Student ID:		**Lab Section:**	
Name:		**Lab Group:**	

Software Lab 1

Introduction to Robot Operating System (ROS)

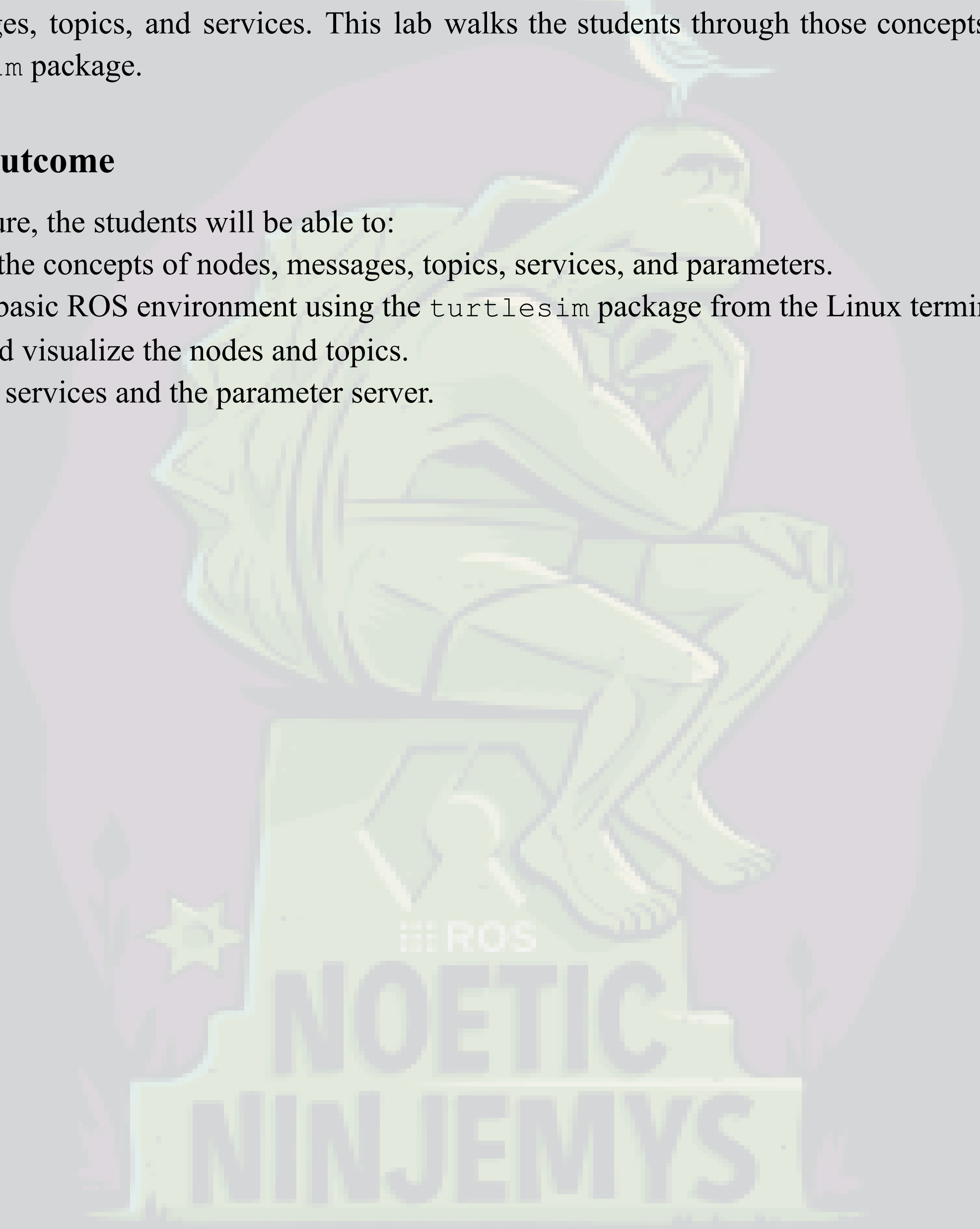
Topic Overview

This lab aims to acquaint students with the fundamentals of Robot Operating System (ROS) using ROS Noetic Ninjemys distribution on Ubuntu 20.04 LTS (Focal Fossa). It covers the concepts of nodes, messages, topics, and services. This lab walks the students through those concepts utilizing the `turtlesim` package.

Learning Outcome

After this lecture, the students will be able to:

a. Know the concepts of nodes, messages, topics, services, and parameters.
b. Run a basic ROS environment using the `turtlesim` package from the Linux terminal.
c. List and visualize the nodes and topics.
d. Utilize services and the parameter server.



Prepared by Shayekh Bin Islam and Mir Sayeed Mohammad

What is Robot Operating System (ROS)?

- An *open-source*, *meta-operating system* for your robot. (A meta operating system is built on top of the operating system and allows different processes (nodes) to communicate with each other at runtime.)
- 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 (Plumbing).
- Provides *tools* and *libraries* for obtaining, building, writing, and running code across multiple computers.

Overview of Graph Concepts

The fundamental concepts of the ROS implementation are **nodes**, **messages**, **topics**, and **services**.

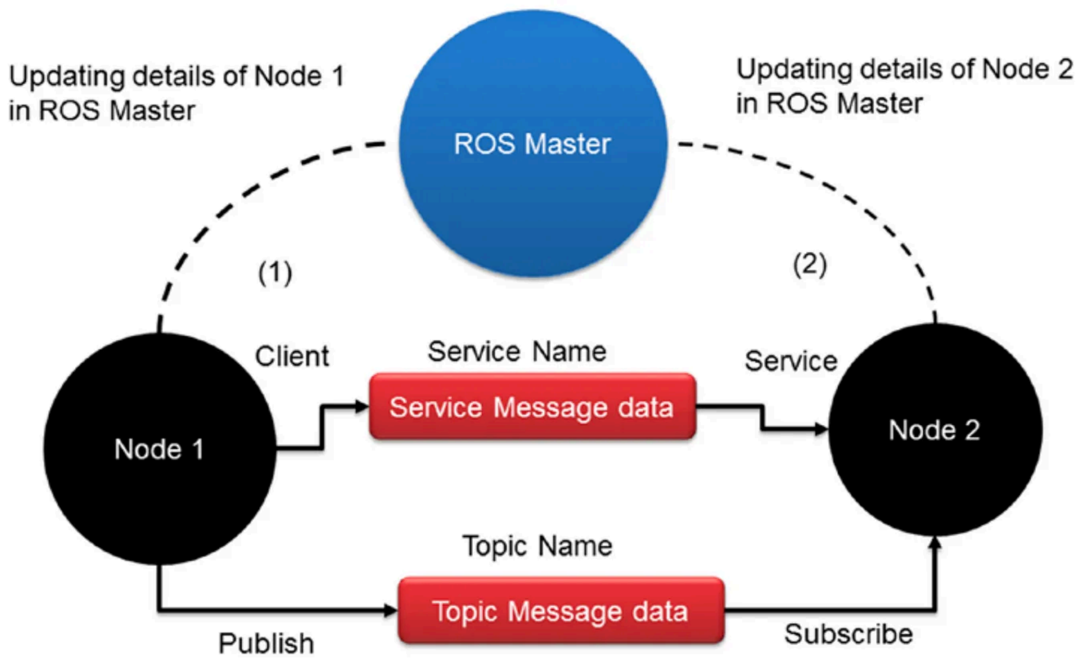


Figure: ROS Nodes, Topics, and Services

A **node** is an executable that uses ROS to communicate with other nodes. **Messages** are ROS data types consisting of typed fields (integer, floating point, boolean, array, etc.) used when *subscribing* to or *publishing* a **topic**. Nodes can *publish* messages to a **topic** and *subscribe* to a topic to receive **messages**.

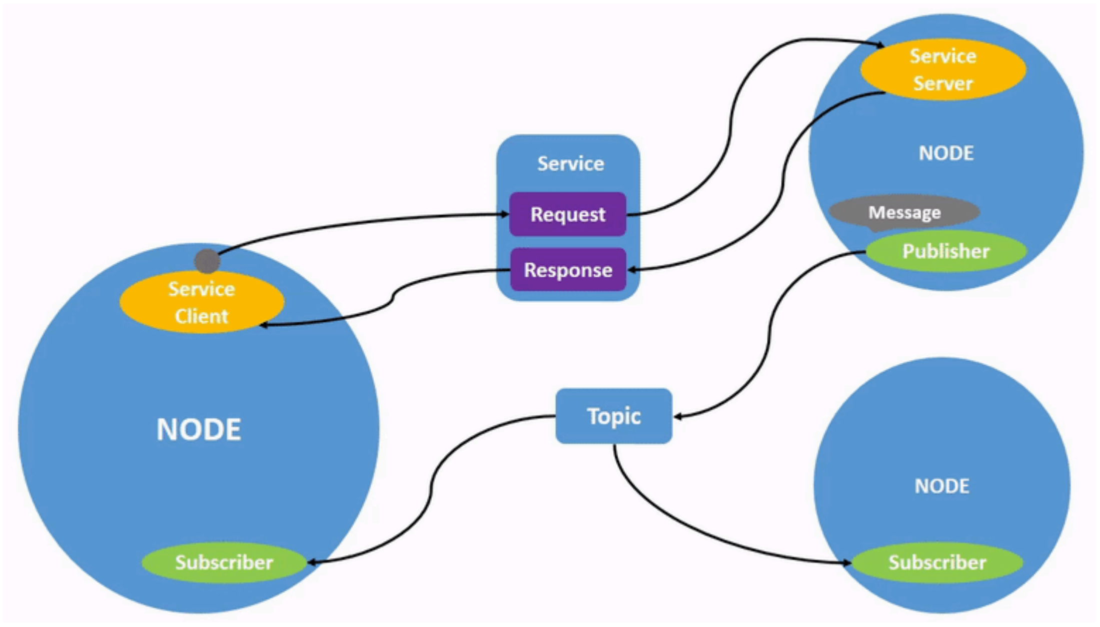


Figure: ROS Topics and Services

The **Master** node provides naming and registration services to the rest of the nodes in the ROS system. It *tracks* publishers and subscribers to topics as well as services. The role of the Master is to enable individual ROS nodes to *locate* one another. Once these nodes have located each other they *communicate* with each other peer-to-peer. The Master also provides the **Parameter Server**.

The **services** are a type of request/response communication between ROS nodes. One node will send a request and *wait until it gets a response from the other* (that is services are **synchronous** vs topics are **asynchronous**). The request/response communication is also using the ROS **message** description.

A **parameter server** is a shared, multi-variate dictionary that is accessible via network APIs. Nodes use this server to store and retrieve parameters at runtime. It is meant to be globally viewable so that tools can easily inspect the configuration state of the system and modify it if necessary.

How Things Work Together in ROS

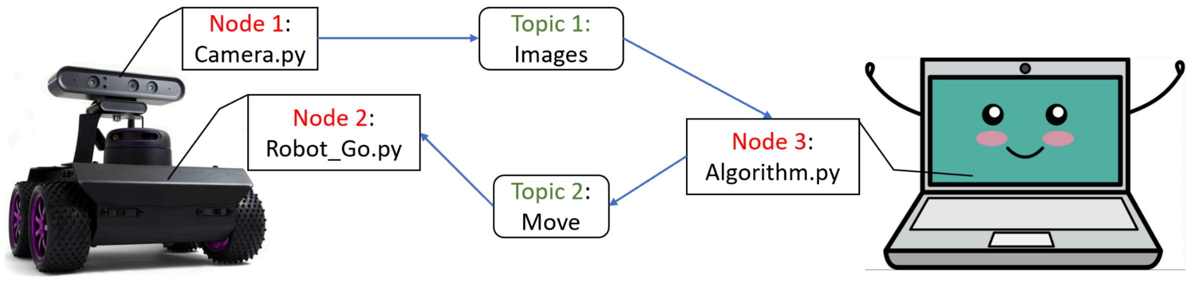


Figure: A brief overview of how ROS really works in a robot

ROS, in short, is a system that ensures that all the different mechanical parts, data, and algorithms can work together to accomplish the desired task. In a simple robot, nodes can be thought of as independent Python scripts or C++ executables that complete a specific task.

For example, in the robot from the figure, we have a node called `Camera.py` which operates the physical camera on the robot. After image acquisition, it will send out the image data in the form of a topic. The topic becomes immediately available to all other nodes connected to the Master, and any node can use the message contained in the topic.

Next, there is a node called `Algorithm.py` which can take a look at the image data, as well as other necessary information such as robot position, orientation, velocity, etc. The algorithm node will then compute the required movement for the desired goal. Finally, it will broadcast another topic called "move" which is then read by the `Robot_Go.py` node. This node communicates with the robot actuator hardware to enable movement.

Walk through of the Turtle Simulator in ROS

ROS Master Node

`roscore` is the first thing you should run in the terminal when using ROS. It will start the master node.

```
ubuntu@ubuntu2004:~$ roscore
```

You will see something similar to:

```
... logging to /home/ubuntu/.ros/log/a/roslaunch-ubuntu2004-2166.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.
started roslaunch server http://ubuntu2004:39575/
ros_comm version 1.16.0
SUMMARY
========
PARAMETERS
 * /rosdistro: noetic
 * /rosversion: 1.16.0
NODES
auto-starting new master
process[master]: started with pid [2176]
ROS_MASTER_URI=http://ubuntu2004:11311/

setting /run_id to ae447852-e1d9-11ee-906d-c7aab3362b1f
process[rosout-1]: started with pid [2186]
started core service [/rosout]
```

ROS Node

Open up a **new terminal**, and let's use **rosnode** to see what running **roscore** did... Bear in mind to keep the previous terminal open either by opening a new tab or simply minimizing it.

The **`rosnode list`** command lists these active nodes.

```
ubuntu@ubuntu2004:~$ rosnode list
/rosout
```

This showed us that there is only one node running: **rosout**. This is always running as it collects and logs nodes' debugging output.
The `rosnode info` command returns information about a specific node.

```
ubuntu@ubuntu2004:~$ rosnode info /rosout
--------------------------------------------------------------------------
Node [/rosout]
Publications:
 * /rosout_agg [rosgraph_msgs/Log]

Subscriptions:
 * /rosout [unknown type]
Services:
 * /rosout/get_loggers
 * /rosout/set_logger_level
contacting node http://ubuntu2004:34877/ ...
Pid: 2599
```

This gave us more information about **rosout**, such as the fact that it publishes `/rosout_agg` which is an aggregated feed for subscribing to console logging messages.

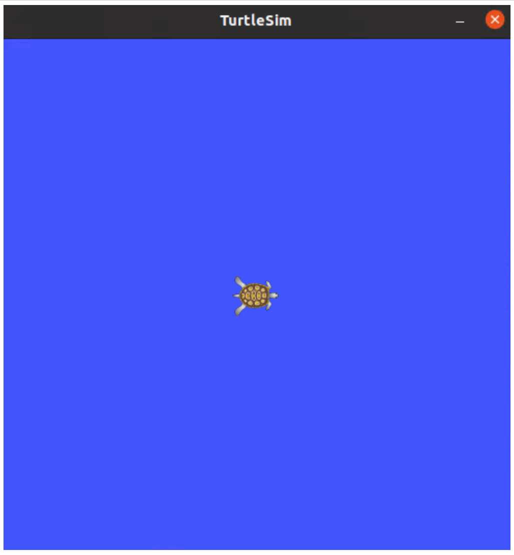
ROSRUN: Run a Node from a **Package**

`rosrun` allows you to use the package name to directly run a node within a package (without having to know the package path).

```
$ rosrun [package_name] [node_name]
```

For example, we can run the `turtlesim_node` in the **`turtlesim`** package. A graphical window will pop up.

```
ubuntu@ubuntu2004:~$ rosrun turtlesim turtlesim_node
[ INFO] [1710405578.824199742]: Starting turtlesim with node name /turtlesim
[ INFO] [1710405578.828309505]: Spawning turtle [turtle1] at x=[5.544445], y=[5.544445],
theta=[0.000000]
```



Now, the `rosnode list` command lists the turtlesim node as an active node.

```
ubuntu@ubuntu2004:~$ rosnode list
/rosout
/turtlesim
```

To test the node, we can run `rosnode ping`.

```
ubuntu@ubuntu2004:~$ rosnode ping turtlesim
rosnode: node is [/turtlesim]
pinging /turtlesim with a timeout of 3.0s
xmlrpc reply from http://ubuntu2004:44801/ time=0.397682ms
xmlrpc reply from http://ubuntu2004:44801/ time=1.785994ms
xmlrpc reply from http://ubuntu2004:44801/ time=1.433372ms
xmlrpc reply from http://ubuntu2004:44801/ time=1.536846ms
xmlrpc reply from http://ubuntu2004:44801/ time=1.388073ms
```

ROS Topics

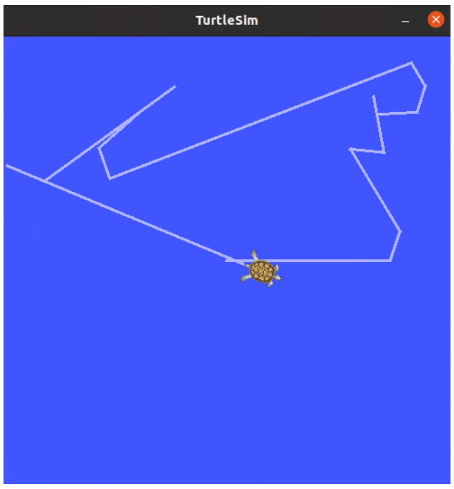
We can see that `turtlesim` subscribes to the topic `/turtle1/cmd_vel`:

```
ubuntu@ubuntu2004:~$ rosnode info turtlesim
--------------------------------------------------------------------------------
Node [/turtlesim]
Publications:
 * /rosout [rosgraph_msgs/Log]
 * /turtle1/color_sensor [turtlesim/Color]
 * /turtle1/pose [turtlesim/Pose]
Subscriptions:
 * /turtle1/cmd_vel [unknown type]
Services:
 * /clear
 * /kill
 * /reset
 * /spawn
 * /turtle1/set_pen
 * /turtle1/teleport_absolute
 * /turtle1/teleport_relative
 * /turtlesim/get_loggers
 * /turtlesim/set_logger_level
contacting node http://ubuntu2004:44801/ ...
Pid: 2801
Connections:
 * topic: /rosout
    * to: /rosout
    * direction: outbound (38163 - 127.0.0.1:51666) [26]
    * transport: TCPROS
```

turtle keyboard teleoperation

We'll also need something to drive the turtle around with.

```
ubuntu@ubuntu2004:~$ rosrun turtlesim turtle_teleop_key
Reading from keyboard
---------------------------
Use arrow keys to move the turtle. 'q' to quit.
```



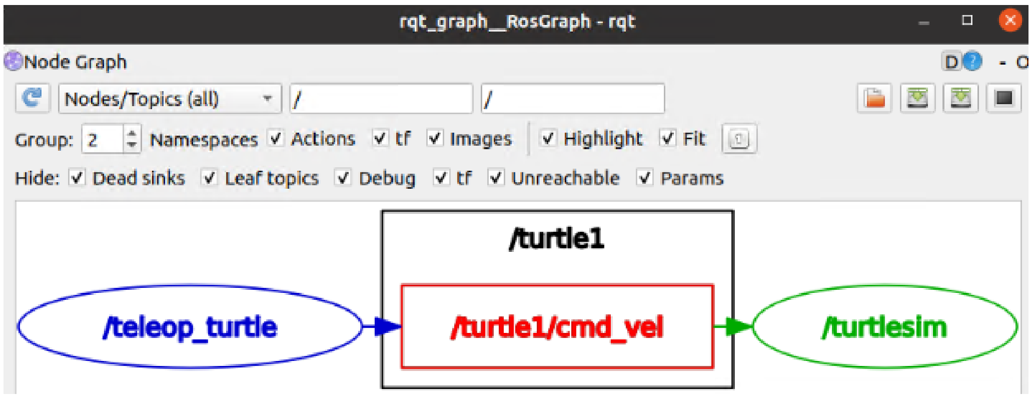
Now you can use the arrow keys of the keyboard to drive the turtle around. If you can not drive the turtle **select the terminal window of the turtle_teleop_key** to make sure that the keys that you type are recorded.

The `turtlesim_node` and the `turtle_teleop_key` node are communicating with each other over a ROS **Topic**. `turtle_teleop_key` publishes the keystrokes on a topic, while turtlesim **subscribes** to the same topic to receive the keystrokes. Let's use **rqt_graph** which shows the nodes and topics currently running.

```
ubuntu@ubuntu2004:~$ sudo apt-get install ros-noetic-rqt
ubuntu@ubuntu2004:~$ sudo apt-get install ros-noetic-rqt-common-plugins

ubuntu@ubuntu2004:~$ rosrun rqt_graph rqt_graph
```

rqt_graph creates a dynamic graph of what's going on in the system. **rqt_graph** is part of the **rqt** package.



If you place your mouse over **/turtle1/cmd_vel** it will highlight the ROS **nodes** (here blue and green) and **topics** (here red). As you can see, the **turtlesim_node** and the **turtle_teleop_key** nodes are communicating on the topic named **turtle1/cmd_vel.**

rostopic

The `rostopic` tool allows you to get information about ROS topics.

You can use the help option to get the available sub-commands for rostopic.

```
ubuntu@ubuntu2004:~$ rostopic -h
rostopic bw     display bandwidth used by topic
rostopic echo   print messages to screen
rostopic hz     display publishing rate of topic
rostopic list   print information about active topics
rostopic pub    publish data to topic
rostopic type   print topic type
```

`rostopic echo` shows the data published on a topic. You probably won't see anything happen because no data is being published on the topic. Let's make *turtle_teleop_key* publish data by pressing the arrow keys. **Remember if the turtle isn't moving you need to select the *turtle_teleop_key* terminal again.**

```
ubuntu@ubuntu2004:~$ rostopic echo /turtle1/cmd_vel
---
linear:
  x: -2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
```

ROS Messages

Communication on topics happens by sending ROS **messages** between nodes. For the publisher (*turtle_teleop_key*) and subscriber (*turtlesim_node*) to communicate, the publisher and subscriber must send and receive the same **type** of message. This means that a topic **type** is defined by the message **type** published on it. The **type** of the message sent on a topic can be determined using rostopic type.

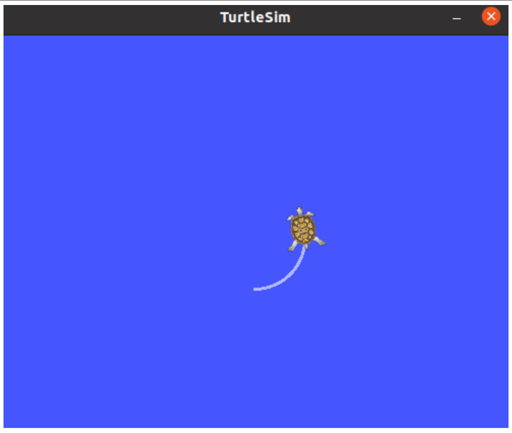
```
ubuntu@ubuntu2004:~$ rostopic type /turtle1/cmd_vel
geometry_msgs/Twist
```

We can look at the details of the message using `rosmsg`:

```
ubuntu@ubuntu2004:~$ rosmsg show geometry_msgs/Twist
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

`rostopic pub` publishes data on to a topic currently advertised.

```
ubuntu@ubuntu2004:~$ rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist --
'[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'
---------------------------------------------------------------------------
publishing and latching message for 3.0 seconds
```

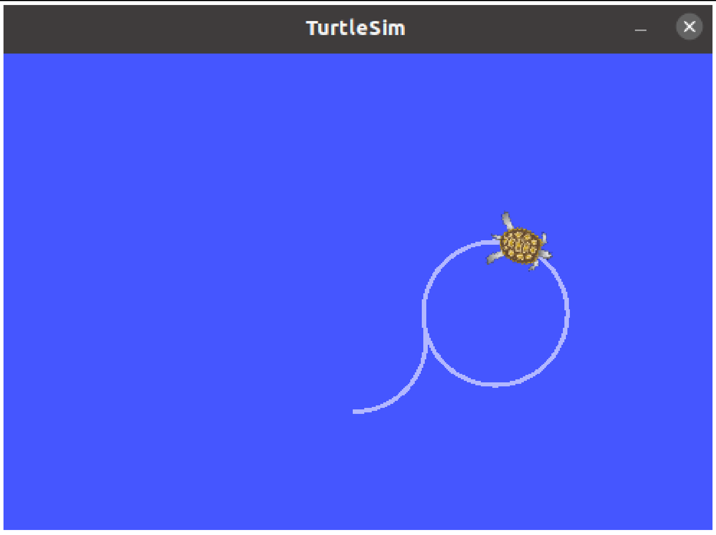


The option dash-one causes rostopic only to publish one message then exit. The option double-dash tells the option parser that none of the following arguments is an option. A geometry_msgs/Twist msg has two vectors of three floating point elements each: linear and angular.

In this case, `'[2.0, 0.0, 0.0]'` becomes the linear value with x=2.0, y=0.0, and z=0.0, and `'[0.0, 0.0, 1.8]'` is the angular value with x=0.0, y=0.0, and z=1.8.

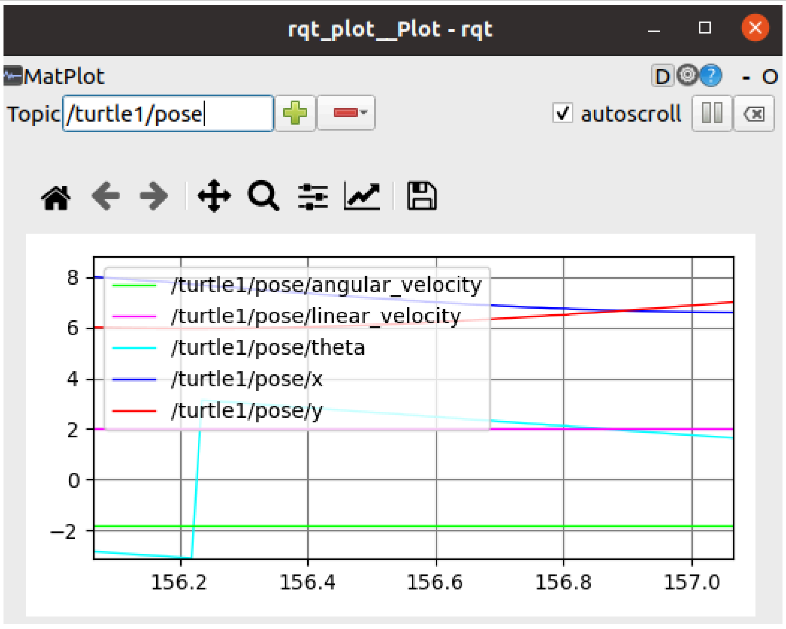
You may have noticed that the turtle has stopped moving; this is because the turtle requires a steady stream of commands at 1 Hz to keep moving. We can publish a steady stream of commands using the `rostopic pub -r` command:

```
ubuntu@ubuntu2004:~$ rostopic pub /turtle1/cmd_vel geometry_msgs/Twist
-r 1 -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, -1.8]'
```



`rqt_plot` displays a scrolling time plot of the data published on topics. Here we'll use `rqt_plot` to plot the data being published on the `/turtle1/pose` topic. A text box in the upper left corner gives you the ability to add any topic to the plot.

```
ubuntu@ubuntu2004:~$ rosrun rqt_plot rqt_plot
```



ROS Services

Services are another way that nodes can communicate with each other. Services allow nodes to send a **request** and receive a **response**. `rosservice` can easily attach to ROS's client/service framework with services.

```
ubuntu@ubuntu2004:~$ rosservice list
/clear
/kill
/reset
/rosout/get_loggers
/rosout/set_logger_level
/spawn
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/get_loggers
/turtlesim/set_logger_level
```

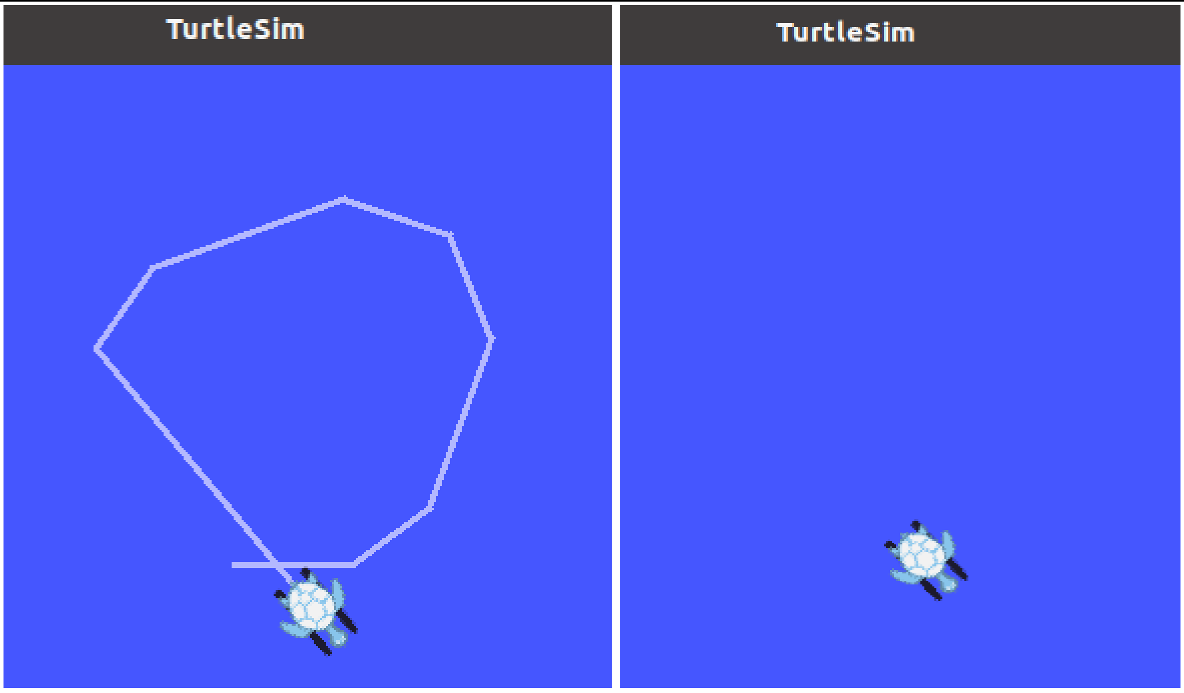
The list command shows us that the *turtlesim* node provides nine services: `reset`, `clear`, `spawn`, `kill`, `turtle1/set_pen`, `/turtle1/teleport_absolute`, `/turtle1/teleport_relative`, `turtlesim/get_loggers`, and `turtlesim/set_logger_level`. There are also two services related to the separate *rosout* node: `/rosout/get_loggers` and `/rosout/set_logger_level`.

Let's find out what type the `clear` service is:

```
ubuntu@ubuntu2004:~$ rosservice type /clear
std_srvs/Empty
```

This service is empty, which means when the service call is made it takes no arguments (i.e. it sends no data when making a **request** and receives no data when receiving a **response**). Let's call this service using `rosservice call`:

```
ubuntu@ubuntu2004:~$ rosservice call /clear
```



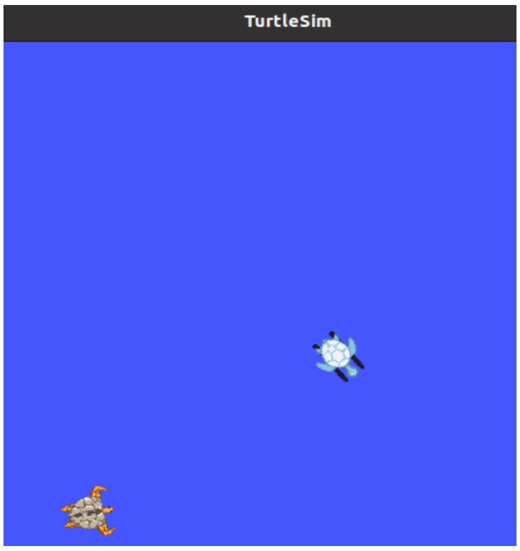
This does what we expect, it clears the background of the `turtlesim_node`.

Let's look at the case where the service has **arguments** by looking at the information for the service spawn:

```
ubuntu@ubuntu2004:~$ rosservice type /spawn | rossrv show
float32 x
float32 y
float32 theta
string name
---
string name
```

This service lets us spawn a new turtle at a given location and orientation. The name field is optional, so let's not give our new turtle a name and let turtlesim create one for us.

```
ubuntu@ubuntu2004:~$ rosservice call /spawn 2 2 0.2 ""
name: "turtle2"
```



rosparam

`rosparam` allows you to store and manipulate data on the ROS **Parameter Server**. The Parameter Server can store integers, floats, boolean, dictionaries, and lists. `rosparam` uses the YAML markup language for syntax. In simple cases, YAML looks very natural: 1 is an integer, 1.0 is a float, one is a string, true is a boolean, [1, 2, 3] is a list of integers, and {a: b, c: d} is a dictionary.

```
ubuntu@ubuntu2004:~$ rosparam list
/rosdistro
/roslaunch/uris/host_ubuntu2004__38275
/rosversion
/run_id
/turtlesim/background_b
/turtlesim/background_g
/turtlesim/background_r
```

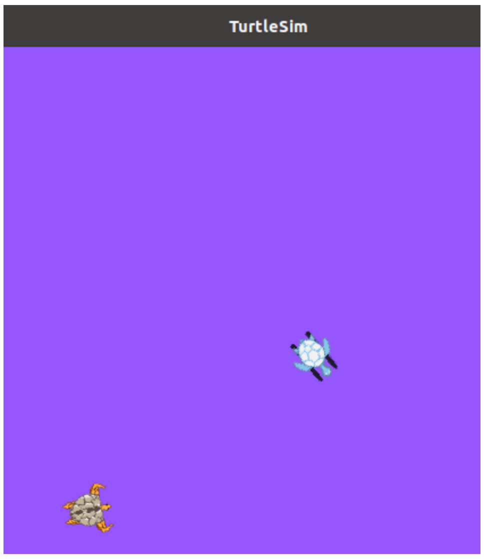
Here we can see that the turtlesim node has three parameters on the param server for background color.

Let's get the value of the green background channel:

```
ubuntu@ubuntu2004:~$ rosparam get /turtlesim/background_g
86
```

Here, we will change the red channel of the background color. This changes the parameter value, then we have to call the `clear` service *for the parameter change to take effect.*

```
ubuntu@ubuntu2004:~$ rosparam set /turtlesim/background_r 150
ubuntu@ubuntu2004:~$ rosservice call /clear
```



Resources

1. https://app.theconstruct.ai/Desktop Free virtual machine online hosted by The Construct. Very useful for running experiments out of the box without the hassle of installing ROS on your own system. They also have an extensive list of tutorials and courses on ROS. https://app.theconstruct.ai/courses/
2. ROS official tutorials: https://wiki.ros.org/ROS/Tutorials