



---

# ROS TUTORIAL

---



**ARCHANA RAMALINGAM**

**SJSU ID: 010761114**

**CMPE 220 - SYSTEM SOFTWARE**

## Contents

1. What is ROS?.....	2
2. Why ROS? .....	2
3. Supported platforms .....	3
3.1. Robot hardware platforms.....	3
3.2. Robotic sensors .....	3
4. ROS Structure .....	3
5. ROS Installation.....	4
6. Understanding ROS workspace.....	4
6.1. Overlays .....	6
6.2. Setting up the environnement .....	6
7. Creating a ROS workspace.....	7
7.1. Basic ROS commands.....	7
7.2. Creating a catkin package.....	8
7.3. Building ROS packages .....	10
8. Handling ROS nodes .....	11
9. Understanding ROS topics.....	14
10. Project specific details.....	17
11. Pros and cons.....	17
12. References.....	18

## 1. What is ROS?

**ROS (Robot operating system)** is an open source software to control robotic components from a PC. It is used for both research and commercial purposes. It acts as a framework to write robotic applications by providing the necessary tools and libraries to software developers. It provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more. ROS is licensed under an open source, BSD license. It was developed by Stanford Artificial Intelligence Laboratory and Willow Garage Robotics Research Lab and released in 2007.

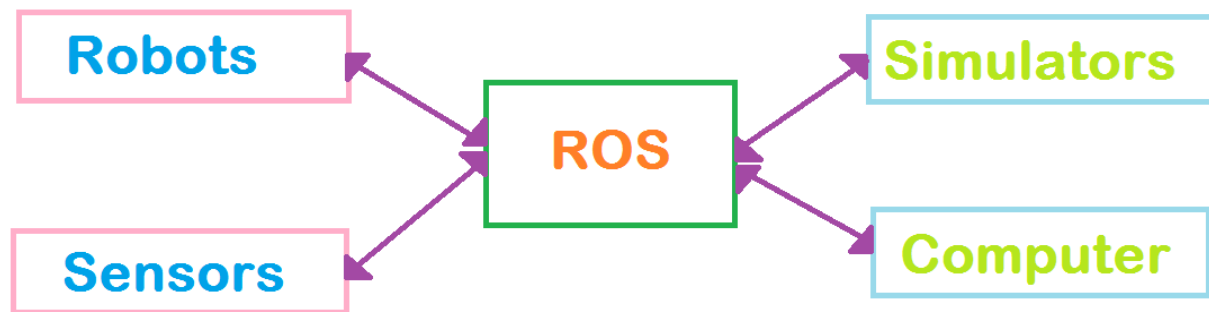


Figure 1 ROS Layout

## 2. Why ROS?

ROS is a powerful software being used widely by robots of all shapes and sizes. Being open source, ROS is constantly growing with the extensive support from the solution-focused community of users from around the world. Most of the available ROS packages have been contributed by either academic institution like Stanford, Carnegie Mellon University, ETH Zurich or commercial institution like BMW, Bosch. The goal of ROS is to promote open-source collaborative robotics software development.

In order to widen the usage, ROS has also released a **ROS Industrial** version to provide a robust and reliable software that meets the industry standards. It also provides vendor-specific packages.

ROS provides documentation in various languages too ([ROS wiki](#)). It supports both C++ and Python.

### 3. Supported platforms

ROS packages support a wide field of

#### 3.1. Robot hardware platforms: [ROS Robots](#)

4-wheel robots, robotic arms and hands, mobile manipulators, humanoid robots, micro aerial vehicles (MAVs) like quadcopters and drones, Unmanned Aerial Vehicle (UAV), autonomous underwater vehicle (AUV), self-driving vehicles.

#### 3.2. Robotic sensors: [ROS Sensors](#)

2D/3D sensors, touch sensors, audio/speech recognition, cameras, motion capture, RFID, Arduino sensor, pose estimation (GPS).

### 4. ROS Structure

In order to work with ROS, the basic concepts should be understood well by the user. We will first see the ROS structure, its components and how it is built internally. ROS is a very powerful networking software too and hence it treats all sensors and hardware components connected to it as a node. Hence ROS architecture is a network of nodes. The ROS structure consists:

- Stack: collection of packages
- Package: basic structure of ROS
- Node: an executable that does all computation and connects to other nodes using ROS topics. It has to be either a publisher (sending data) or a subscriber (receiving data).
- Topic: Used by nodes to communicate by either publishing messages to a topic or subscribing to a topic to receive messages. Any node can publish and only selected nodes can subscribe.
- Message: ROS datatype to publish/subscribe by nodes.
- Header: Messages are numbered with headers to identify sending and receiving nodes.
- Manifest: It contains package information and generated with package.xml file. Each package must have a package.xml file. It contains 2 basic tags: <depend> tells which packages the current package is dependent on. <export> shows which packages get data from the current package.
- Service: Nodes can provide or use a service
- Bag: It is a format or tool to store data from nodes (like sensor data)
- Parameter server: used to configure nodes
- Master: It is a main node that provides naming, registration services and Parameter server to the other nodes. It tracks publishers and subscribers to topics and services. It basically enables nodes to locate each other after which communication is purely peer-to-peer. Master has to be active for ROS to work.

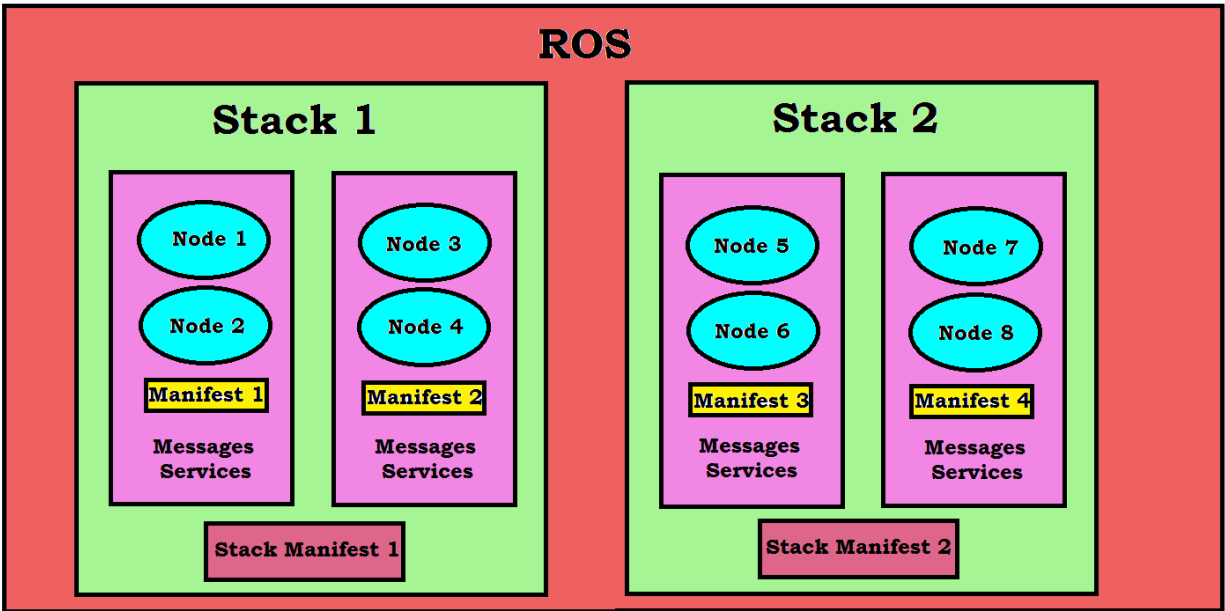


Figure 2 ROS Structure

## 5. ROS Installation

To install ROS, follow the instructions given in the link - [ROS Installation](#). It runs on a Linux OS (Ubuntu and Debian).

Once ROS is installed and the environment variables are set, we can start by creating a ROS workspace, which is covered in the next section.

### Note:

1. All ROS distributions are named alphabetically starting from 'A', with the latest being **ROS Kinetic Kame**. This is the distribution used for the project as well as this tutorial. To create and manage multiple packages in a single place (workspace) we use catkin. For ROS Groovy and later use catkin and for ROS Fuerte and later use rosbuilt.
2. Stacks concept has been removed with catkin to simplify the code base. We can use metapackages to collect multiple packages in one place. Also a single workspace can have multiple packages too.

## 6. Understanding ROS workspace

In order to work with ROS source code, it is convenient to do so in a 'workspace'. For the coming tutorials, new ROS stacks and packages will be created in this space. We use catkin workspace to build, modify and install catkin packages. It contains upto 4 different spaces that serve a unique role in the software development process. They are Source space, Build space, Development space (Devel), Install space and Result space. A typical catkin workspace layout looks like:

```

workspace_folder/      -- WORKSPACE
  src/                  -- SOURCE SPACE
    CMakeLists.txt     -- The 'toplevel' CMake file
    package_1/
      CMakeLists.txt
      package.xml
      ...
    package_n/
      CMakeLists.txt
      package.xml
      ...
  build/                -- BUILD SPACE
    CATKIN_IGNORE      -- Keeps catkin from walking this directory
  devel/                -- DEVELOPMENT SPACE (set by CATKIN_DEVEL_PREFIX)
    bin/
    etc/
    include/
    lib/
    share/
    .catkin
    env.bash
    setup.bash
    setup.sh
    ...
  install/              -- INSTALL SPACE (set by CMAKE_INSTALL_PREFIX)
    bin/
    etc/
    include/
    lib/
    share/
    .catkin
    env.bash
    setup.bash
    setup.sh

```

...

1. Source space:  
It contains the source code of the catkin packages, which we can extract or checkout or clone for any packages we intend to build. Each folder has one or more catkin packages. It should remain unaffected by configuring, building or installing. The source space's root is linked to 'toplevel' CMakeLists.txt file, which is invoked by CMake during the workspace's ROS projects configuration.
2. Build space:  
The catkin packages are built in this space when CMake is invoked. CMake and catkin's cache information and other intermediate files are kept here too. It is recommended to be placed outside source space.
3. Development space (Devel):  
After build stage, the built targets are placed here before being installed. The targets are organized the same manner they were installed. Hence, this space provides a good testing and development environment, without invoking installation. It is better to set this space as a peer of build space (both are placed at the same level same as source space).
4. Install space:  
The built targets are installed into the install space. It does not have to be inside the workspace, as it is already set by CMAKE\_INSTALL\_PREFIX.
5. Result space:  
It is used to refer to the Development space or Install space as a generic term.

To build catkin packages use cmake workflow in the build space directory:

1. Invoke 'cmake <path/to/source space>'
2. 'make'
3. 'make install'

### 6.1. Overlays

Every ROS distribution has a new feature in it. Most users install the available distribution and then build their own packages against the distribution or mix and match with various other distribution packages. ROS has a concept of overlays, which allows the build system to traverse multiple package installations in order to find dependencies.

### 6.2. Setting up the environment

The shell environment needs to be extended, so that ROS packages can be found and used. The catkin package contains a set of environment setup files that has to be sourced using :

```
source /opt/ros/<distribution_name>/setup.bash
```

These files are arranged in the following manner: (Shown example is for ROS Kinetic on Ubuntu)

```
/
opt/
  ros/
    groovy/
      setup.bash -- Environment setup file for Bash shell
      setup.sh   -- Environment setup file for Bourne shell
      setup.zsh  -- Environment setup file for zshell
      ...
```

## 7. Creating a ROS workspace

A catkin workspace is created using the following commands

```
$ mkdir -p ~/catkin_ws/src // A catkin workspace folder and source space
                             folder are created

$ cd ~/catkin_ws/src

$ catkin_init_workspace // This command creates the 'toplevel'
                          CMakeLists.txt file inside source folder
```

Now we can try to 'build' the workspace, even though no packages are present.

```
$ cd ~/catkin_ws/ // Always move to main workspace folder before building
$ catkin_make      // Builds the workspace
```

Now the workspace has 2 more folders: build and devel. Devel contains several setup.\*sh files. As mentioned earlier, we must source the environment path file so that this workspace is overlayed on top of the environment. This is done with this command:

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

To check if the current workspace is overlayed properly by setup script, we can use the following command to view ROS\_PACKAGE\_PATH environment variable includes current directory.

```
$ echo $ROS_PACKAGE_PATH // Command to display the variable contents
/home/youruser/catkin_ws/src:/opt/ros/kinetic/share:/opt/ros/kinetic/stacks
```

### 7.1. Basic ROS commands

There are some basic ROS commands which makes navigating the system easy.



- **rospack**: used to get information about a package using many options. Eg: using 'find' option returns package's path. (`$rospack find [package_name]`)
- **roscd**: similar to 'cd' command in Linux, except that it can directly jump to any location (folder) inside the system and does not have to follow hierarchy like 'cd' does. (`$ roscd [locationname[/subdir]]`)
- **pwd**: Prints working directory path. (`$pwd`)
- **roscd log**: takes the user to the folder where ROS stores log files. If no ROS programs are present, then this command yields an error that it does not exist. (`$roscd log`)
- **rosls**: similar to 'ls' Linux command. (`($ rosls [locationname[/subdir]])`)

**Tab completion:** ROS supports tab completion, where we just have to type part of the folder/file name and it autocompletes it when a tab is pressed. Pressing tab twice displays all ROS packages that share the same name.

## 7.2. Creating a catkin package

A package created qualifies as a catkin package only if it meets a few requirements:

1. Must contain catkin compliant package.xml manifest file containing meta information about the package.
2. Must contain CMakeLists.txt file
3. Each folder must not contain more than one package. (no nested/multiple packages)

Simplest package structure should look like:

```
my_package/
  CMakeLists.txt
  package.xml
```

**catkin\_create\_pkg**: used to create a package. Might optionally include all dependencies of a package.

```
$catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
```

Eg:

```
$ cd ~/catkin_ws/src
```

```
$catkin_create_pkg beginner_tutorials std_msgs rospy roscpp
```

This will create a package with package.xml and CMakeLists.txt file, which is partially filled by the information provided in the command.

Now we can use the following command to know the dependencies of the current package:

```
$ rospack depends1 beginner_tutorials // to display first order dependencies
```

will display

```
std_msgs
rospy
roscpp
```

This information is also present in the package.xml file, which can be checked by following commands:

```
$ roscd beginner_tutorials
$ cat package.xml // to open and view the xml file
```

will display:

```
<package>
...
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>
...
</package>
```

Sometimes dependencies might have its own dependencies too. It can be checked using the same rospack command as follows:

```
$ rospack depends1 rospy
```

```
genpy
rosgraph
rosgraph_msgs
roslib
std_msgs
```

rospack with **depends** option displays all dependencies recursively.

```
$ rospack depends beginner_tutorials
cpp_common
rostime
roscpp_traits
roscpp_serialization
```

```
genmsg
genpy
message_runtime
roscconsole
std_msgs
rosgraph_msgs
xmlrpcpp
roscpp
rosgraph
catkin
rospack
roslib
rospy
```

Package.xml can be customized by the user. There are several tags that needs to be filled up.

- **Description tag:** contains the short user description of the package  
`<description>The beginner_tutorials package</description>`
- **Maintainer tag:** a name and email is required to be filled in here in order to be able to contact the maintainer. Multiple maintainers can exist for a package.  
`<maintainer email="you@yourdomain.tld">Your Name</maintainer>`
- **License tag:** Choose and fill in an open source license like BSD, GPLv3, etc.  
`<license>BSD</license>`
- **Dependencies tag:** lists dependencies of the package. All dependencies are available at build time and hence are added to `build_depend` by default. If we require a dependency to be available at run time, then we can add it under the `run_depend` tag as follows:  
`<build_depend>roscpp</build_depend>`  
`<run_depend>roscpp</run_depend>`

### 7.3. Building ROS packages

**catkin\_make:** used to build a package. Usually run from the main workspace folder. Combines `cmake` and `make` in the CMake workflow.

```
$ catkin_make [make_targets] [-DCMAKE_VARIABLES=...]
```

For example, we can run the following to build our packages in our workspace. This builds all (zero to multiple) packages.

```
$ catkin_make
```

```
$ catkin_make install # (optionally)
```

If the source code is present at a different location (say my\_src), then run the following commands:

```
$ catkin_make --source my_src
```

```
$ catkin_make install --source my_src # (optionally)
```

Now add the workspace to the ROS environment by using the following command:

```
$ . ~/catkin_ws/devel/setup.bash
```

Try running catkin\_make command in catkin\_ws folder to build the recently created beginner\_tutorials package. We can do an rosrun or ls to see the newly created: build and devel folders.

Note: This tutorial uses certain examples from already created packages, which needs to be installed: (Replace <distro> with distribution name as in 'kinetic')

```
$sudo apt-get install ros-<distro>-ros-tutorials
```

## 8. Handling ROS nodes

ROS nodes use a ROS client library to communicate with each other. The library allows nodes written in different programming languages to communicate easily:

- rospy = python client library
- roscpp = c++ client library

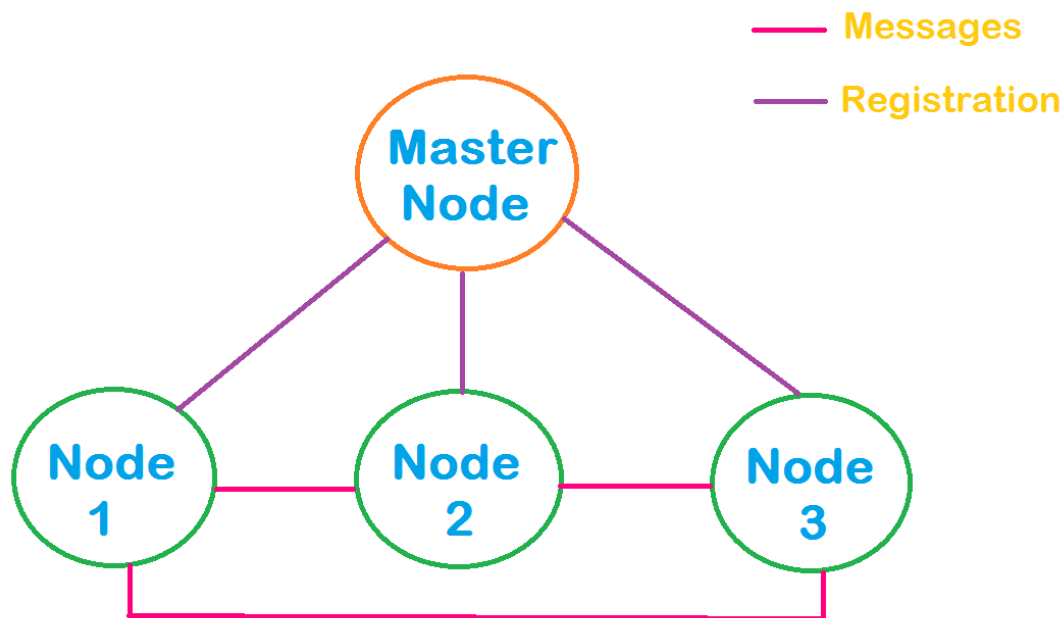


Figure 3 ROS Node structure

**rosout** is the stdout/stderr equivalent of ROS.

**roscore** is basically a combination of Master + rsout + parameter server. It is a collection of nodes and programs that are compulsory pre-requisites of a ROS system. In order to have the nodes communicating, roscore must be running. It starts up:

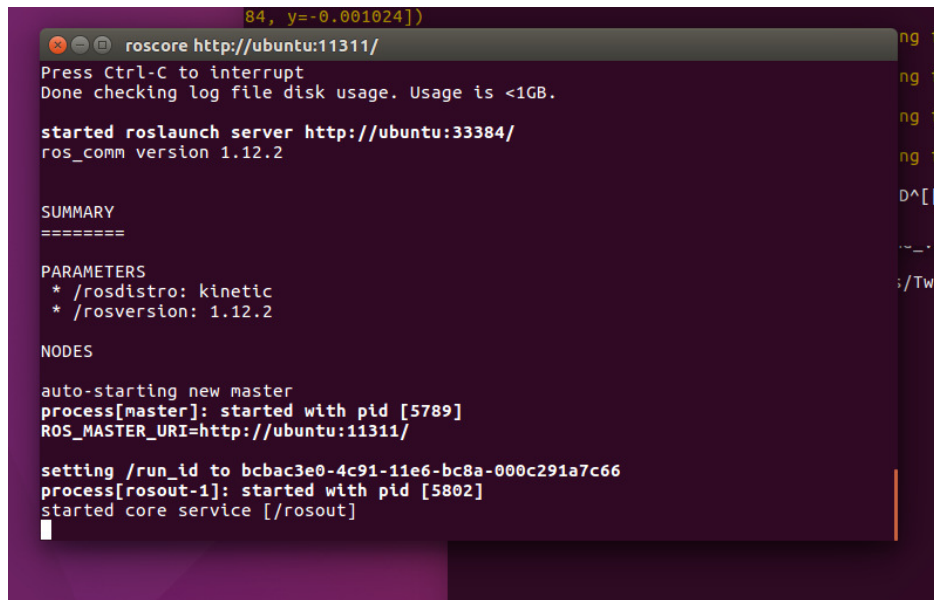
1. a ROS Master
2. a ROS parameter server
3. a rosout logging node

It is run as:

```
$ roscore
```

or

```
$roscore -p 1234 //if port number to run master has to be  
specified
```



```
84, y=-0.001024])
roscore http://ubuntu:11311/
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://ubuntu:33384/
ros_comm version 1.12.2

SUMMARY
=====

PARAMETERS
* /rostdistro: kinetic
* /rosversion: 1.12.2

NODES
auto-starting new master
process[master]: started with pid [5789]
ROS_MASTER_URI=http://ubuntu:11311/

setting /run_id to bcbac3e0-4c91-11e6-bc8a-000c291a7c66
process[rosout-1]: started with pid [5802]
started core service [/rosout]
```

Figure 4 roscore running

**roscnode**: displays information about the ROS nodes currently running. This has to be run on a separate terminal, with roscore running parallely in another terminal.

```
$roscnode list // lists all the active nodes
```

```
$roscnode info /rosout // returns information about the  
mentioned node
```

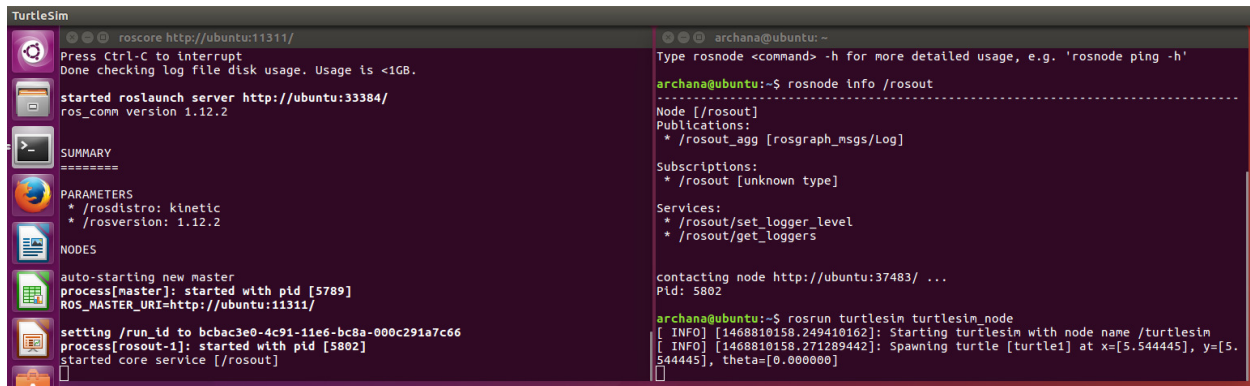


Figure 5 roscore and rosnode running at the same time

**roswarn:** run a node inside a package (without having to know or mention package path)

```
$ roswarn [package_name] [node_name]
```

This example can be run for example.

```
$ roswarn turtlesim turtlesim_node // To run turtlesim node in
                                     turtlesim package
```

We will see a turtle sim window as below:

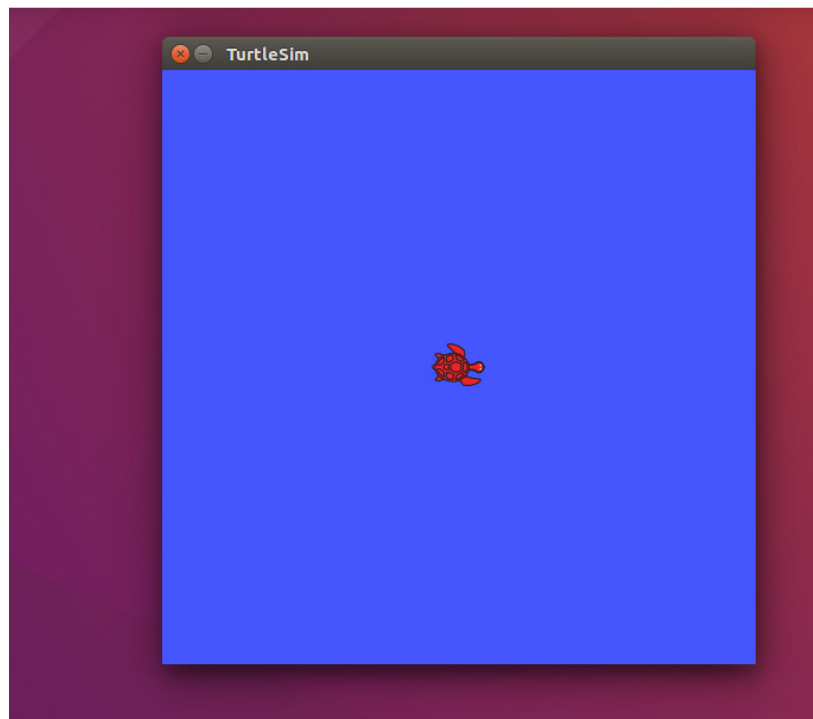


Figure 6 Output of sample turtlesim example

Now a rosnode list returns a list of two nodes: rosout and turtlesim.

The following command is used to change the node's name from the command line itself. But stop the node by closing the turtlesim window, before running this command.

```
$roslaunch turtlesim turtlesim_node __name:=my_turtle
```

We can use `roslaunch ping` option to test if the node is active.

```
$roslaunch ping my_turtle
```

## 9. Understanding ROS topics

With the turtlesim running, run the following command to drive the turtle around using arrow keys on keyboard:

```
$roslaunch turtlesim turtle_teleop_key
```

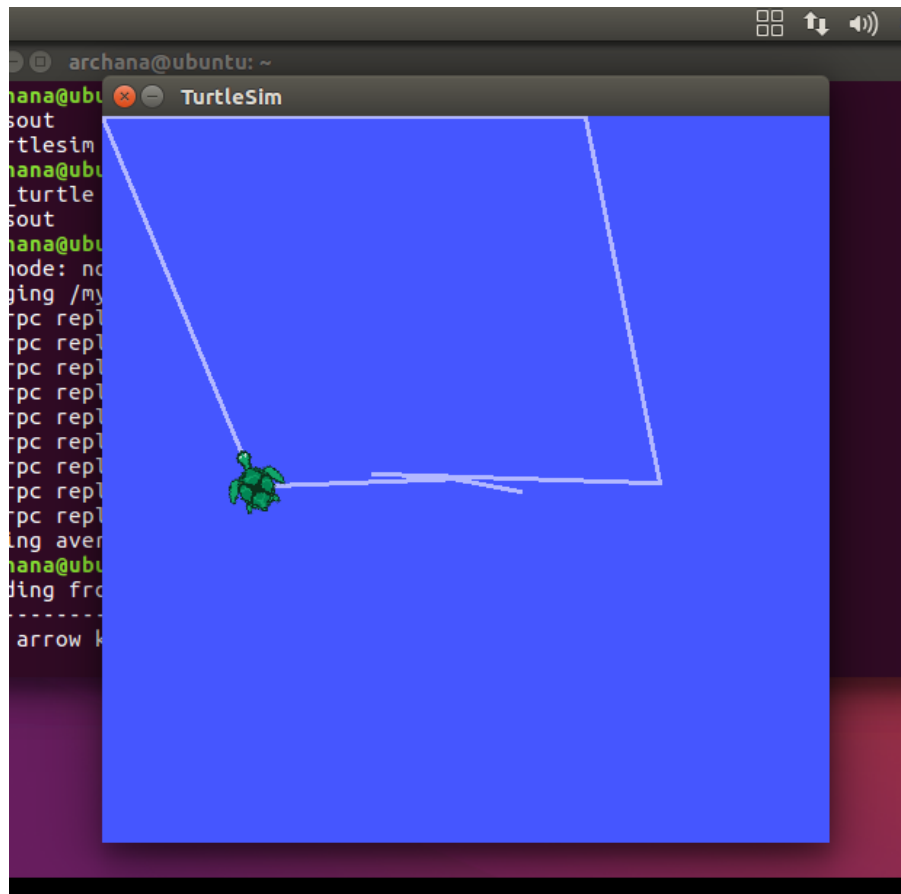


Figure 7 Output of sample moving turtlesim example

In the background, `turtlesim_node` and the `turtle_teleop_key` nodes are communicating with each other over a ROS topic.

`turtle_teleop_key` is publishing the received key strokes on a topic, while `turtlesim_node` subscribes to the same topic to receive the key strokes. This can be visualized using `rqt_graph` installed using:

```
$sudo apt-get install ros-<distro>-rqt
```

```
$sudo apt-get install ros-<distro>-rqt-common-plugins
```

In a new terminal run the following command to view the `rqt_graph`

```
$roslaunch rqt_graph rqt_graph
```

The `rqt_graph` looks like the following:



Figure 8 rqt graph

Here the ROS nodes are blue and green, while the topic is in red. The 2 nodes are communicating over the topic.

**rostopic:** gives information about ROS topics.

```
$rostopic -h // gives available options to be used with rostopic  
command
```

'echo' option displays the data as the arrow keys are pressed.

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

The output is as follows:

```
---  
linear: 2.0  
angular: 0.0  
---  
linear: 2.0  
angular: 0.0  
---  
linear: 2.0  
angular: 0.0
```



```
---  
linear: 2.0  
angular: 0.0  
---  
linear: 2.0  
angular: 0.0
```

\$ rostopic list -v // lists all topics with full information

Messages are the basic form of data exchange between the nodes. For the publisher and subscriber to communicate properly, both sent and received messages should be of same type of message. The topic type is determined by the message type published on it. The type of topic is known by running:

```
$ rostopic type /turtle1/cmd_vel
```

```
$ rosmmsg show geometry_msgs/Twist
```

The output is as follows:

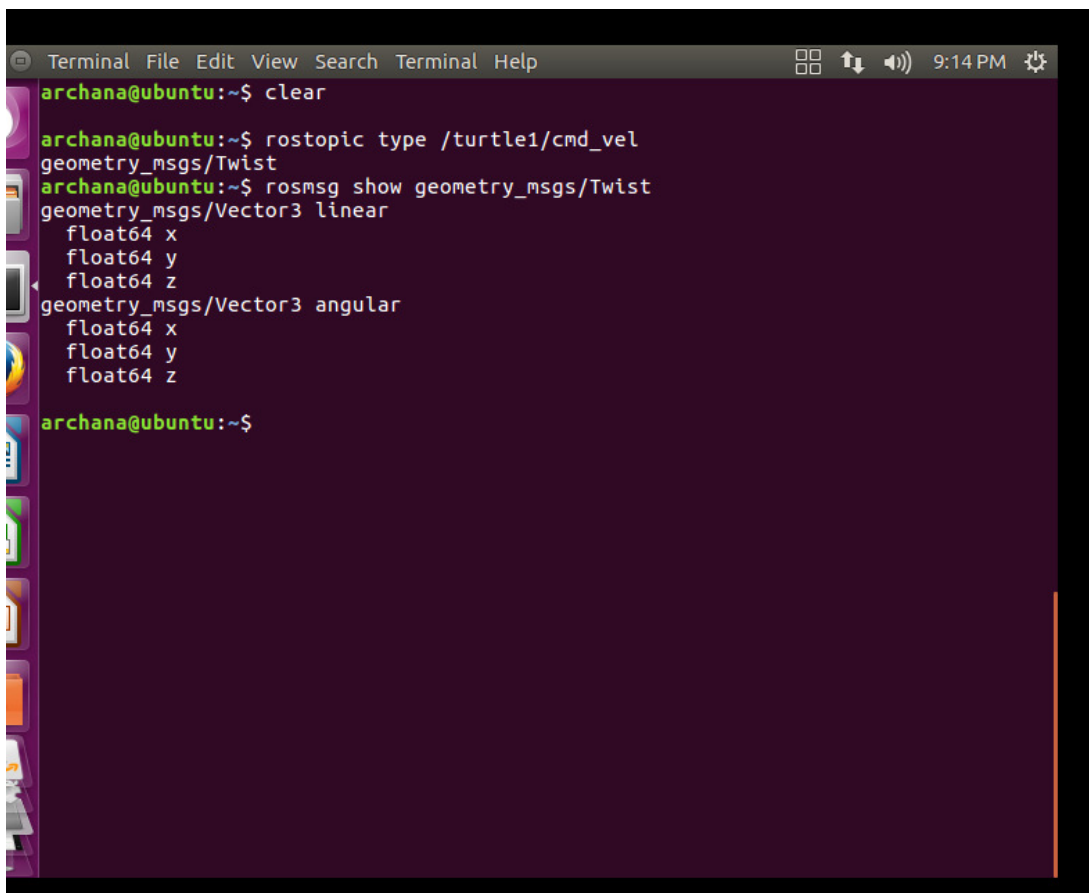
A screenshot of a Linux terminal window with a dark purple background. The terminal shows the following commands and their outputs:  
1. Command: `clear`  
2. Command: `rostopic type /turtle1/cmd_vel`  
 Output: `geometry_msgs/Twist`  
3. Command: `rosmmsg show geometry_msgs/Twist`  
 Output: `geometry_msgs/Vector3 linear`  
 `float64 x`  
 `float64 y`  
 `float64 z`  
 `geometry_msgs/Vector3 angular`  
 `float64 x`  
 `float64 y`  
 `float64 z`  
The terminal window has a title bar with "Terminal" and standard Ubuntu window controls. The prompt is `archana@ubuntu:~$`.

Figure 9 Output of rostopic type command

Instead of using arrows, we can also manually publish data onto the topic, by creating a message using command:

```
$ rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'
```

```
// publish one message with the given type and the given data.
```

There are several useful commands that gives an insight into how the node behaves, what topic it has subscribed to, what kind of data is published/subscribed by it, etc.

Some of them are:

`rostopic hz` : publishes rate at which data is published

`rqt_plot`: displays a moving time plot of data published on topics.

`rosservice list`, `rosservice type`, `rosservice call [service] [args]` are used to know details of the services used by the nodes.

`rosparam` is used to store and manipulate data on ROS parameter server.

`rqt_console` and `rqt_logger_level` are used to debug. `roslaunch` is used to launch many nodes at once.

`roscd` can be used to edit a file inside a package, with just the package name.

`roscd+roswtf` examines system for problems and displays the errors.

Interested members could go ahead with [ROS Tutorials](#) to look at various topics and levels (Beginner & Intermediate). There are also tutorials to write a publisher and subscriber, service and client in C++ and python, recording and playing back data.

Also a ROS cheat sheet with all necessary commands has been written for ROS Hydro, but applies to all later distributions too: [ROS Cheat sheet](#).

## 10. Project specific details

My project uses a LIDAR to gather data from surrounding environment. ROS has a package dedicated to LIDAR called **rplidar**. It reads the raw scan result of LIDAR and converts it into ROS LaserScan message, which can then be manipulated using ROS commands.

## 11. Pros and cons

The positive things are that:

- ROS is an open source software.
- The basic part of ROS is the core installation, however it can be extended using the numerous packages.

- Existing packages and distributions can be mixed and matched using ROS's feature of overlaying.
- It has a huge list of packages which are hardware specific and hence invoking passion and promoting contribution.
- ROS's closely connected user community is a gift to robotic enthusiasts and newbies.
- The various nodes do not have to be on the same system or architecture, making ROS very flexible and simple.

However, ROS does have its own shortcomings:

- Only runs in Linux.
- Still mainly used only in research/academia and not standardized/robust enough to be adopted by the industry.

## 12. References

- [http://library.isr.ist.utl.pt/docs/roswiki/Robots\(2f\)Nao.html](http://library.isr.ist.utl.pt/docs/roswiki/Robots(2f)Nao.html)
- <http://blog.pal-robotics.com/>
- <http://www.active8robots.com/services/programming/>
- <http://www.slideshare.net/AtlayMayada/ros-based-programming-and-visualization-of-quadrotor-helicopters>
- <http://wiki.ros.org/ROS/Tutorials>
- <http://wiki.ros.org/rplidar>
- <https://www.youtube.com/watch?v=MD255BS0YH4>
- <http://robohub.org/ros-101-intro-to-the-robot-operating-system/>