Robot Operating System Developer Guide

By
Murtaza Ameen
School of Electronics, Devi Ahilya Vishwavidyalaya, Indore.

Index

By
Murtaza Ameen
(murtaza678@gmail.com)
School of Electronics, Devi Ahilya Vishwavidyalaya, Indore

1. Overview

In this guide we go through the steps to develop software section of robot. Here we are using Open source Robotics foundation framework i.e. ROS. Before starting with this guide, you should have knowledge of Ubuntu OS and command line tools. This guide is prepared for beginners of ROS. All the steps will only work on ubuntu 16.04 lts xenial xerus.

By
Murtaza Ameen
(murtaza678@gmail.com)
School of Electronics, Devi Ahilya Vishwavidyalaya, Indore

2. Introduction to ROS

2.1 What is ROS ?

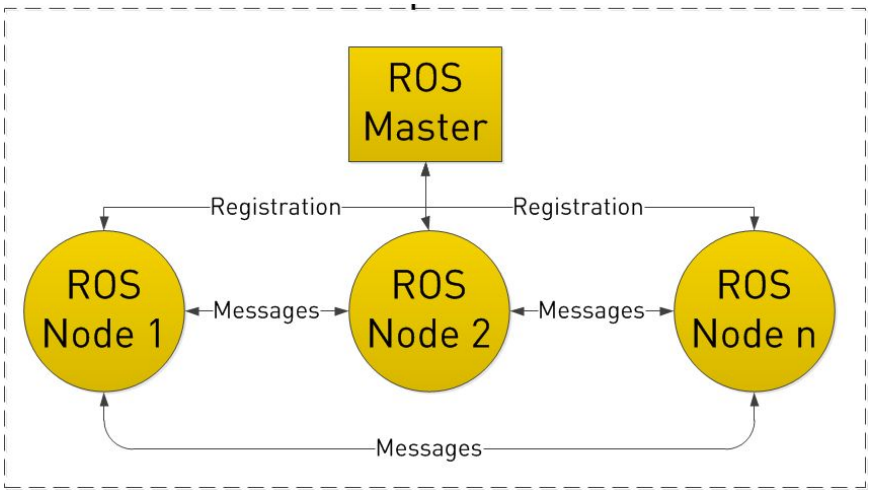
- ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers. [source]

- In simple words ROS is not an operating system, but it provides services like an OS that's why we call it as ROS. Further It is development framework which only runs on UNIX based systems. There are command line tools available to install ROS on UNIX based systems.

- There are various distros available for unix/Linux, but Ubuntu is most suitable and stable one. There are various versions of ROS available, but ROS kinetic is the only latest LTS available, while i am writing this guide. And I have developed source code for ROS kinetic on Ubuntu OS 16.04.

2.2 How ROS works ?

- ROS starts with the ROS Master. The Master allows all other ROS pieces of software (Nodes) to find and talk to each other.



- Just for understanding purpose we can consider an example, which may clarify the concept of ROS master and ROS nodes. We can assume ROS master as a server and the ROS nodes as clients. The purpose of ROS master is to provide the services and resources to ROS nodes. On the other hand ROS nodes are the tasks or we can say processes or simple script. Every node have to register with ROS master so that it can access the services provided by the ROS master. Nodes could be written in python or c++. Using binding tools you can also write your nodes in any language, but i suggest for python.

- Topics are named buses over which nodes exchange messages. After successful registration of node with Master, now it can publish and subscribe to the topic(s).

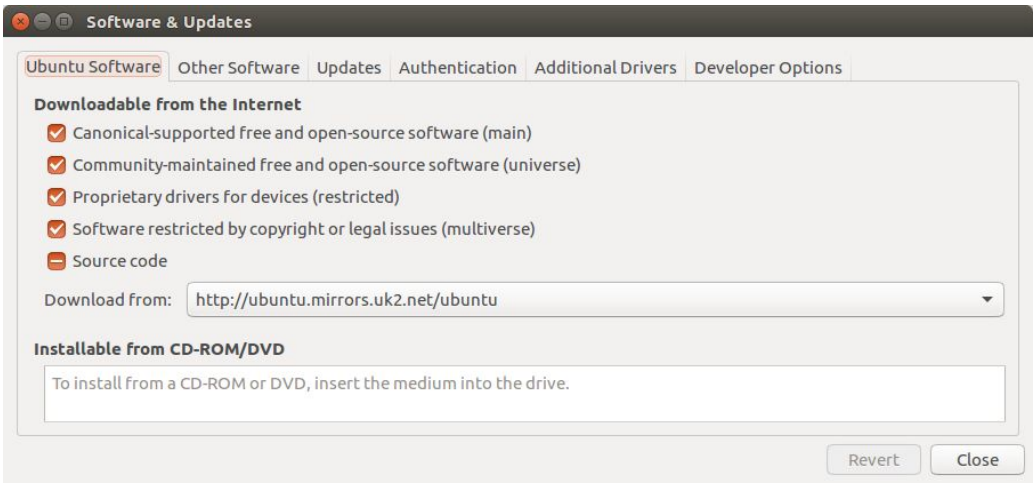
3. Installation

ROS (kinetic) Installation on Ubuntu (16.04)

- ROS Kinetic ONLY supports Wily (Ubuntu 15.10), Xenial (Ubuntu 16.04) and Jessie (Debian 8) for debian packages.

3.1 Configure Ubuntu Repositories:

- Configure your Ubuntu repositories to allow restricted, universe and multiverse.



3.2 Setup your source.list

- Setup your computer to accept software from packages.ros.org.
  - **sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'**

Comment: sh calls the program sh as interpreter and the -c flag means execute the following command as interpreted by this program.
In Ubuntu, sh is usually symlinked to /bin/dash, meaning that if you execute a command with sh -c the dash shell will be used to execute the command instead of bash. You

should use sh -c when you want to execute a command specifically with that shell instead of bash.

3.3 Setup your keys

- **sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-key 421C365BD9FF1F717815A3895523BAEEB01FA116**

Comment: apt-key is used to manage the list of keys used by apt to authenticate packages. Packages which have been authenticated using these keys will be considered trusted.
Adv: Pass advanced options to gpg. With adv --recv-key you can e.g. download key from keyservers directly into the the trusted set of keys.

3.4 Installation

- First make sure your debian package index is up-to-date.
  - **sudo apt-get update**
  - **sudo apt-get install ros-kinetic-desktop-full**

3.5 Initialize rosdep

- Before you can use ROS, you will need to initialize rosdep. rosdep enables you to easily install system dependencies for source you want to compile and is required to run some core components in ROS.
  - **sudo rosdep init**
  - **rosdep update**

3.6 Environment setup

- It's convenient if the ROS environment variables are automatically added to your bash session every time a new shell is launched:
  - **echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc**
  - **source ~/.bashrc**

3.7 Dependencies for building packages

- Up to now you have installed what you need to run the core ROS packages. To create and manage your own ROS workspaces, there are various tools and requirements that are distributed separately. For example, rosinstall is a frequently used command-line tool that enables you to easily download many source trees for ROS packages with one command.
- To install this tool and other dependencies for building ROS packages, run:
    - **sudo apt-get install python-rosinstall python-rosinstall-generator python-wstool build-essential**

4. Environment Setup

4.1 Installation

- Before starting this chapter please complete installation procedure described in Chapter 3

4.2 Managing Your Environment

- During the installation of ROS, you will see that you are prompted to source one of several setup.*sh files, or even add this 'sourcing' to your shell startup script (as we had done in section 3.6). This is required because ROS relies on the notion of combining spaces using the shell environment. This makes developing against different versions of ROS or against different sets of packages easier.
- If you are ever having problems finding or using your ROS packages make sure that you have your environment properly setup. A good way to check is to ensure that environment variables like ROS_ROOT and ROS_PACKAGE_PATH are set:
    - **printenv | grep ROS**

If everything is installed properly, you will get similar output as above.

```
murtaza@murtaza-Vostro-15-3568:~$ printenv | grep ROS
murtaza@murtaza-Vostro-15-3568:~$
```

If not sourced properly, then you will get output as above.

Solution:

- Source the shell file by running the below command.
- **source /opt/ros/kinetic/setup.bash**
- Every time you need to run above command whenever you open new shell.
- Write this command in .bashrc file so that we do not need to source every time, for that run the command:
- **echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc**
- And restart shell.

4.3 ROS Workspace

- Till now we had settle up the requirements of ROS. Now we want to write our own package and code. For working with ROS you need to have a Workspace, where you use to save your codes and other resources.
- Workspace is nothing but just a directory/folder. The ROS workspace is not an ordinary workspace with some python and c++ codes. This workspace is build using a catkin build system.
- For us(human) it is easy to understand what type of data is present in this directory, but for machine it is not as simple. For that you need to arrange the things as per the understanding of machine. You need to write your codes for ROS working, but along with that you need to write few files which is used by the ROS for understand and interpreting your code with ROS master.
- To generate that ROS supporting files, we use Catkin build tool. Which build all the essential files.
- Soo let's create and build a catkin workspace:
- **mkdir -p ~/soex_ws/src**
- The above command will create a directory named soex_ws, you can rename with your name of choice. Inside cakin_ws there is a directory named src. Catkin make searches for src folder in ROS workspace. So a catkin workspace can have any name but it should contain src folder.
- **cd ~/soex_ws/**
- **catkin_make**

```
murtaza@murtaza-Vostro-15-3568:~$ mkdir -p ~/soex_ws/src
murtaza@murtaza-Vostro-15-3568:~$ cd soex_ws/
murtaza@murtaza-Vostro-15-3568:~/soex_ws$ catkin_make
```

```
Base path: /home/murtaza/soex_ws
Source space: /home/murtaza/soex_ws/src
Build space: /home/murtaza/soex_ws/build
Devel space: /home/murtaza/soex_ws/devel
Install space: /home/murtaza/soex_ws/install
Creating symlink "/home/murtaza/soex_ws/src/CMakeLists.txt" pointin
g to "/opt/ros/kinetic/share/catkin/cmake/toplevel.cmake"
####
#### Running command: "cmake /home/murtaza/soex_ws/src -DCATKIN_DEV
EL_PREFIX=/home/murtaza/soex_ws/devel -DCMAKE_INSTALL_PREFIX=/home/
murtaza/soex_ws/install -G Unix Makefiles" in "/home/murtaza/soex_w
s/build"
####
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Using CATKIN_DEVEL_PREFIX: /home/murtaza/soex_ws/devel
-- Using CMAKE_PREFIX_PATH: /opt/ros/kinetic
-- This workspace overlays: /opt/ros/kinetic
-- Found PythonInterp: /usr/bin/python (found version "2.7.12")
-- Using PYTHON_EXECUTABLE: /usr/bin/python
-- Using Debian Python package layout
-- Using empy: /usr/bin/empy
-- Using CATKIN_ENABLE_TESTING: ON
-- Call enable_testing()
-- Using CATKIN_TEST_RESULTS_DIR: /home/murtaza/soex_ws/build/test_
results
-- Looking for pthread.h
-- Looking for pthread.h - found
-- Looking for pthread_create
-- Looking for pthread_create - not found
```

- The catkin_make command is a convenience tool for working with catkin workspaces. Running it the first time in your workspace, it will create a

CMakeLists.txt link in your 'src' folder. Additionally, if you look in your current directory you should now have a 'build' and 'devel' folder. Inside the 'devel' folder you can see that there are now several setup.*sh files. Sourcing any of these files will overlay this workspace on top of your environment. To understand more about this see the general catkin documentation: catkin. Before continuing source your new setup.*sh file:[source]

- **source ~/soex_ws/devel/setup.bash**
- To make sure your workspace is properly overlayed by the setup script, make sure ROS_PACKAGE_PATH environment variable includes the directory you're in.
- **echo $ROS_PACKAGE_PATH**

  Output: /home/youruser/soex_ws/src:/opt/ros/kinetic/share

5. ROS Filesystem

- This chapter introduces ROS filesystem concepts, and covers using the roscd, rosls, and rospack command line tools.

5.1 Prerequisites

- For this tutorial we will inspect a package in ros-tutorials, please install it using
- **sudo apt-get install ros-kinetic-ros-tutorials.**

5.2 Quick Overview of Filesystem Concepts

- **Packages**: Packages are the software organization unit of ROS code. Each package can contain libraries, executables, scripts, or other artifacts.
- **Manifest**(package.xml): A manifest is a description of a *package*. It serves to define dependencies between *packages* and to capture meta information about the *package* like version, maintainer, license, etc…

Comment: As we already discussed that, all our code will be in a package. Now for our understanding it is easy to say that this directory is a package because it consist of codes, but ROS doesn't understand that. For that we need to add two files in any package for make it ROS readable. One is package.xml file and another is CmakeList.txt file. If any directory in workspace, contain these two files will be considered as ROS package.

5.3 Filesystem Tools

- If you have single package in your ROS workspace, then there is no issue to navigate to that package. Now assume a scenario, You are developing a system for autonomous car, which is consist of thousands of scripts in hundreds of packages, in that case navigating with command-line tools such as ls and cd can be very tedious which is why ROS provides tools to help you.

5.3.1 Using **rospack**

- rospack allows you to get information about packages.
  Ex:1

```
murtaza@murtaza-Vostro-15-3568:~$ rospack find turtlesim
/opt/ros/kinetic/share/turtlesim
murtaza@murtaza-Vostro-15-3568:~$
```

  Ex:2

```
murtaza@murtaza-Vostro-15-3568:~$ rospack find rospy
/opt/ros/kinetic/share/rospy
murtaza@murtaza-Vostro-15-3568:~$
```

  Rospack tool takes two argument, first one is operation and second one is package name. In example 1 turtlesim is the package name and find is the operation. It is used when you know the package name, and you want to find the location of that package.
- There are other operation also present, which you can use with rospack. rospack find is the most common and useful operation. To explore other operation type **rospack help.**

5.3.2 Using **roscd**

- It allows you to change directory (cd) directly to a package or a stack.
  Ex:1

```
murtaza@murtaza-Vostro-15-3568:~$ roscd rospy
murtaza@murtaza-Vostro-15-3568:/opt/ros/kinetic/share/rospy$
```

- roscd is used for changing ros directory same as cd. But in case of cd you need to provide exact path of the directory where you want to switch. Here you need to just type roscd and package name.

- Now let's print the working directory using the Unix command pwd:

```
murtaza@murtaza-Vostro-15-3568:/opt/ros/kinetic/share/rospy$ pwd
/opt/ros/kinetic/share/rospy
```

  You should see: YOUR_INSTALL_PATH/share/rospy

- You can see that YOUR_INSTALL_PATH/share/roscpp is the same path that rospack find gave in the previous example.

  Note that roscd, like other ROS tools, will *only* find ROS packages that are within the directories listed in your ROS_PACKAGE_PATH. To see what is in your ROS_PACKAGE_PATH, type:
  echo $ROS_PACKAGE_PATH

```
murtaza@murtaza-Vostro-15-3568:~$ echo $ROS_PACKAGE_PATH/
/opt/ros/kinetic/share/
```

  Your ROS_PACKAGE_PATH should contain a list of directories where you have ROS packages separated by colons.

5.3.3 roscd log

- roscd log will take you to the folder where ROS stores log files. Note that if you have not run any ROS programs yet, this will yield an error saying that it does not yet exist.

5.3.4 Using rosls

- It allows you to **ls** directly in a package by name rather than by absolute path.
  Ex:1

```
murtaza@murtaza-Vostro-15-3568:~$ rosls roscpp_tutorials/
cmake  launch  package.xml  srv
```

5.3.5 Tab completion

- It can get tedious to type out an entire package name. In the previous example, roscpp_tutorials is a fairly long name. Luckily, some ROS tools support TAB completion.

  Start by typing:

  roscd roscpp_tut<<< now push the TAB key >>>

  After pushing the **TAB** key, the command line should fill out the rest:

  roscd roscpp_tutorials/

This works because roscpp_tutorials is currently the only ROS package that starts with roscpp_tut.

Now try typing:

roscd tur<<< now push the TAB key >>>

After pushing the **TAB** key, the command line should fill out as much as possible:

roscd turtle

However, in this case there are multiple packages that begin with turtle. Try typing TAB another time. This should display all the ROS packages that begin with turtle:

```
murtaza@murtaza-Vostro-15-3568:~$ roscd turtle
turtle_actionlib/  turtlesim/         turtle_tf/
   turtle_tf2/
murtaza@murtaza-Vostro-15-3568:~$ roscd turtle
```

On the command line you should still have:

roscd turtle

Now type an s after turtle and then push TAB:

roscd turtles<<< now push the TAB key >>>

Since there is only one package that starts with turtles, you should see:

roscd turtlesim/

If you want to see a list of all currently installed packages, you can use tab completion for that as well:

rosls <<< now push the TAB key twice >>>

5.4 Review

- You may have noticed a pattern with the naming of the ROS tools:
    - rospack = ros + pack(age)
    - roscd = ros + cd
    - rosls = ros + ls

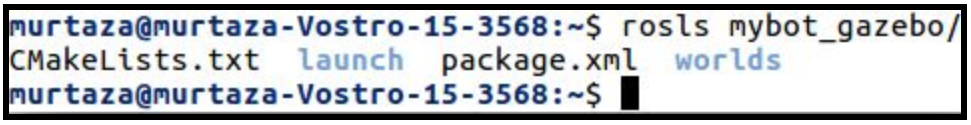
This naming pattern holds for many of the ROS tools.

6. ROS Package

- This chapter shows, how to create a ROS package inside a ROS workspace. As we have already discussed a brief view of a package in last chapter. This package discuss in detail about ROS package. For building a ROS package, we will use catkin build tool. So, the package will be called as catkin package.

6.1 What makes up a catkin Package?

- For a package to be considered a catkin package it must meet a few requirements:
    - The package must contain a catkin compliant package.xml file.
        - That package.xml file provides meta information about the package.
    - The package must contain a CMakeLists.txt which uses catkin.
        - If it is a catkin metapackage it must have the relevant boilerplate CMakeLists.txt file.
    - Each package must have its own folder
        - This means no nested packages nor multiple packages sharing the same directory.



Note: As we can see that, mybot_gazebo is package, which contains two mandatory files package.xml and CmakeList.txt

- The simplest possible package might have a structure which looks like this:

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

6.2 Packages in Catkin Workspace

- The recommended method of working with catkin packages is using a catkin workspace, but you can also build catkin packages standalone. A trivial workspace might look like this:

```
workspace_folder/        -- WORKSPACE
    src/                 -- SOURCE SPACE
        CMakeLists.txt       -- 'Toplevel' CMake file, provided by catkin
        package_1/
            CMakeLists.txt     -- CMakeLists.txt file for package_1
            package.xml        -- Package manifest for package_1
        ...
        package_n/
            CMakeLists.txt     -- CMakeLists.txt file for package_n
            package.xml        -- Package manifest for package_n
```

Comment: As we had seen in section 4.3 that, there is workspace in which your packages are present. All packages are lies in src folder. Each package has two files which contains the information about the package. In src folder there is a CmakeList.txt file, this cmakelist file is the toplevel cmake file which contains the information of packages present in src folder.

- Create a catkin workspace as defined in the section 4.3, if you have not created yet.

6.3 Creating a Catkin Package

- This chapter will demonstrate how to use the catkin_create_pkg script to create a new catkin package, and what you can do with it after it has been created.
- First change to the source space directory of the catkin workspace you created in section 4.3

  cd ~/soex_ws/src

```
murtaza@murtaza-Vostro-15-3568:~$ cd ~/soex_ws/src/
murtaza@murtaza-Vostro-15-3568:~/soex_ws/src$
```

- Now use the catkin_create_pkg script to create a new package called ‘murtaza_’ which depends on std_msgs and rospy:

  catkin_create_pkg murtaza_ rospy std_msgs

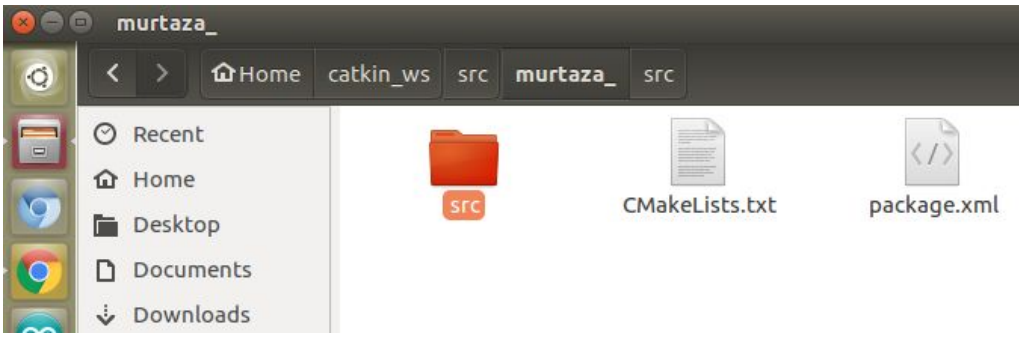
```
murtaza@murtaza-Vostro-15-3568:~/soex_ws/src$ catkin_create_pkg murtaza_
 rospy std_msgs
Created file murtaza_/package.xml
Created file murtaza_/CMakeLists.txt
Created folder murtaza_/src
Successfully created files in /home/murtaza/soex_ws/src/murtaza_. Please
 adjust the values in package.xml.
murtaza@murtaza-Vostro-15-3568:~/soex_ws/src$
```

This will create a murtaza_ folder which contains a package.xml and a CMakeLists.txt, which have been partially filled out with the information you gave catkin_create_pkg.

The first dependency is rospy, rospy package contains all the methods and function to access the services provided by ROS. You need to import rospy in your python code to access ROS functionalities. Rospy is used when you are writing nodes in python and roscpp is used when you are writing nodes in cpp. The second package is std_msgs, which contains all the message(Data) types, used for transferring data. Do not worry, this things will be clear enough when you start writing nodes.

catkin_create_pkg requires that you give it a package_name and optionally a list of dependencies on which that package depends:

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



6.4 Building a catkin workspace and sourcing the setup file

- Now you need to build the packages in the catkin workspace:

  cd ~/soex_ws
  catkin_make

- When you will run the above command, lots of processing will done on the terminal. Actually, catkin is a build system, it builds the required data for ROS, just by reading package.xml and CmakeList.txt files. The output of the terminal will be look like as shown below.

```
Base path: /home/murtaza/soex_ws
Source space: /home/murtaza/soex_ws/src
Build space: /home/murtaza/soex_ws/build
Devel space: /home/murtaza/soex_ws/devel
Install space: /home/murtaza/soex_ws/install
Creating symlink "/home/murtaza/soex_ws/src/CMakeLists.txt" pointin
g to "/opt/ros/kinetic/share/catkin/cmake/toplevel.cmake"
####
#### Running command: "cmake /home/murtaza/soex_ws/src -DCATKIN_DEV
EL_PREFIX=/home/murtaza/soex_ws/devel -DCMAKE_INSTALL_PREFIX=/home/
murtaza/soex_ws/install -G Unix Makefiles" in "/home/murtaza/soex_w
s/build"
####
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Using CATKIN_DEVEL_PREFIX: /home/murtaza/soex_ws/devel
-- Using CMAKE_PREFIX_PATH: /opt/ros/kinetic
-- This workspace overlays: /opt/ros/kinetic
-- Found PythonInterp: /usr/bin/python (found version "2.7.12")
-- Using PYTHON_EXECUTABLE: /usr/bin/python
-- Using Debian Python package layout
-- Using empy: /usr/bin/empy
-- Using CATKIN_ENABLE_TESTING: ON
-- Call enable_testing()
-- Using CATKIN_TEST_RESULTS_DIR: /home/murtaza/soex_ws/build/test_
results
-- Looking for pthread.h
-- Looking for pthread.h - found
-- Looking for pthread_create
-- Looking for pthread_create - not found
```

- After the workspace has been built it has created a similar structure in the devel subfolder as you usually find under /opt/ros/kinetic.

```
murtaza@murtaza-Vostro-15-3568:~$ cd /opt/ros/kinetic/
murtaza@murtaza-Vostro-15-3568:/opt/ros/kinetic$ ls
bin     etc      lib          setup.sh         setup.zsh
env.sh  include  setup.bash  _setup_util.py  share
```

And

```
murtaza@murtaza-Vostro-15-3568:~$ cd catkin_ws/devel/
murtaza@murtaza-Vostro-15-3568:~/catkin_ws/devel$ ls
env.sh  include  lib  setup.bash  setup.sh  _setup_util.py
 setup.zsh  share
murtaza@murtaza-Vostro-15-3568:~/catkin_ws/devel$
```

- To add the workspace to your ROS environment you need to source the generated setup file:

  . ~/soex_ws/devel/setup.bash

  Comment: One thing to be noticed, in section 3.6 (environment setup) we have sourced a setup file from /**opt/ros/kinetic/setup.bash.** Also we had written source command in .bashrc file so that, we do not need to source this setup file every time when we start a new terminal. This setup file sets the environmental variables, which allows our shell terminal to interpret with ROS command line tools and ROS inbuilt packages. To use inbuilt packages we have written sourcing command in .bashrc. But now we have created our own packages in our workspace. ROS is not aware of our workspace. To make ROS aware to our packages we need to source setup file of our workspace. That's why we run the above command. If you only want to work with single workspace then you can add this sourcing command in .bashrc file.

6.5 Package Dependencies

6.5.1 First order dependencies:

- When using catkin_create_pkg earlier, a few package dependencies were provided. These **first-order** dependencies can now be reviewed with the rospack tool.
  rospack depends1 murtaza_

```
murtaza@murtaza-Vostro-15-3568:~$ rospack depends1 murtaza_
rospy
std_msgs
```

- As you can see, rospack lists the same dependencies that were used as arguments when running catkin_create_pkg.These dependencies for a package are stored in the **package.xml** file:

  roscd muraza_

cat package.xml

```
murtaza@murtaza-Vostro-15-3568:~$ . soex_ws/devel/setup.bash
murtaza@murtaza-Vostro-15-3568:~$ roscd murtaza_/
murtaza@murtaza-Vostro-15-3568:~/soex_ws/src/murtaza_$ cat pack
age.xml
<?xml version="1.0"?>
<package format="2">
  <name>murtaza_</name>
  <version>0.0.0</version>
  <description>The murtaza_ package</description>

  <!-- One maintainer tag required, multiple allowed, one perso
n per tag -->
  <!-- Example:  -->
  <!-- <maintainer email="jane.doe@example.com">Jane Doe</maint
ainer> -->
  <maintainer email="murtaza@todo.todo">murtaza</maintainer>


  <!-- One license tag required, multiple allowed, one license
per tag -->
  <!-- Commonly used license strings: -->
  <!--   BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2
.1, LGPLv3 -->
  <license>TODO</license>


  <!-- Url tags are optional, but multiple are allowed, one per
 tag -->
  <!-- Optional attribute type can be: website, bugtracker, or
repository -->
  <!-- Example: -->
  <!-- <url type="website">http://wiki.ros.org/murtaza_</url> -
->


  <!-- Author tags are optional, multiple are allowed, one per
tag -->
  <!-- Authors do not have to be maintainers, but could be -->
  <!-- Example: -->
  <!-- <author email="jane.doe@example.com">Jane Doe</author> -
->
```

6.5.2 Indirect Dependencies

- In many cases, a dependency will also have its own dependencies. For instance, rospy has other dependencies.

rospack depends1 rospy

```
murtaza@murtaza-Vostro-15-3568:~$ rospack depends1 rospy
genpy
roscpp
rosgraph
rosgraph_msgs
roslib
std_msgs
```

- A package can have quite a few indirect dependencies. Luckily rospack can recursively determine all nested dependencies.

rospack depends murtaza_

```
murtaza@murtaza-Vostro-15-3568:~$ rospack depends murtaza_
catkin
genmsg
genpy
cpp_common
rostime
roscpp_traits
roscpp_serialization
message_runtime
gencpp
geneus
gennodejs
genlisp
message_generation
rosbuild
rosconsole
std_msgs
rosgraph_msgs
xmlrpcpp
roscpp
rosgraph
rospack
roslib
rospy
```

7. ROS Nodes

- This chapter introduces ROS graph concepts and discusses the use of roscore, rosnode, and rosrun command line tools.

7.1 Prerequisites

- To understand this we will use a lightweight simulator, please install using
- **sudo apt-get install ros-kinetic-ros-tutorials**

7.2 Quick Overview of Graph concepts

- Nodes: A node is an executable that uses ROS to communicate with other nodes.
- Messages: ROS data type used when subscribing or publishing to a topic.
- Topics: Nodes can *publish* messages to a topic as well as *subscribe* to a topic to receive messages.
- Master: Name service for ROS (i.e. helps nodes find each other)
- rosout: ROS equivalent of stdout/stderr
- roscore: Master + rosout + parameter server (parameter server will be introduced later)

7.3 Nodes

- A node really isn't much more than an executable file within a ROS package. ROS nodes use a ROS client library to communicate with other nodes. Nodes can publish or subscribe to a Topic. Nodes can also provide or use a Service.

7.4 Client Libraries

- ROS client libraries allow nodes written in different programming languages to communicate:
    - roscpp = cpp client library
    - Rospy = python client library

Comment: ROS basically provides stable supports for two languages for writing your nodes. Python and c plus plus are used. For calling ROS internal functions you need to import client libraries, if you are writing in python then you need to import rospy client library.

7.5 roscore

- roscore is the first thing you should run when using ROS.
- **roscore**
- You will see something similar to:

```
murtaza@murtaza-Vostro-15-3568:~$ roscore
... logging to /home/murtaza/.ros/log/dcee01f6-5926-11e8-95
98-34f64b9c171f/roslaunch-murtaza-Vostro-15-3568-4063.log
Checking log directory for disk usage. This may take awhile
.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://murtaza-Vostro-15-3568:4162
6/
ros_comm version 1.12.12


SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.12

NODES

auto-starting new master
process[master]: started with pid [4075]
ROS_MASTER_URI=http://murtaza-Vostro-15-3568:11311/

setting /run_id to dcee01f6-5926-11e8-9598-34f64b9c171f
process[rosout-1]: started with pid [4088]
started core service [/rosout]
```

7.6 Using rosnode

- Open up a **new terminal**, and let's use **rosnode** to see what running roscore did... Bare in mind to keep the previous terminal open either by opening a new tab or simply minimizing it.
- rosnode displays information about the ROS nodes that are currently running. The rosnode list command lists these active nodes:
- **rosnode list**

```
murtaza@murtaza-Vostro-15-3568:~$ 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.
- **rosnode info /rosout**

```
murtaza@murtaza-Vostro-15-3568:~$ 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://murtaza-Vostro-15-3568:40682/ ...
Pid: 4088
```

Summary: rosnode is a ros command line tool used to perform various operation on active nodes. Beside rosnode list and rosnode info there are other operations also. Type rosnode and press tab twice, you will get all other operations name.

7.7 Using rosrun

- rosrun allows you to use the package name to directly run a node within a package (without having to know the package path).
- Usage:
    - `$ rosrun [package_name] [node_name]`
- So now we can run the turtlesim_node in the turtlesim package.
- Then in a new terminal:
    - `$ rosrun turtlesim turtlesim_node`

```
murtaza@murtaza-Vostro-15-3568:~$ rosrun turtlesim turtl
esim_node
[ INFO] [1526489115.308064537]: Starting turtlesim with
node name /turtlesim
[ INFO] [1526489115.328930139]: Spawning turtle [turtle1
] at x=[5.544445], y=[5.544445], theta=[0.000000]
```

- You will see the turtlesim window

- **NOTE:** The turtle may look different in your turtlesim window. Don't worry about it - there are many types of turtle and yours is a surprise!
- In a **new terminal**:
    - **rosnode list**

```
murtaza@murtaza-Vostro-15-3568:~$ rosnode list
/rosout
/turtlesim
```

- One powerful feature of ROS is that you can reassign Names from the command-line.
- Close the turtlesim window to stop the node (or go back to the rosrun turtlesim terminal and use ctrl-C). Now let's re-run it, but this time use a Remapping Argument to change the node's name:
- **rosrun turtlesim turtlesim_node __name:=soex_turtle**
- Now, if we go back and use rosnode list:

```
murtaza@murtaza-Vostro-15-3568:~$ rosnode list
/rosout
/soex_turtle
/turtlesim
```

- Note: If you still see /turtlesim in the list, it might mean that you stopped the node in the terminal using ctrl-C instead of closing the window, or that you don't have the $ROS_HOSTNAME environment variable defined as described in Network Setup - Single Machine Configuration. You can try cleaning the rosnode list with: $ rosnode cleanup
    - **rosnode cleanup**

```
murtaza@murtaza-Vostro-15-3568:~$ rosnode cleanup
ERROR: connection refused to [http://murtaza-Vostro-
15-3568:44888/]
Unable to contact the following nodes:
 * /turtlesim
Warning: these might include alive and functioning n
odes, e.g. in unstable networks.
Cleanup will purge all information about these nodes
 from the master.
Please type y or n to continue:
y
Unregistering /turtlesim
done
murtaza@murtaza-Vostro-15-3568:~$
```

- Let's use another rosnode command, ping, to test that it's up:
    - **rosnode ping /soex_turtle**

```
murtaza@murtaza-Vostro-15-3568:~$ rosnode ping
/soex_turtle
rosnode: node is [/soex_turtle]
pinging /soex_turtle with a timeout of 3.0s
xmlrpc reply from http://murtaza-Vostro-15-3568
:39642/ time=0.379086ms
xmlrpc reply from http://murtaza-Vostro-15-3568
:39642/ time=1.612186ms
xmlrpc reply from http://murtaza-Vostro-15-3568
:39642/ time=1.620054ms
xmlrpc reply from http://murtaza-Vostro-15-3568
:39642/ time=1.591921ms
xmlrpc reply from http://murtaza-Vostro-15-3568
:39642/ time=1.651049ms
^Cping average: 1.370859ms
```

7.8 Review

- What was covered
    - roscore = ros+core : master (provides name service for ROS) + rosout (stdout/stderr) + parameter server (parameter server will be introduced later)
    - rosnode = ros+node : ROS tool to get information about a node.
    - rosrun = ros+run : runs a node from a given package.

Comment: Each and every function method and codes must be packed in a package. Similarly, turtlesim is also a package present in ROS. turtlesim contains different nodes for different functions. Write now we have used one of the node present in turtlesim. In the next chapter we will see, how we can send data from one node to other via topics.

8. ROS Topics

- This chapter introduces ROS topics as well as using the rostopic and rqt_plot command line tools.

8.1 Setup

8.1.1 roscore

- Let's start by making sure that we have roscore running, **in a new terminal**:
  - **roscore**
  -
```
murtaza@murtaza-Vostro-15-3568:~$ roscore
... logging to /home/murtaza/.ros/log/dcee01f6-5926-11e8-95
98-34f64b9c171f/roslaunch-murtaza-Vostro-15-3568-4063.log
Checking log directory for disk usage. This may take awhile
.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://murtaza-Vostro-15-3568:4162
6/
ros_comm version 1.12.12


SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.12

NODES

auto-starting new master
process[master]: started with pid [4075]
ROS_MASTER_URI=http://murtaza-Vostro-15-3568:11311/

setting /run_id to dcee01f6-5926-11e8-9598-34f64b9c171f
process[rosout-1]: started with pid [4088]
started core service [/rosout]
```
- If you left roscore running from the last tutorial, you may get the error message:

```
murtaza@murtaza-Vostro-15-3568:~$ roscore
... logging to /home/murtaza/.ros/log/dcee01f6-
5926-11e8-9598-34f64b9c171f/roslaunch-murtaza-V
ostro-15-3568-6269.log
Checking log directory for disk usage. This may
 take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1G
B.

started roslaunch server http://murtaza-Vostro-
15-3568:34324/
ros_comm version 1.12.12


SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.12

NODES

roscore cannot run as another roscore/master is
 already running.
Please kill other roscore/master processes befo
re relaunching.
The ROS_MASTER_URI is http://murtaza-Vostro-15-
3568:11311/
The traceback for the exception was written to
the log file
murtaza@murtaza-Vostro-15-3568:~$
```

- This is fine. Only one roscore needs to be running.

8.1.2 turtlesim

- For understanding topics, we will again use turtlesim.
- rosrun turtlesim turtlesim_node

```
murtaza@murtaza-Vostro-15-3568:~$ rosrun turtlesim turtl
esim_node
[ INFO] [1526489115.308064537]: Starting turtlesim with
node name /turtlesim
[ INFO] [1526489115.328930139]: Spawning turtle [turtle1
] at x=[5.544445], y=[5.544445], theta=[0.000000]
```

8.1.2 turtle keyboard Teleoperation

- We'll also need something to drive the turtle around with. Please run **in a new terminal**:
- **rosrun turtlesim turtle_teleop_key**

```
murtaza@murtaza-Vostro-15-3568:~$ rosrun turtlesim turtle
_teleop_key
Reading from keyboard
---------------------------
Use arrow keys to move the turtle.
```

- 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.

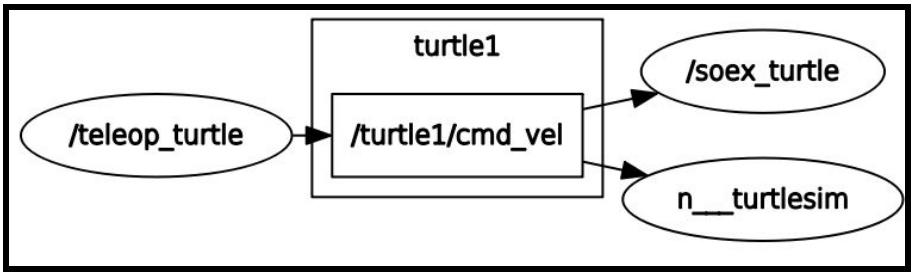
- Now that you can drive your turtle around, let's look at what's going on behind the scenes.

8.2 ROS Topics

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

8.2.1 Using rqt_graph

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



- Elliptical shapes are nodes and rectangular shapes are topics. The direction of arrow showing which node is publishing/subscribing to which topic.

8.2.2 Using rostopic

- The rostopic tool allows you to get information about ROS **topics**.
- You can use the help option to get the available subcommands for rostopic.
  - **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

```
^Cmurtaza@murtaza-Vostro-15-3568:~$ rostopic -h
rostopic is a command-line tool for printing information
 about ROS Topics.

Commands:
        rostopic bw     display bandwidth used by topic
        rostopic delay  display delay of topic from time
stamp in header
        rostopic echo   print messages to screen
        rostopic find   find topics by type
        rostopic hz     display publishing rate of topic

        rostopic info   print information about active t
opic
        rostopic list   list active topics
        rostopic pub    publish data to topic
        rostopic type   print topic or field type

Type rostopic <command> -h for more detailed usage, e.g.
 'rostopic echo -h'

murtaza@murtaza-Vostro-15-3568:~$
```

8.2.3 Using rostopic echo

- rostopic echo shows the data published on a topic.
- **rostopic echo /turtle1/cmd_vel**
- 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.**

```
murtaza@murtaza-Vostro-15-3568:~$ 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
---
```

- Now let's look at rqt_graph again. Press the refresh button in the upper-left to show the new node. As you can see rostopic echo, shown here.

8.2.4 Using rostopic list

- rostopic list returns a list of all topics currently subscribed to and published.
- Let's figure out what argument the list sub-command needs. In a **new terminal** run:
- **rostopic list -h**

```
murtaza@murtaza-Vostro-15-3568:~$ rostopic list -h
Usage: rostopic list [/namespace]

Options:
  -h, --help            show this help message and exit
  -b BAGFILE, --bag=BAGFILE
                        list topics in .bag file
  -v, --verbose         list full details about each top
ic
  -p                    list only publishers
  -s                    list only subscribers
  --host                group by host name
murtaza@murtaza-Vostro-15-3568:~$
```

- For rostopic list use the **verbose** option:
- rostopic list -v

```
murtaza@murtaza-Vostro-15-3568:~$ rostopic list -v

Published topics:
 * /turtle1/color_sensor [turtlesim/Color] 2 publishers
 * /turtle1/cmd_vel [geometry_msgs/Twist] 1 publisher
 * /rosout [rosgraph_msgs/Log] 3 publishers
 * /rosout_agg [rosgraph_msgs/Log] 1 publisher
 * /turtle1/pose [turtlesim/Pose] 2 publishers

Subscribed topics:
 * /turtle1/cmd_vel [geometry_msgs/Twist] 2 subscribers
 * /rosout [rosgraph_msgs/Log] 1 subscriber
```

- Similarly you can explore other options also.

8.3 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.

8.3.1 Using Rostopic type

- rostopic type returns the message type of any topic being published.
- **rostopic type turtle1/cmd_vel**

```
murtaza@murtaza-Vostro-15-3568:~$ rostopic type /turtle1
/cmd_vel
geometry_msgs/Twist
```

- You should get
  - geometry_msgs/Twist
- We can look at the details of the message using rosmsg:
  - **rosmsg show geometry_msgs/Twist**

```
murtaza@murtaza-Vostro-15-3568:~$ rosmsg show geometry_m
sgs/Twist
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

8.4 rostopic continued

- Now that we have learned about ROS **messages**, let's use rostopic with messages.

8.4.1 Using rostopic pub

- rostopic pub publishes data on to a topic currently advertised.
- Usage:
  - rostopic pub [topic] [msg_type] [args]

- **rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist -- '[2,0,0]' '[0,0,2]'**

```
murtaza@murtaza-Vostro-15-3568:~$ rostopic pub -1 /turtl
e1/cmd_vel geometry_msgs/Twist -- '[2,0,0]' '[0,0,2]'
publishing and latching message for 3.0 seconds
murtaza@murtaza-Vostro-15-3568:~$
```

- The previous command will send a single message to turtlesim telling it to move with an linear velocity of 2, and an angular velocity of 2 .
- This option (dash-one) causes rostopic to only publish one message then exit:
- This is the name of the topic to publish to: **/turtle1/cmd_vel**
- This is the message type to use when publishing to the topic: **geometry_msgs/Twist**
- This option (double-dash) tells the option parser that none of the following arguments is an option. This is required in cases where your arguments have a leading dash -, like negative numbers.
- As noted before, 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, 2.0]' is the angular value with x=0.0, y=0.0, and z=1.8. These arguments are actually in YAML syntax, which is described more in the YAML command line documentation.
- 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 rostopic pub -r command:
- Check out the changes by using -r command: **rostopic pub /turtle1/cmd_vel geometry_msgs/Twist -r 1 -- '[2,0,0]' '[0,0,2]'**

8.4.2 Using rostopic hz

- rostopic hz reports the rate at which data is published.
- Usage:
  - rostopic hz [topic]
- Let's see how fast the turtlesim_node is publishing /turtle1/pose:

- rostopic hz /turtle1/pose

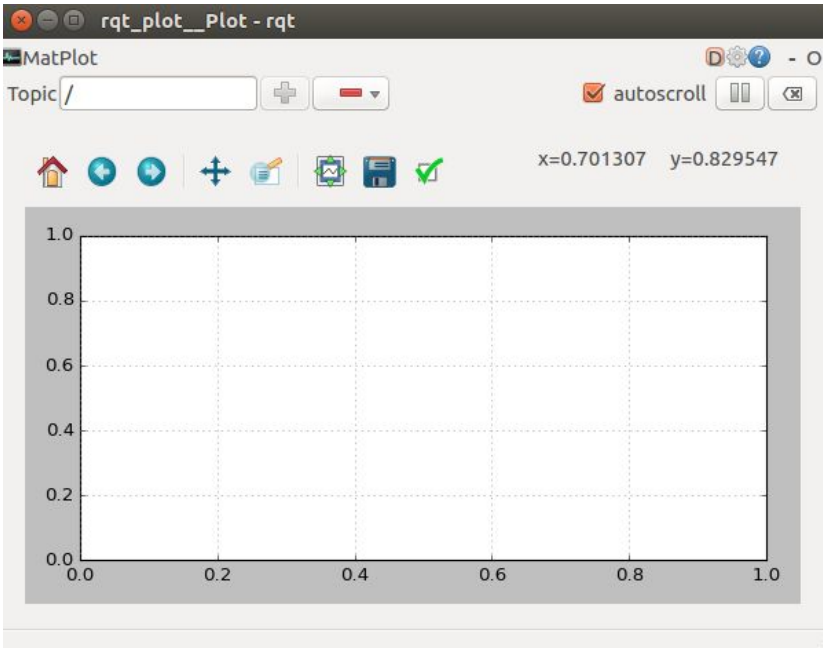
```
murtaza@murtaza-Vostro-15-3568:~$ rostopic hz /turtle
1/pose
subscribed to [/turtle1/pose]
average rate: 62.479
        min: 0.015s max: 0.016s std dev: 0.00048s win
dow: 60
average rate: 62.468
        min: 0.015s max: 0.016s std dev: 0.00048s win
dow: 122
average rate: 62.476
        min: 0.015s max: 0.016s std dev: 0.00048s win
dow: 185
average rate: 62.491
```

- Now we can tell that the turtlesim is publishing data about our turtle at the rate of 62 Hz. We can also use rostopic type in conjunction with rosmsg show to get in depth information about a topic:
    - rostopic type /turtle1/cmd_vel | rosmsg show

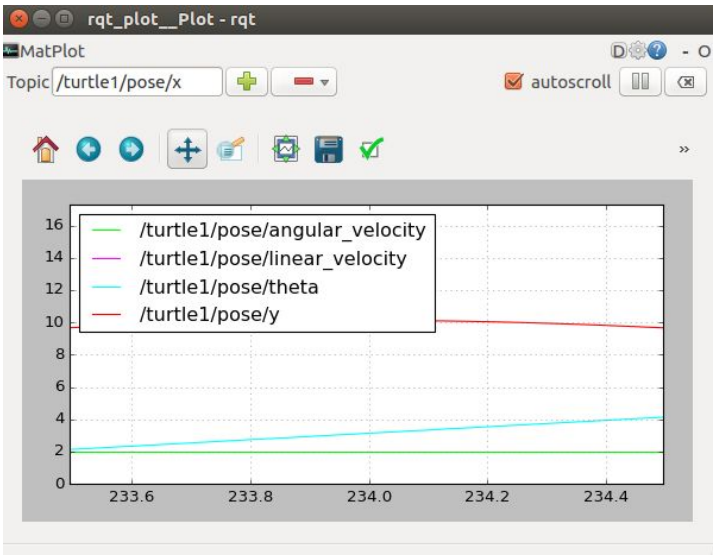
```
murtaza@murtaza-Vostro-15-3568:~$ rostopic type /turt
le1/cmd_vel | rosmsg show
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z

murtaza@murtaza-Vostro-15-3568:~$
```

8.5 Using rqt_plot

- 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. First, start rqt_plot by typing
    - **rosrun rqt_plot rqt_plot**



- In the new window that should pop up, a text box in the upper left corner gives you the ability to add any topic to the plot. Typing /turtle1/pose/x will highlight the plus button, previously disabled. Press it and repeat the same procedure with the topic /turtle1/pose/y. You will now see the turtle's x-y location plotted in the graph.



8.6 ROS parameter and Parameter Server

8.6.1 Parameter Server

- 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. As it is not designed for high-performance, it is best used for static, non-binary data such as configuration parameters. It is meant to be globally viewable so that tools can easily inspect the configuration state of the system and modify if necessary.

8.6.1.1 Parameter

- Parameters are named using the normal ROS naming convention. This means that ROS parameters have a hierarchy that matches the namespaces used for topics and nodes. This hierarchy is meant to protect parameter names from colliding. The hierarchical scheme also allows parameters to be accessed individually or as a tree. For example, for the following parameters:
    - `/camera/left/name: leftcamera`
      `/camera/left/exposure: 1`
      `/camera/right/name: rightcamera`
      `/camera/right/exposure: 1.1`
- The parameter /camera/left/name has the value leftcamera. You can also get the value for /camera/left, which is the dictionary
    - `name: leftcamera`
      `exposure: 1`
- And you can also get the value for /camera, which has a dictionary of dictionaries representation of the parameter tree:
    - `left: { name: leftcamera, exposure: 1 }`
      `right: { name: rightcamera, exposure: 1.1 }`

8.6.2 Using 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. rosparam has many commands that can be used on parameters, as shown below:

```
murtaza@murtaza-Vostro-15-3568:~$ rosparam
rosparam is a command-line tool for getting, setting,
 and deleting parameters from the ROS Parameter Serve
r.

Commands:
        rosparam set    set parameter
        rosparam get    get parameter
        rosparam load   load parameters from file
        rosparam dump   dump parameters to file
        rosparam delete delete parameter
        rosparam list   list parameter names

murtaza@murtaza-Vostro-15-3568:~$
```

8.6.2.1 rosparam list

- Let's look at what parameters are currently on the param server:
- **rosparam list**

```
murtaza@murtaza-Vostro-15-3568:~$ rosparam list
/background_b
/background_g
/background_r
/rosdistro
/roslaunch/uris/host_murtaza_vostro_15_3568__41626
/rosversion
/run_id
murtaza@murtaza-Vostro-15-3568:~$
```

8.6.2.2 rosparam set and rosparam get

- Usage
    - `rosparam set [param_name]`
    - `rosparam get [param_name]`
- Here will change the red channel of the background color:
    - `rosparam set /background_r 150`
- This changes the parameter value, now we have to call the clear service for the parameter change to take effect:

- rosservice call /clear

- Now let's look at the values of other parameters on the param server. Let's get the value of the green background channel:
  - rosparam get /background_g

```
murtaza@murtaza-Vostro-15-3568:~$ rosparam get /backgrou
nd_g
86
```

8.6.2.3 rosparam dump and rosparam load

- You may wish to store this in a file so that you can reload it at another time. This is easy using rosparam:
- Usage
  - rosparam dump [file_name] [namespace]
  - rosparam load [file_name] [namespace]
- Here we write all the parameters to the file params.yaml
  - rosparam dump params.yaml
- You can even load these yaml files into new namespaces, e.g. copy:
  - $ rosparam load params.yaml copy

9. Simple Publisher and Subscriber

- This chapter covers how to write a publisher and subscriber node in python.

9.1 Writing the Publisher Node

- "Node" is the ROS term for an executable that is connected to the ROS network. Here we'll create the publisher ("talker") node which will continually broadcast a message.
- Change directory into the murtaza_ package, you created in the section 6.3 , 'creating a catkin package'
  - **roscd murtaza_**
- If you will get error as below:

Then there might be chances that, you forgot to source the setup file. Source the file by running below command:
source ~/soex_ws/devel/setup.bash

9.1.1 The code

- First lets create a 'scripts' folder to store our Python scripts in:
  - mkdir scripts
  - cd scripts
- Run the below command to open a editor for writing your own node.

- **sudo gedit publisher.py**

```
#!/usr/bin/python

import rospy
from std_msgs.msg import String

def publisher_node():
    rospy.init_node("Publisher_node",anonymous=False)
    pub = rospy.Publisher("String_topic",String,queue_size=10)
    while not rospy.is_shutdown():
        string_data = "This is my first node %s" %rospy.get_time()
        pub.publish(string_data)
        rospy.sleep(2)

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

- Write the code and save the file.
- Make the file executable by running the command:
    - **sudo chmod +x publisher.py**
- Run the node by running below command:
    - **rosrun murtaza_ publisher.py**
- If you get the output on terminal as below:

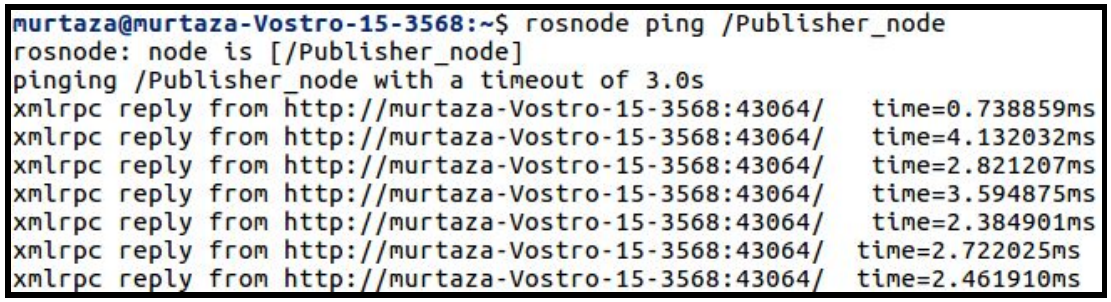
```
murtaza@murtaza-Vostro-15-3568:~/soex_ws/src/murtaza_/scrip
ts$ rosrun murtaza_ publisher.py
Unable to register with master node [http://localhost:11311
]: master may not be running yet. Will keep trying.
```

  Then check whether roscore is running or not, if not then start roscore in a new terminal by running command **roscore**.
- We have studied rostopic and rosnode in previous chapters. Let's test this command on our node and topic.
- **rosnode list**

```
murtaza@murtaza-Vostro-15-3568:~$ rosnode list
/Publisher_node
/rosout
```

- **rosnode info /Publisher_node**

- **rosnode ping /Publisher_node**



- **rostopic list**

- **rostopic echo /String_topic**

```
murtaza@murtaza-Vostro-15-3568:~$ rostopic echo /String_topic
data: "This is my first node 1526492686.23"
---
data: "This is my first node 1526492688.23"
---
data: "This is my first node 1526492690.23"
---
data: "This is my first node 1526492692.23"
```

- **rostopic bw /String_topic**

```
^Cmurtaza@murtaza-Vostro-15-3568:~$ rostopic bw /String_topic
subscribed to [/String_topic]
average: 28.37B/s
        mean: 39.00B min: 39.00B max: 39.00B window: 2
average: 20.80B/s
        mean: 39.00B min: 39.00B max: 39.00B window: 2
average: 24.42B/s
```

- **rostopic hz /String_topic**

```
murtaza@murtaza-Vostro-15-3568:~$ rostopic hz /String_topic
subscribed to [/String_topic]
no new messages
no new messages
average rate: 0.499
        min: 2.002s max: 2.002s std dev: 0.00000s window: 2
no new messages
average rate: 0.500
        min: 2.001s max: 2.002s std dev: 0.00068s window: 3
```

9.1.3 Code explanation

```
1   #!/usr/bin/python
```

Either the usage of #!/usr/bin/env python or #!/usr/bin/python plays a role if the script is executable, and called without the preceding language. The script then calls the language's interpreter to run the code inside the script, and the shebang is the "guide" to find, in your example, python.

Using #!/usr/bin/env python instead of the absolute (full path) #!/usr/bin/python makes sure python (or any other language interpreter) is found, in case it might not be in exactly the same location across different Linux- or Unix -like distributions, as explained e.g. here.

Although #!/usr/bin/python will work on a default Ubuntu system, it is therefore good practice to use #!/usr/bin/env python instead.[source]

```
3   import rospy
4   from std_msgs.msg import String
```

Import rospy imports the rospy client library. rospy is a pure Python client library for ROS. The rospy client API enables Python programmers to quickly interface with ROS Topics, Services, and Parameters. The design of rospy favors implementation speed (i.e. developer time) over runtime performance so that algorithms can be quickly prototyped and tested within ROS. It is also ideal for non-critical-path code, such as configuration and initialization code. Many of the ROS tools are written in rospy to take advantage of the type introspection capabilities. Many of the ROS tools, such as rostopic and rosservice, are built on top of rospy. [source]

from std_msgs.msg import String, imports the String type from package std_msgs.msg. Standard ROS Messages including common message types representing primitive data types and other basic message constructs. std_msgs contains wrappers for ROS primitive types, which are documented in the msg specification. It also contains the Empty type, which is useful for sending an empty signal. However, these types do not convey semantic meaning about their contents: every message simply has a field called "data". Therefore, while the messages in this package can be useful for quick prototyping, they are **NOT intended for "long-term" usage**. For ease of documentation and collaboration, we recommend that existing messages be used, or new messages created, that provide meaningful field name(s).

```
def publisher_node():
```

def is a keyword in python, used to defining a function. publisher _node is the name of function.

```
rospy.init_node("Publisher_node",anonymous=False)
```

One of the first calls you will likely execute in a rospy program is the call to rospy.init_node(), which initializes the ROS node for the process. You can only have one node in a rospy process, so you can only call rospy.init_node() once.

Init_node is a function defined in rospy package. It is taking two arguments, first one is name for the node and second argument is used to keep the node name unique. Use rosnode list and see the name of node, keeping anonymous= true. Edit the file set anonymous = False, you will understand the meaning of this parameter. Publisher_node is the name of topic, it can be of your choice. The anonymous keyword argument is mainly used for nodes where you normally expect many of them to be running and don't care about their names (e.g. tools, GUIs). It adds a random number to the end of your node's name, to make it unique. Unique names are more important for nodes like drivers, where it is an error if more than one is running. If two nodes with the same name are detected on a ROS graph, the older node is shutdown.
This function initializes the node.

```
pub = rospy.Publisher("String_topic",String,queue_size=10)
```

pub is the object of Publisher class. The construction is taking three argument. First argument is the name of topic over which this node is going to publish. Second argument is the type of data, which is going to be published on the topic. So for this case String is the data type of the topic. We can publish any String data. And the third argument is queue_size, this is the size of buffer, means suppose if this node is have published 100 messages, so now not all the 100 messages is present in the buffer but only latest 10 messages will be there.

```
while not rospy.is_shutdown():
```

This is the most common usage patterns for testing for shutdown. Ctrl + c is command line tool you can use to shutdown a running node. rospy.is_shutdown() function monitors for the shutdown signal. It returns false when there is no shutdown signal.

```
string_data = "This is my first node %s" %rospy.get_time()
```

rospy.get_time() Get the current time in float seconds.

```
pub.publish(string_data)
```

pub is the object of Publisher class. publish is a function defined in Publisher class. This function takes message argument to be published, and publishes over topic defined in constructor.

```
rospy.sleep(2)
```

This function is providing delay for two seconds.

9.2 Writing the Subscriber Node

- Change directory into the murtaza_ package, you created in the section 6.3 , 'creating a catkin package'
    - **roscd murtaza_**

9.2.1 The code

- Open a new terminal (Alt + Ctrl + t)
- Run the below command to open a editor for writing your own node.
- **sudo gedit subscriber.py**

```
#!/usr/bin/python

import rospy
from std_msgs.msg import String

def callback(data):
    print data.data

def subscriber_node():
    rospy.init_node("subscriber_node",anonymous=False)
    rospy.Subscriber('String_topic', String , callback)
    rospy.spin()

if __name__ == "__main__":
    subscriber_node()

```