# AA 274A: Principles of Robot Autonomy I
# 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.

**NOTE**: The commands below were written from the perspective of a Ubuntu user. If you are using the VMware VM then these commands should work within it. If you are using Docker then remember to use the `run.sh` script from last section to run the ROS commands. Also, be mindful of the paths you use to execute files (keep in mind Docker's remapping of directories).

To obtain the code for this section, navigate to your section directory from last time, and execute the following

```
git pull
```

## 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 # In Docker, this would be ./run.sh 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 # In Docker, this would be ./run.sh 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 strongly-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/kinetic/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.

Take a look through the `publisher.py` and `subscriber.py` files located in the code folder of this section to see how `rospy` works.

**Problem 2: Create a publisher and subscriber that publish and subscribe to your custom message type, respectively.**

### 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 create a new package for our code, run the following from within the `catkin_ws/src` directory:

```
catkin_create_pkg aa274_s2 std_msgs rospy message_generation
# In Docker, the easiest way to do this would be to first get a shell with
# ./run.sh bash
# and then cd to the src directory, followed by running the
# catkin_create_pkg ... command.
```

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`. If you're using Docker, remember that you can change files as normal (without Docker) under the `catkin_ws` folder, since they are mapped to the Docker container during use. 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. Since we have custom messages, we need to take an extra step.

At the bottom of this file, at the end of the "declare ROS messages, services and actions" (before `catkin_package()` is called), 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 (this is not necessary if you're using Docker). You're ready to build your package. To build, run

```
catkin_make # In Docker, this would be ./run.sh catkin_make
source devel/setup.bash # Not necessary in Docker.
```

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.

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
python subscriber.py
```

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

```
rosrun aa274_s2 publisher.py
rosrun aa274_s2 subscriber.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.

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.

**Problem 3: Include screenshots or terminal output text that shows your publisher and subscriber are working.**

### 1.6 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("")
```

### 1.7 Helpful ROS commands

Here is a list of ROS commands that you will use often to understand what topics are being used.

1. `rostopic list` - This lists all of the topics that have been used or are in use.
2. `rostopic echo` -This echos the messages that are being sent to the particular topic.
3. `rostopic hz` - This returns the frequency at which the topic is being published to
4. `rostopic pub` - This allows you to publish to a topic. It's useful for debugging a subscriber.

**Problem 4: Run the first three of these to**

**(a) See that your topic is registered and visible**

**(b) Show what your publisher is publishing**

(c) **Determine the frequency with which your publisher is publishing messages.**

**Include these in your writeup.**

## 2 Workstation

As some of you may have realized last section (and potentially even this section), VMs can be computationally expensive, making them cumbersome to work with. As a result, we've obtained a powerful server which gives you a VM-free ROS option in this course. For any homework that has a ROS or Gazebo question, you can either use your own VMs or this workstation. Further, if you or your robot group need to simulate something with Gazebo for your final project, you can also use this workstation.

Please note that to interact with your robots later on, you'll need to have a local machine with ROS on it (the workstation is on a different network). Otherwise, if the homework only requires Python then you should use your local machines and/or Stanford FarmShare machines.

Below are several useful commands you'll use to either interface with the workstation or directly use it.

1. `ssh` - Use this command to access the server
2. `htop` - Use this command to see what processes are currently running
3. `screen` - Use this to open new screens that can run over ssh, even if you disconnect.
4. `nvidia-smi` - Use this to see what processes are using the GPU.
5. `scp` - Use this to copy files from your machine to the workstation or vice versa.

**Problem 5: Once logged into the machine, determine the following**

(a) **How many GPUs are there?**

(b) **How much RAM is available on the machine?**

(c) **How many CPU cores are there?**

(d) **What version of Python is available on the machine?**

**Include these in your writeup.**

## 3 Group Accounts

There are 30 group accounts on the machine (30 because we will support 30 final project groups because we have 30 robots). They are named `group01` to `group30`, each identically set up with ROS and all its dependencies.