AA 274: Principles of Robotic Autonomy
Section 2: ROS and Workstation

Our goals for this section:

1. Gain a basic understanding of the Robot Operating System (ROS) by implementing classes, nodes, and topics. Use catkin build tools and other commands to interact with ROS through the terminal.
2. Learn useful workstation commands.
3. Learn how group login accounts will work for the rest of the class.

1 ROS

1.1 What is ROS?

Although ROS is termed the Robot Operating System, it's not a full-fledged OS like Windows or Unix. More accurately, it's a set of programs (mostly written in C++) that perform many of the basic tasks that we need for robotics.

Over the course of this quarter, you'll master many of these components by using them for homeworks, sections, and your final project. However, ROS is a full-stack system, meaning that ROS programs will encompass everything from the lowest level drivers to the highest level visualizers. This means that ROS will break, and it will break often. Most of these bugs have been experienced before, and Google will become your best friend. However, if you get stuck on any one issue, please reach out to a TA. An important goal of this class is to teach you how to fix ROS when it fails, but you shouldn't be spending the majority of your time on bugs.

So what are some of the basic tasks that ROS implements for us?

1. Communication: A robot is a collection of hundreds of software programs interacting with one another. Therefore, there has to be some way for these programs to communicate with one another. ROS implements this communication for us.
2. Visualization: One of the most crucial things we need to do is visualize how our robot is performing. ROS provides multiple tools for visualizing a robot's internal processes.
3. Package management: We don't want to rewrite our robot's programs from scratch. ROS provides a way for downloading and managing community-sourced packages.
4. And many more including ... simulators, debuggers, planners, controllers, drivers, 3D processing, grasping, motion tracking, face recognition, and stereo vision.

Before we jump into these, let's go over the basics of how to use ROS.

1.2 Starting ROS

To start ROS, run the following command

```
roscore
```

This command starts a ROS master, which is just a naming service. Any time a new node starts in the system, it will have to register with the master. Then, the master will keep track of that node until it closes. The ROS master, in turn, provides a bridge that allows nodes to communicate with each other.

Running roscore will also start a few other processes including **rosout**, which is a ROS-specific stdout, and a **parameter server** that allows you to share parameters across nodes.

What's a **node**? A node is any executable program that uses ROS to communicate. So, when you run any program that uses ROS to communicate, it is considered a node.

To list all of your running nodes, run:

```
rosnode list
```

1.3 ROS Communication

One way that nodes can communicate with each other is by sending **messages** over **topics**. A **message** is a strong-typed set of data, and the structure of that data is a **.msg** file. A topic is an address that nodes can either send data to or receive data from. Importantly, only one message type can be sent over a topic.

Some standard messages are included in ROS libraries like std_msgs and $geometry_msgs$.

For example, here is the *String.msg* file in std_msgs.

```
string data
```

It's a single line!

Here is the *Pose.msg* file in $geometry_msgs$.

```
geometry_msgs/Point position
geometry_msgs/Quaternion orientation
```

Note that it references message types defined in other message files.

ROS uses these pre-defined message types so that nodes can know how to communicate with one another over a given topic.

Problem 1: Create your own message file consisting of multiple standard data types. This can be bool, string, float64, char, int64, and many more.

See $http://docs.ros.org/melodic/api/std_msgs/html/index-msg.html$ for the full list of standard messages.

1.4 Publishing and Subscribing

Now that we've created our custom message type, let's create a script that will publish a message to a topic. All our code will be in Python, using a library called *rospy*, but you can also write the same scripts in C++ using the *roscpp* library. This script is included in the section code, but it is reproduced below.

```
import rospy
from aa274_s2.msg import MyMessage

def publisher():
    pub = rospy.Publisher('my_topic', MyMessage, queue_size=10)
    rospy.init_node('my_node', anonymous=True)
    rate = rospy.Rate(1)
    while not rospy.is_shutdown():
        my_message = MyMessage()
        pub.publish(my_message)
        rate.sleep()

if __name__ == '__main__':
    try:
        publisher()
    except rospy.ROSInterruptException:
        pass
```

Let's also create a script that will subscribe to the same topic and print each message it receives.

```
import rospy
from aa274_s2.msg import MyMessage

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)

def subscriber():
    rospy.init_node('my_subscriber', anonymous=True)
    rospy.Subscriber("my_topic", MyMessage, callback)
    rospy.spin()

if __name__ == '__main__':
    subscriber()
```

1.5 Making a node

Now that we have our message, publisher, and subscriber, let's create a fully functioning node.

The core build system used by ROS is called Catkin. When working with C code, we usually have to use a tool like cmake to build and package our code. Catkin is simply the ROS equivalent of that.

All of the code we write will be located inside of a catkin workspace. To initialize this workspace run:

```
cd ~
mkdir catkin_ws
cd catkin_ws
catkin init
```

You will see that this creates a src folder in your directory. Now to create a new package for our code, run:

```
catkin_create_pkg aa274_s2 std_msgs rospy message_generation
```

This will create a new package called aa274_s2 in the src folder. The last three arguments are library dependencies that this package will require to run.

Now change into the aa274_s2 directory. You will see that there are three items in it: CMakeLists.txt, package.xml, and an empty src folder. Go ahead and copy the scripts folder and the msg folder from the section code into aa274_s2. Now, let's take a few minutes to inspect those first two files.

CMakeLists.txt is the most important file here, since it specifies what needs to be built and generated when we run our catkin build command. Because we have custom messages, we need to take an extra step.

At the bottom of this file, add the following

```
add_message_files(FILES MyMessage.msg)
```

```

generate_messages(
  DEPENDENCIES
  std_msgs
)
```

Without this declaration, catkin would not know to look for our custom message and any attempt to use it in another script would result in an error.

Next, ensure that both of your scripts in your scripts folder are executable by running:

```
chmod +x scripts/publisher.py
chmod +x scripts/subscriber.py
```

Now, switch back into the catkin_ws folder. You're ready to build your package. To build, run

```
catkin build aa274_s2
source devel/setup.bash
```

The first command calls catkin to build our package, and the second command updates the ROS environment so that it recognizes your newly built package.

Note that there is another, older way of building with catkin called catkin_make, but we will not use it in this class.

Now, you should have a fully functional package! You can now run your scripts in one of two ways. You can directly treat them like Python scripts by switching into the scripts folder and running

```
python publisher.py
```

Or, you can run your script from anywhere using the rosrun command:

```
rosrun aa274_s2 publisher.py
```

The advantage of this second method is that it allows you to run your script from anywhere on your system, while the first method requires you to know the full path to the script.

Problem 2: Try running both of your scripts now using one of these methods. Note that if your custom message does not include a "data" member, then the subscriber will error. To fix this, change the callback to print either one or several valid members.

Now that we've created a simple publisher and subscriber using the ROS topic paradigm, let's see how else nodes can communicate.

1.6 Making a ROS service

One other way that nodes can communicate is through services. One disadvantage of using topics is that you end up publishing a lot of unused data. Furthermore, in a distributed system, we need a way for programs to request and reply to one another. This is where services come in.

Services allow us to specify a client and a service. A client can send a request message to a service and receive a reply. Similar to messages, we must pre-define services using a a **.srv** file. As an example, let's define a service that takes two strings and returns them concatenated. Create a new folder **srv** and a new file **cat.srv** with the following:

```
string a
string b
---
string cat
```

This specifies that the cat service expects two strings a and b as input and that the client should expect a string cat as output.

As with messages, we have to configure our CCMakeLists.txt to properly build this service file.

```
add_service_files(
  FILES
  Cat.srv
)
```

Now, we can implement our client and service

```
Service:
from aa274_s2.srv import Cat,CatResponse
import rospy

def handle_cat(req):
    print("Returning [%s + %s = %s]"%(req.a, req.b, (req.a + req.b)))
    return CatResponse(req.a + req.b)

def cat_service():
    rospy.init_node('cat_service')
    s = rospy.Service('cat', Cat, handle_cat)
    rospy.spin()

if __name__ == "__main__":
    cat_service()
```

```
Client:
from aa274_s2.srv import Cat,CatResponse
import rospy
import sys

def cat_client(x, y):
    rospy.wait_for_service('cat')
    try:
        cat_proxy = rospy.ServiceProxy('cat', Cat)
        resp1 = cat_proxy(x, y)
        return resp1.cat
    except rospy.ServiceException, e:
        print("Service call failed")

if __name__ == "__main__":
        s1 = "hi"
        s2 = "hello"
        result = cat_client(s1,s2)
        print(result)
```

Once again, we build the catkin environment.

To check that our service correctly appears, we can run:

```
rossrv show Cat
```

1.7 Making a ROS action

The big disadvantage with services is that they do not let us know about the progress of a service call. This might be fine for short services. However, if we want to do something like plan a path through a room, we want to be able to know more about the state of our call.

This is where actions come in. Actions are just like services, but use a goal, result, and feedback instead of a request and response.

1.8 Setting a ROS parameter

Sometimes, we just want to share specific values between programs. ROS allows us to do so using a parameter server, which is just a shared dictionary. To view how you can interact with ROS parameters, type **rosparam** into your terminal.

You can also directly access parameters from Python using rospy, as below:

```
rospy.get_param("")
rospy.set_param("")
rospy.search_param("")
rospy.delete_param("")
```

Problem 4: What is the difference between a message, service, action, and parameter?

2 Workstation

1. htop
2. screen
3. nvidia-smi

3 Group Accounts

Will be discussed in section.