Lab 1: Introduction to Robot Operating System (ROS) *

EECS/ME/BIOE C106A/206A Fall 2023

Goals

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

- Set up a new ROS environment, including creating a new workspace and creating a package with the appropriate dependencies specified
- Use the `catkin` tool to build the packages contained in a ROS workspace
- Run nodes using `rosrun`
- Use ROS's built-in tools to examine the topics and services used by a given node

---

If you get stuck at any point in the lab you may submit a help request during your lab section at tinyurl.com/fa23-106alab.

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 What is ROS? 2

2 Initial configuration 4
2.1 Workspace setup . . . 4
2.2 Shell Environment Setup . . . 4

3 Navigating the ROS file system 5
3.1 File system . . . 5
3.2 Anatomy of a package . . . 5
3.3 File system tools . . . 7

4 Creating ROS Workspaces and Packages 7
4.1 Creating a workspace . . . 7
4.2 Creating a new package . . . 7
4.3 Building a package . . . 8

5 Understanding ROS nodes 10
5.1 Running roscore . . . 10
5.2 Running turtlesim . . . 10

*Rewritten in Fall 2023 by Mingyang Wang and Eric Berndt. Minor edits for new protocols in Fall 2022 by Emma Stephan, and in Fall 2021 by Josephine Koe and Jaeyun Stella Seo. Developed by Aaron Bestick and Austin Buchan, Fall 2014.

6 Understanding ROS topics **11**
6.1 Using rqt_graph . . . 11
6.2 Using rostopic . . . 11

7 Understanding ROS services **12**
7.1 Using rosservice . . . 12
7.2 Calling services . . . 12

8 Lab Checkoffs on Gradescope **13**

9 Pushing to Github **13**

10 Additional Lab Etiquette **14**

1 What is ROS?

The ROS website says:

> ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.
>
> The ROS runtime "graph" is a peer-to-peer network of processes that are loosely coupled using the ROS communication infrastructure. ROS implements several different styles of communication, including synchronous RPC-style communication over services, asynchronous streaming of data over topics, and storage of data on a Parameter Server.

This isn't terribly enlightening to a new user, so we'll simplify by demonstrating ROS's functionality through an example. Consider a two-joint manipulator arm for a pick-and place task.

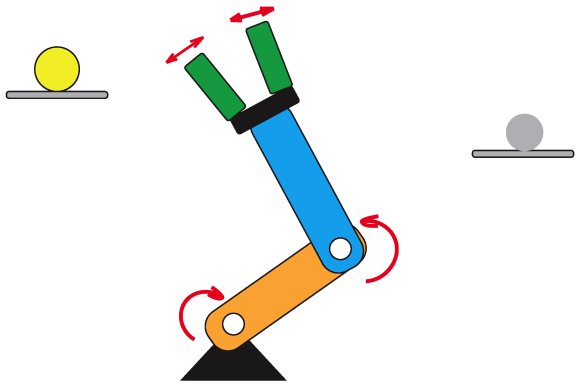

Figure 1: Example two-joint manipulator arm.

A typical robotic system has numerous sensing, actuation, and computing components. Suppose our system has:

- Two motors, each connected to a revolute joint (orange and blue)
- A motorized gripper on the end of the arm (green)
- A stationary camera that observes the robot's workspace

To pick up an object, the robot might:

- Use the camera to measure the position of the object
- Command the arm to move toward the object's position
- Once properly positioned, command the gripper to close around the object.

Given this sequence of tasks, how should we structure the robot's control software?

A useful abstraction for many robotic systems (and computer science in general) is to divide the control software into various low-level, independent control loops, each controlling a single task on the robot. In our example system above, we might divide the control software into:

- A control loop for each joint that, given a position or velocity command, controls the power applied to the joint motor based on position sensor measurements at the joint
- Another control loop that receives commands to open or close the gripper, then switches the gripper motor on and off while controlling the power applied to it to avoid crushing objects
- A sensing loop that reads individual images from the camera

Given this structure for the robot's software, we then couple these low-level loops together via a single high-level module that performs supervisory control of the whole system:

- Query the camera sensing loop for a single image.
- Use a vision algorithm to compute the location of the object to grasp
- Compute the joint angles necessary to move the manipulator arm to this location
- Sends position commands to each of the joint control loops telling them to move to this position
- Signal the gripper control loop to close the gripper to grab the object

An important feature of this design is that the supervisor need not know the implementation details of any of the low-level control loops: it interacts with each only through simple control messages. This encapsulation of functionality makes the system modular, making it easier to reuse code across robotic platforms.

In ROS, each individual control loop is known as a **node**, an individual software process that performs a specific task. Nodes exchange control messages, sensor readings, and other data by publishing or subscribing to **topics** or by sending requests to **services** offered by other nodes (these concepts will be discussed in detail later in the lab).

Nodes can be written in a variety of languages (including Python and C++), and ROS transparently handles the details of converting between different datatypes, exchanging messages between nodes, etc.

We can then visualize the communication and interaction between different software components via a **computation graph**, where:

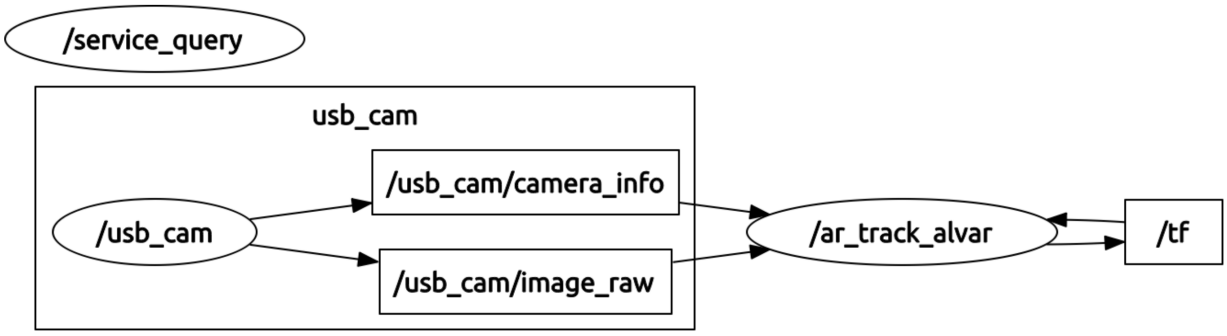

Figure 2: Example computation graph.

- Nodes are represented by ovals (ie `/usb_cam` or `/ar_track_alvar`)).
- Topics are represented by rectangles (ie `/usb_cam/camera_info` and `/usb_cam/image_raw`).
- The flow of information to and from topics and represented by arrows. In the above example, `/usb_cam` publishes to the topics `/usb_cam/camera_info` and `/usb_cam/image_raw`, which are subscribed to by `/ar_track_alvar`.
- While not shown here, services would be represented by dotted arrows.

2 Initial configuration

The lab machines you're using already have ROS and the Sawyer robot SDK installed, but you'll need to perform a few user-specific configuration tasks the first time you log in with your class account.

2.1 Workspace setup

Throughout the semester, course staff will publish the starter code to fix bugs or release new labs. Clone the starter code repository by running the following command from your home folder (~):

```
cd ~
git clone https://github.com/ucb-ee106/106a-fa23-labs-starter.git ros_workspaces
```

This clones the directory as "`ros_workspaces`" (we'll explain what this name means later), which currently should only contain a `/lab1` workspace. Since lab 1 doesn't contain any starter code, it should currently only contain a `pdf` of this lab document. `ros_workspaces` will eventually contain several lab-specific workspaces (`/lab2`, `/lab3`, etc.).

Run the following command to rename the remote from "`origin`" to "`starter`".

```
cd ~/ros_workspaces
git remote rename origin starter
```

Now, if you ever want to pull updated starter code, you'd execute the following command:

```
git pull starter main
```

Instructions to publish your lab code to your personal github repository are discussed in Section 9.

Two other useful commands to know are `rmdir` to remove an empty directory and `rm -r` to remove a non-empty directory.

2.2 Shell Environment Setup

The `.bashrc` file is a script that is executed whenever a new interactive non-login shell is started; it provides a convenient way to configure and customize your Bash shell environment.

Open the `.bashrc` file (located in your home directory, `~/.bashrc`) in a text editor. If you don't have a preferred editor, we recommend Sublime Text, which is preinstalled on lab computers and can be accessed using `subl <filename>`. Your `~/.bashrc` should look like

```
# Filename: .bashrc
# Description: This is the standard .bashrc for new named accounts.
#
# Please (DO NOT) edit this file unless you are sure of what you are doing.
# This file and other dotfiles have been written to work with each other.
# Any change that you are not sure off can break things in an unpredicatable
# ways.

# Set the Class MASTER  variables and source the class master version of .cshrc

[[ -z ${MASTER} ]] && export MASTER=${LOGNAME%-*}
[[ -z ${MASTERDIR} ]] && export MASTERDIR=$(eval echo ~${MASTER})

# Set up class wide settings
for file in ${MASTERDIR}/adm/bashrc.d/* ; do [[ -x ${file} ]] && . "${file}"; done

# Set up local settings
for file in ${HOME}/bashrc.d/* ; do [[ -x ${file} ]] && . "${file}"; done
```

There is no need to fully understand the code above, but note that `~/.bashrc` runs a series of set-up scripts, including all files in the `/home/ff/ee106a/adm/bashrc.d/` folder. One file of interest is `95-select_robot.sh`, which looks like

```
case "$HOSTNAME" in
   c105-1|c105-2|c105-3|c105-4|c105-5|c105-11 )
       source /home/ff/ee106a/turtlebot_setup.bash
       echo "Setup Turtlebot!" >&2
       ;;
   c105-6|c105-7|c105-8|c105-9|c105-10 )
       source /home/ff/ee106a/sawyer_setup.bash
       echo "Setup Sawyer!" >&2
       ;;
   * )
       echo "This is not a Sawyer or Turtlebot workstation!" >&2
       ;;
esac
```

This runs a different custom configuration script for ROS based on whether we are at a Turtlebot (1-5) or Sawyer lab station (6-10).

In the Turtlebot `turtlebot_setup.bash` script, we

- Run `source /opt/ros/noetic/setup.bash`, which sets up the default ROS environment in the terminal.
- Set the `$ROS_MASTER_URI` environment variable, which specifies where the ROS Master can be reached, to your local computer. The ROS Master will be detailed later in Section 5.1.

On the other hand, in the Sawyer `sawyer_setup.bash` script, we

- Run `source /opt/ros/eecsbot_ws/devel/setup.bash`, which sets up the ROS environment in the terminal just like `source /opt/ros/noetic/setup.bash` does, but also includes additional packages to interact with the Sawyer robot.
- Sets `$ROS_MASTER_URI` to be the Sawyer robot instead of the local computer.

3 Navigating the ROS file system

3.1 File system

Large software systems written using the ROS node, topic, and service model can quickly become quite complex (nodes written in different languages, nodes that depend on third-party libraries and drivers, nodes that depend on other nodes, etc.). To help with this situation, ROS provides a system for organizing your ROS code into logical units and managing dependencies between these units.

The basic unit of software organization in ROS is the **package**, a folder containing executables, source code, libraries, and other resources. In the following subsections, we will have you explore an example package for the Baxter robot SDK.

3.2 Anatomy of a package

`cd` into `/opt/ros/eecsbot_ws/src/baxter_examples`. The `baxter_examples` package contains several example nodes which demonstrate the motion control features of Baxter. The package contains several items:

- `package.xml` - the package's metadata, configuration, and dependencies; included in the root directory of every package.
- `/src` - source code for ROS nodes provided by the package
- `/launch` - launch files that start and configure ROS nodes with specific parameters and configurations
- `/scripts` - executable scripts, usually used to interact with ROS nodes

Other packages might contain some additional items:

- `/lib` - extra libraries used by the package
- `/msg` and `/srv` - message and service definitions nodes use to exchange data

Open the `package.xml` file. It should look something like this:

```
<?xml version="1.0"?>
<package>
  <name>baxter_examples</name>
  <version>1.2.0</version>
  <description>
    Example programs for Baxter SDK usage.
  </description>

  <maintainer email="rsdk.support@rethinkrobotics.com">
    Rethink Robotics Inc.
  </maintainer>
  <license>BSD</license>
  <url type="website">http://sdk.rethinkrobotics.com</url>
  <url type="repository">
    https://github.com/RethinkRobotics/baxter_examples
  </url>
  <url type="bugtracker">
    https://github.com/RethinkRobotics/baxter_examples/issues
  </url>
  <author>Rethink Robotics Inc.</author>

  <buildtool_depend>catkin</buildtool_depend>

  <build_depend>rospy</build_depend>
  <build_depend>xacro</build_depend>
  <build_depend>actionlib</build_depend>
  <build_depend>sensor_msgs</build_depend>
  <build_depend>control_msgs</build_depend>
  <build_depend>trajectory_msgs</build_depend>
  <build_depend>cv_bridge</build_depend>
  <build_depend>dynamic_reconfigure</build_depend>
  <build_depend>baxter_core_msgs</build_depend>
  <build_depend>baxter_interface</build_depend>

  <run_depend>rospy</run_depend>
  <run_depend>xacro</run_depend>
  <run_depend>actionlib</run_depend>
  <run_depend>sensor_msgs</run_depend>
  <run_depend>control_msgs</run_depend>
  <run_depend>trajectory_msgs</run_depend>
  <run_depend>cv_bridge</run_depend>
  <run_depend>dynamic_reconfigure</run_depend>
  <run_depend>baxter_core_msgs</run_depend>
  <run_depend>baxter_interface</run_depend>

</package>
```

This `package.xml` file is used by ROS tools to understand and manage the package during build, installation, and runtime. Along with some metadata about the package, the `package.xml` specifies 11 packages on which `baxter_examples` depends on. The packages with `<build_depend>` are the packages used during the build phase, and the ones with

`<run_depend>` are used during the runtime phase.

The `rospy` dependency is important - `rospy` is the ROS library that Python nodes use to communicate with other nodes in the computation graph. The corresponding library for C++ nodes is `roscpp`.

3.3 File system tools

ROS provides a collection of tools to create, edit, and manage packages. One of the most useful is `rospack`, which returns information about a specific package. In a new terminal, try running the command

```
rospack find baxter_examples
```

which should return the same directory you looked at earlier.

Note: To get info on the options and functionality of many ROS command line utilities, run the utility plus "`help`" (e.g., "`rospack help`").

Next, let's test out a couple more convenient commands for working with packages. Run

```
rosls baxter_examples
```

and then

```
roscd baxter_examples
```

The function of these commands should become apparent quickly. Any ideas what they do?

4 Creating ROS Workspaces and Packages

You're now ready to create your own ROS package.

4.1 Creating a workspace

First, we first have to create a ROS workspace – a directory used to organize and manage ROS packages. Since all ROS code must be contained within a package in a workspace, this is something you'll do each time you start a new lab or final project.

In Section 2.1, you created the `ros_workspaces` directory, which contains the folder `lab1`. As the name suggests, this will be your workspace for lab 1. Next, create a folder `src` in your `/lab1` workspace directory via `mkdir [folder name]`.

ROS uses the `catkin` tool to build all code in a workspace and does some bookkeeping to easily run code in packages. After you fill `/src` with packages, you can build them by running "`catkin_make`" from the workspace directory.

Try running this command now, just to make sure the build system works. You should notice two new directories alongside `src`: `build` and `devel`. ROS uses these directories to store information related to building your packages (in `build`) as well as automatically generated files, like binary executable and header files (in `devel`).

4.2 Creating a new package

You're now ready to create a package. From the `src` directory, run

```
catkin_create_pkg foo
```

Examine the contents of your newly created package `foo` and open its `package.xml` file. By default, you will see that the only dependency created is for the catkin tool itself:

```
<buildtool_depend>catkin</buildtool_depend>
```

Next, we'll try creating a new package while specifying a few dependencies. Return to the `src` directory and run the following command:

```
catkin_create_pkg bar rospy roscpp std_msgs geometry_msgs turtlesim
```

Examine the `package.xml` file for the new package `bar` and verify that the dependencies have been added. You're now ready to add source code, message and service definitions, and other resources to your project.

4.3 Building a package

Once you've added all your resources to the new package, the last step before you can use the package with ROS is to *build* it. Run the `catkin_make` command again from the `lab1` directory.

```
catkin_make
```

`catkin_make` builds all the packages and their dependencies in the correct order. If everything worked, `catkin_make` should print configuration and build information for your new packages with no errors.

Note that the `devel` directory contains the script `setup.bash`, generated by `catkin_make`. "Sourcing" this script will prepare your shell environment for using the ROS packages contained in this workspace (among other actions, it adds "`~/ros_workspaces/lab1/src`" to the `$ROS_PACKAGE_PATH`). Run the commands

```
echo $ROS_PACKAGE_PATH
source devel/setup.bash
echo $ROS_PACKAGE_PATH
```

and note the difference between the output of the first and second `echo`.

Note: *Any time that you want to use a non-built-in package, such as one that you have created, you will need to source the* `devel/setup.bash` *file for that package's workspace.*

Here's what your directory structure should now look like:

```
ros_workspaces
  lab1
    build
    devel
      setup.bash
    src
      CMakeLists.txt
      foo
        CMakeLists.txt
        package.xml
      bar
        CMakeLists.txt
        package.xml
        include
        src
```

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 the contents of your `~/.bashrc` file
- Explain the contents of your `~/ros_workspaces` directory
- Demonstrate the use of the `catkin_make` command
- Explain the contents of a `package.xml` file
- Use ROS's utility functions to find the path of a package

Following Checkpoint 1, you may close all previous terminal windows.

5 Understanding ROS nodes

A quick review of some computation graph concepts:

- **Node**: an executable representing an individual ROS software process
- **Message**: a ROS datatype used to exchange data between nodes
- **Topic**: nodes can *publish* messages to a topic and/or *subscribe* to a topic to receive messages

We're now ready to test out some actual software running on ROS.

5.1 Running roscore

In a terminal window, run the command

```
roscore
```

`roscore` starts the ROS master node, the centralized server for managing nodes, topics, services, communication, and more. It is typically the main entry point and the first for running any ROS system.

Leave `roscore` running and open a second terminal window (`Ctrl+Shift+T` creates a new tab or `Ctrl+Shift+N` creates a new window).

As with packages, ROS provides a collection of tools we can use to get information about the nodes and topics that make up the current computation graph. Try running

```
rosnode list
```

You should see that the only node currently running is `/rosout`, which listens for debugging and error messages published by other nodes and logs them to a file. We can get more information on the `/rosout` node by running

```
rosnode info /rosout
```

whose output shows that `/rosout` publishes the `/rosout_agg` topic, subscribes to the `/rosout` topic, and offers the `/set_logger_level` and `/get_loggers` services.

The `/rosout` node isn't very exciting. Let's look at some other built-in ROS nodes that have more interesting behavior.

5.2 Running turtlesim

To start additional nodes, we use the `rosrun` command. The syntax is

```
rosrun [package_name] [executable_name]
```

The ROS equivalent of a "hello world" program is turtlesim, a simulated environment to interact with a virtual turtle. To run turtlesim, we want to run the `turtlesim_node` executable, which is located in the `turtlesim` package. Open a new terminal window and run

```
rosrun turtlesim turtlesim_node
```

A turtlesim window should appear. Repeat the two `rosnode` commands from section 5.1, and compare the results. You should see a new node called `/turtlesim` that publishes and subscribes to a number of additional topics.

6 Understanding ROS topics

We're now ready to make our turtle do something. Leave the `roscore` and `turtlesim_node` windows open from the previous section.

In yet another new terminal window, use `rosrun` to run the `turtle_teleop_key` executable in the `turtlesim` package:

```
rosrun turtlesim turtle_teleop_key
```

This starts a node to take keyboard inputs to tell turtlesim what to do. You should now be able to drive your turtle around the screen with the arrow keys when in this terminal window.

6.1 Using rqt_graph

Let's take a closer look at what's going on here. We'll use the tool `rqt_graph` to visualize the current computation graph. Open a new terminal window and run

```
rosrun rqt_graph rqt_graph
```

This should produce an illustration like Figure 3. In this example, the `turtle_teleop_key` node is capturing your keystrokes and publishing them as control messages on the `/turtle1/cmd_vel` topic. The `/turtlesim` node then subscribes to this topic to receive the control messages.

Figure 3: Output of rqt_plot when running turtlesim.

6.2 Using rostopic

Let's take a closer look at the `/turtle1/cmd_vel` topic using the rostopic tool. First, we'll use `rostopic echo` to look at individual messages that `/teleop_turtle` is publishing to the topic. Open a new terminal window and run

```
rostopic echo /turtle1/cmd_vel
```

Move the turtle with the arrow keys and observe the messages published on the topic. Return to your `rqt_graph` window, and click the refresh button (blue circular arrow icon in the top left corner). You should now see that a second node (the `rostopic` node) has subscribed to the `/turtle1/cmd_vel` topic, as shown in Figure 4.

While `rqt_graph` only shows topics with at least one publisher and subscriber, we can view all the topics published or subscribed to by any node by running

```
rostopic list
```

For even more information, including the message type used for each topic, we can use the verbose option:

```
rostopic list -v
```

Keep turtlesim running for use in the next section.

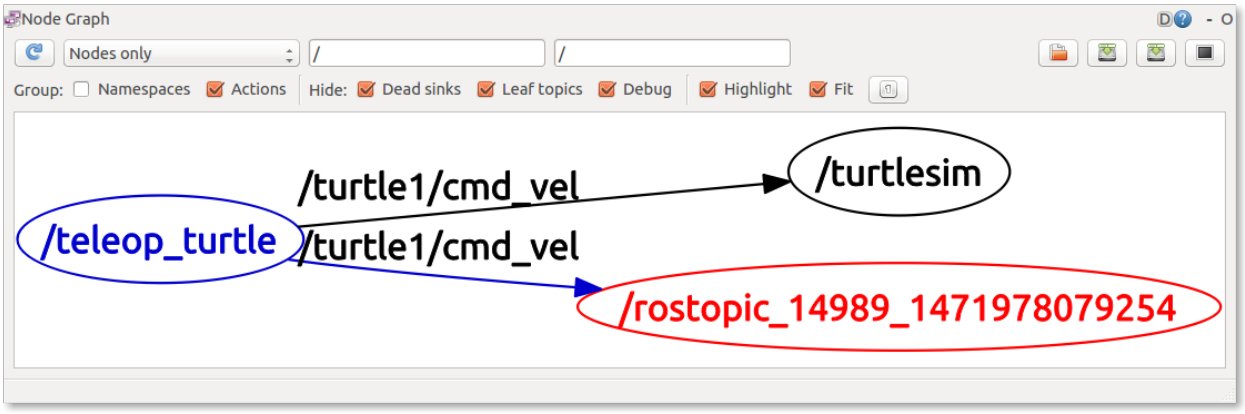

Figure 4: Output of rqt_graph when running turtlesim and viewing a topic using rostopic echo.

7 Understanding ROS services

Services are another method nodes can use to pass data between each other. While topics are typically used to exchange a continuous stream of data, a service allows one node to *request* data from another node, and receive a *response*. Requests and responses are to services as messages are to topics: that is, they are containers of relevant information for their associated service or topic.

7.1 Using rosservice

The `rosservice` tool is analogous to `rostopic`, but for services rather than topics. We can call

```
rosservice list
```

to show all the services offered by currently running nodes.

We can also see what type of data is included in a request/response for a service. Check the service type for the `/clear` service by running

```
rosservice type /clear
```

This tells us that the service is of type `std_srvs/Empty`, which means that the service does not require any data as part of its request, and does not return any data in its response.

7.2 Calling services

Let's try calling the the `/clear` service. While this would usually be done programmatically from inside a node, we can do it manually using the `rosservice call` command. The syntax is

```
rosservice call [service] [arguments]
```

Because the `/clear` service does not take any input data, we can call it without arguments

```
rosservice call /clear
```

If we look back at the `turtlesim` window, we see that our `/clear` call has cleared the background (you can no longer see the path the turtle travelled).

Next, we will call services that require arguments. Use `rosservice type` to find the datatype for the `/spawn` service. The query should return `turtlesim/Spawn`, which tells us that the service is of type `Spawn`, and that this service type is defined in the `turtlesim` package. Use `rospack find` to get the location of the `turtlesim` package, then open the `Spawn.srv` service definition, located in the package's `/srv` subfolder. The file should look like

```
float32 x
float32 y
float32 theta
string name
---
string name
```

The first portion of `Spawn.srv` tells us that the `/spawn` service takes four arguments in its request: three decimal numbers giving the position (`x`, `y`) and orientation (`theta`) of the new turtle, and a single string (`name`) specifying the new turtle's name. The second portion tells us that the service returns one data item: a string with the name we specified in the request.

Call the `/spawn` service to create a new turtle, specifying the values for each of the four arguments (in order):

```
rosservice call /spawn 2.0 2.0 1.2 "newturtle"
```

The service call returns the name of the newly created turtle, and you should see the second turtle appear in the `turtlesim` window.

Checkpoint 2

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 what a **node**, **topic**, and **message** are
- Drive your turtle around the screen using arrow keys
- Use ROS's utility functions to view data on topics and messages

---

8 Lab Checkoffs on Gradescope

Your lab checkoff progress for the semester will be posted to the `Lab Grades` assignment on Gradescope. The autograder for this assignment will typically be re-run daily; if you do not receive credit for a completed lab within a few days, please reach out to lab staff.

9 Pushing to Github

We highly recommend backing up your labs to Github to share code between partners and have access to your lab code outside of class.

The one-time setup instructions to set up git on your instructional machine are borrowed from CS61C.

1. Set your git identity

   ```
   git config --global user.name "your_name"
   git config --global user.email your_email@example.com
   ```

2. Generate a ssh key pair.

   ```
   ssh-keygen -t ed25519 -C "your_email@example.com"
   ```

3. Add key to ssh-agent.

```
eval "$(ssh-agent -s)"
ssh-add ~/.ssh/id_25519
```

4. Print out your public key by running

```
cat ~/.ssh/id_ed25519.pub
```

   The output should look similar to the following (length may differ):

```
ssh-ed25519 AAAAC3NzaC1lZDI1N6jpH3Bnbebi7Xz7wMr20LxZCKi3U8UQTE5AAAAIBTc2HwlbOi8T
your_email@example.com
```

5. In your browser, go to GitHub $\Longrightarrow$ Settings $\Longrightarrow$ SSH and GPG Keys $\Longrightarrow$ New SSH key and add your public key. Set the title to something that helps you remember what device the key is on (e.g. `EECS106A Lab Machine`)

6. Try connecting to GitHub with SSH:

```
ssh -T git@github.com
```

   If all went well, you should see something like:

```
Hi USERNAME! You've successfully authenticated, but GitHub does not provide shell access.
```

After creating a new repository on Github, you can link it to your `ros_workspaces` directory by running

```
git remote add origin git@github.com:{GITHUB_USERNAME}/{YOUR_REPO}.git
```

Make sure that you select SSH when you copy/clone your Github repository URL, NOT your HTTPS link, or you will not be able to commit changes to your repository.

After which, you can push to your repository by running `git push origin`. If you try to `git push` instead of `git push origin`, you will try to push to the class starter code Github repository, which will fail.

Reminder: You may share your repository with your lab partners, but your repository **must be private**. We would like to maintain academic integrity!

10 Additional Lab Etiquette

Because the lab workstations are shared, we sometimes run into strange bugs. The most common is that someone leaves a process running when they leave the lab; when the next person uses the workstation, ROS can get confused by the additional processes running from the prior user. A particularly insidious example is leaving a `roscore` master node running in the background, causing the next person to be unable to run a master node with proper communications.

To avoid issues like this in the lab, please do the following before leaving:

- `Ctrl+C` out of every terminal before closing it. **Do not Ctrl+Z**. While Ctrl+Z may look like it stops your process, it really only pauses it, and the process will continue to run in the background.
- To log out, always use the command `pkill -u $(whoami)`. This will close out every process on your account before logging you out. We want to ensure future lab sections run smoothly!

Congratulations on finishing Lab 1!