2 Steps to follow

We recommend using the example code in `chatter` to help you get started. You'll need to complete the following steps:

1. Create a new package named `my_chatter` with the appropriate dependencies, referencing the `chatter` package. If you have difficulty, refer to Lab 1, Section 5.2.
2. Define a new message type that holds the user input (a string) and message timestamp (a number). Save this in the `msg` folder of the `my_chatter` package, as in Section 3.2.1.
3. Inside of your new package's `src` directory, write Python code for your two new publisher and subscriber nodes. Make the files 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 in `chatter` uses the primitive message type `string`, found in the `std_msgs` package. However, to send both the message text and its timestamp, we need to create a new custom message type with both data structures in it.

A ROS message definition is a text file of the form:

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

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]`

Each `name` identifies one of the data fields contained in the message and must be unique.

3.2.2 Creating the message

In the `msg` folder of the `my_chatter` package, create a new message description file called `TimestampString.msg`. Ensure that it contains the two data types we desire based on the information above.

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

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

Additionally, 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)
```

Take a brief moment to look at this code block to understand what each line might mean.

Now, you can build the new message type using `catkin_make`. Verify that your message is being built correctly by confirming the existence of the file `~/ros_workspaces/lab2/devel/lib/python3/dist-packages/my_chatter/msg/_TimestampString.py`, which should be generated by ROS to understand your message formats. Inspect this file if you are curious, but *do not modify it*.

Additionally, you can confirm the data in your new message type by running "`rosmsg 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 will add a corresponding `import` statement to any Python programs that use it:

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

Do this for both the publisher and subscriber you are writing, integrating the new `TimestampString` type you just created. Think about how you would create a new instance of this type when sending the message from your publisher

Soon enough, you'll be running your own news agency!

Checkpoint 1

Submit a checkoff request at tinyurl.com/fa23-106alab 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`

This is also a good point to switch typers!

---

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.

1. Create a new package `lab2_turtlesim` (that depends on `turtlesim` and other appropriate packages) to hold your node:

   ```
   catkin_create_pkg lab2_turtlesim rospy roscpp std_msgs geometry_msgs turtlesim
   ```

Your node should do the following:

1. Accept a command line argument specifying the name of the turtle it should control. 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 (specifically, sys.argv)

2. 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 `input()` instead.)

3. When you think you have your node working, open a turtlesim window (see lab 1) and spawn multiple turtles in it:

   ```
   rosservice call /spawn 2.0 2.0 1.2 "<<name_of_turtle>>"
   ```

4. See if you can open multiple instances of your new turtle controller node, each linked to a different turtle.

   Note the name of the turtle when running `turtle_controller.py` should match `<<name_of_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/fa23-106alab for a TA to come and check off 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

---

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, **synchronous** function calls in programming.

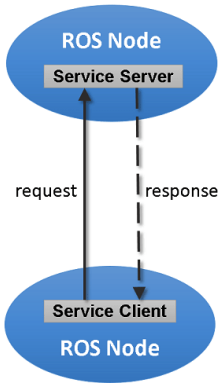

Figure 2: Service-based communication. Source: MathWorks

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 the 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 must 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 new service type with a request and a response 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, which the turtlesim node requires.

The `patrol_server.py` file defines a node that provides the service. Look at this script to understand how the node works. The `patrol_client.py` file defines a node that uses the service. This file serves as an example of how services

are called programmatically from a node.

Next, let us run the service as follows:

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

```
rosrun turtlesim turtlesim_node
```

- Run the server node

```
rosrun turtle_patrol patrol_server.py
```

- Run the client node that calls the service

```
rosrun turtle_patrol patrol_client.py
```

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

You can also 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.

Now that you have understood how the service can control one turtle, we will mirror what we did in the publisher/-subscriber pair and change the code to control multiple turtles with service calls simultaneously. For example, your service call may look like the following:

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

For this to work, you will need to modify the service datatype and server node. Here are some hints that may help:

- You may need to pass an additional argument to the server to specify which turtle to patrol. This will be used to publish on the right topic.
- To set initial patrol poses, you may add three arguments in the service request type to set initial x coordinate, y coordinate, and orientation θ.

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.

Some things to keep in mind:

- Look back at the previous checkpoint to understand how to ingest additional command line arguments.
- You may need to modify the service file to communicate the newly added inputs on the request side.
- How many server nodes are needed to control patrol turtles? 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 *real* robots!

Checkpoint 3

Submit a checkoff request at tinyurl.com/fa23-106alab for a TA to come and check off your work. You should be able to:

- Explain how service calls differ from subscriber callbacks in a ROS node.
- Show that your modified service can make multiple turtles patrol in circles from different initial poses. Explain why your architecture works.

Congrats on finishing Lab 2! Stay strong, you got this :)