Let's create and build a [catkin workspace](http://wiki.ros.org/catkin/workspaces):

$ mkdir ~/catkin\_ws

$ mkdir ~/catkin\_ws/src

$ cd ~/catkin\_wsd

$ catkin\_make

This will create a catkin workspace (a directory/folder where we can create or modify catkin packages).

Catkin\_make will create a CMakeList.txt in src folder.additionally you will find devel and build folder in catkin\_ws folder.

$ source devel/setup.bash

$ sudo apt-get install ros-kinetic-ros-tutorials

$ catkin\_create\_pkg beginner\_tutorials std\_msgs rospy roscpp

## catkin\_create\_pkg <package\_name> [depend1] [depend2] [depend3] (general format)

This will create a catkin package named beginner\_tutorials which depends on std\_msgs, rospy and roscpp

$ cd ~/catkin\_ws

$ catkin\_make

Build the packages in the catkin workspace

$ source devel/setup.bash

Source the created setup file to add the workspace to ROS environment

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

$ roscore

Roscore is the first thing to run while using ROS

$ rosnode list

Ros node displays info about nodes that are currently working.

$ rosnode info /rosout

This gives info about a particular node ‘rosout’

**Sample output:**

tarun@tarun-XPS-L401X:~/Documents/ROS2/catkin\_ws/src/tutorial$ 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://tarun-XPS-L401X:42387/ ...

Pid: 4643

$ 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

Rosrun allows to use the package name to directly run a node within the package(without having to know the path of package)

Use clrt-C to close the node.

$ rosrun turtlesim turtlesim\_node \_\_name:=my\_turtle

To relaunch the node by changing the name of node from turtlesim\_node to my\_turtle.

$ rosnode ping my\_turtle

To check whether the node is running or not.

$ rosrun turtlesim turtle\_teleop\_key

To drive the turtle around the interface using arrow keys.

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

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

Rqt graph creates dynamic graph of whats going on in the system

$ rosrun rqt\_graph rqt\_graph



$ rostopic -h

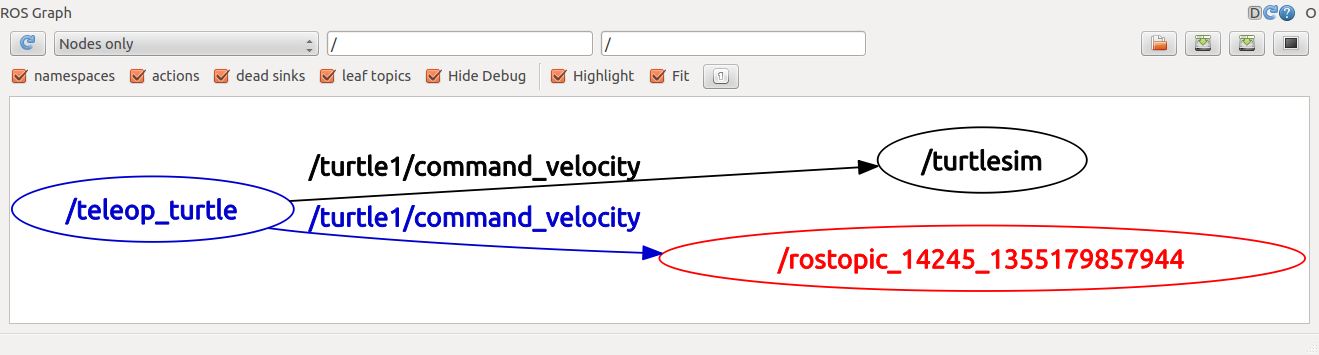
rostopic echo [topic]

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.**

**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 in red, is now also subscribed to the turtle1/command\_velocity topic.**

****

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

Usage: rostopic list [/topic]

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 topic

-p list only publishers

* -s list only subscribers

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

$ rostopic list -v

This displays a verbose list of topics to publish to and subscribe to and their type.

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.

rostopic type returns the message type of any topic being published.

Usage:

rostopic type [topic]

$ rostopic type /turtle1/cmd\_vel

* You should get:  
  geometry\_msgs/Twist

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

$ 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.

Usage:

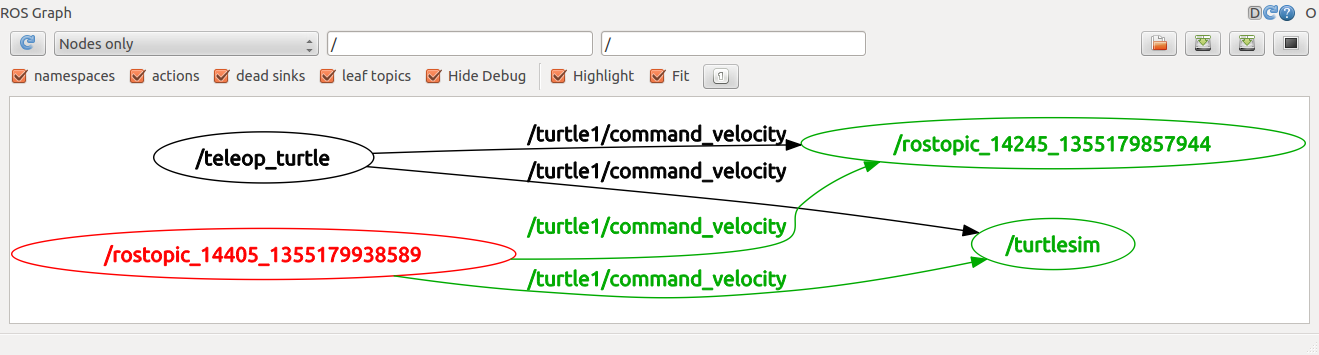
rostopic pub [topic] [msg\_type] [args]

$ rostopic pub -1 /turtle1/cmd\_vel geometry\_msgs/Twist -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'

* This command will publish messages to a given topic:  
  rostopic pub
* This option (dash-one) causes rostopic to only publish one message then exit:  
   -1
* 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, 1.8]' 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](http://wiki.ros.org/ROS/YAMLCommandLine).  
  '[2.0, 0.0, 0.0]' '[0.0, 0.0, 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 rostopic pub -r command:

* $ rostopic pub /turtle1/cmd\_vel geometry\_msgs/Twist -r 1 -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, -1.8]'



As you can see the turtle is running in a continuous circle. In a **new terminal**, we can use rostopic echo to see the data published by our turtlesim:

rostopic echo /turtle1/pose

Let's see how fast the turtlesim\_node is publishing /turtle1/pose:

$ rostopic hz /turtle1/pose

You will see:

subscribed to [/turtle1/pose]

average rate: 59.354

min: 0.005s max: 0.027s std dev: 0.00284s window: 58

average rate: 59.459

min: 0.005s max: 0.027s std dev: 0.00271s window: 118

average rate: 59.539

min: 0.004s max: 0.030s std dev: 0.00339s window: 177

average rate: 59.492

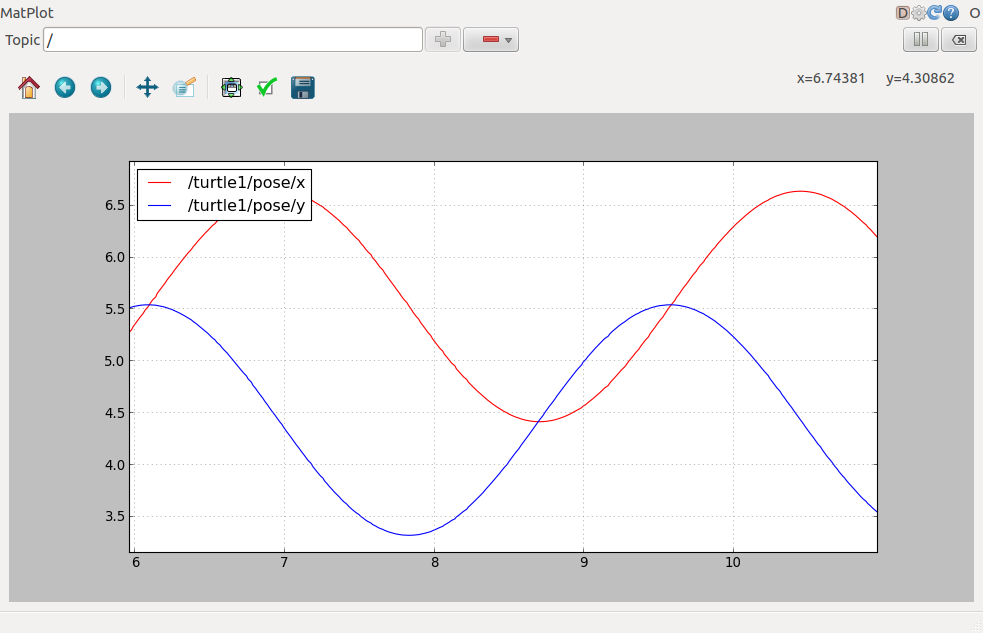
min: 0.004s max: 0.030s std dev: 0.00380s window: 237

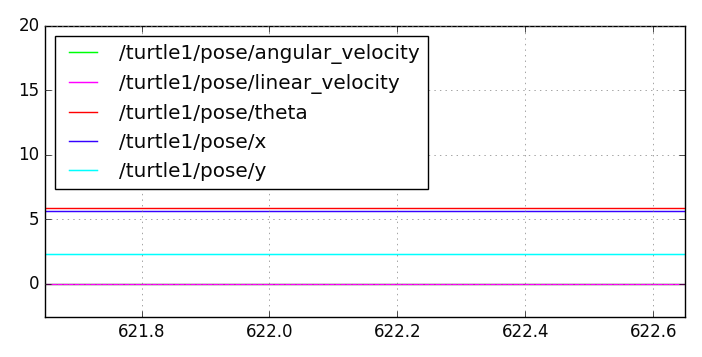
average rate: 59.463

* min: 0.004s max: 0.030s std dev: 0.00380s window: 290

$ rosrun rqt\_plot 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.





## ROS Services

Services are another way that nodes can communicate with each other. Services allow nodes to send a **request** and receive a **response**.

Usage:

rosservice list print information about active services

rosservice call call the service with the provided args

rosservice type print service type

rosservice find find services by service type

rosservice uri print service ROSRPC uri

rosservice call [service] [args]

Here we'll call with no arguments because the service is of type empty:

$ rosservice call /clear

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

$ rosservice type /clear

* std\_srvs/Empty

$ 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

$ rosservice call /spawn 2 2 0.2 ""

name: turtle2

Ros param

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

rosparam set [param\_name]

rosparam get [param\_name]

## Using rqt\_console and rqt\_logger\_level

$ rosrun rqt\_console rqt\_console

$ rosrun rqt\_logger\_level rqt\_logger\_level

$ rosrun turtlesim turtlesim\_node

$ rostopic pub /turtle1/cmd\_vel geometry\_msgs/Twist -r 1 -- '[2.0,0.0,0.0]' '[0.0,0.0,0.0]'

### Using roslaunch

$ roscd beginner\_tutorials

$ mkdir launch

$ cd launch

Now let's create a launch file called turtlemimic.launch and paste the following:

<launch>

<group ns="turtlesim1">

<node pkg="turtlesim" name="sim" type="turtlesim\_node"/>

</group>

<group ns="turtlesim2">

<node pkg="turtlesim" name="sim" type="turtlesim\_node"/>

</group>

<node pkg="turtlesim" name="mimic" type="mimic">

<remap from="input" to="turtlesim1/turtle1"/>

<remap from="output" to="turtlesim2/turtle1"/>

</node>

</launch>

$ roslaunch beginner\_tutorials turtlemimic.launch

Two turtlesims will start and in a **new terminal** send the rostopic command:

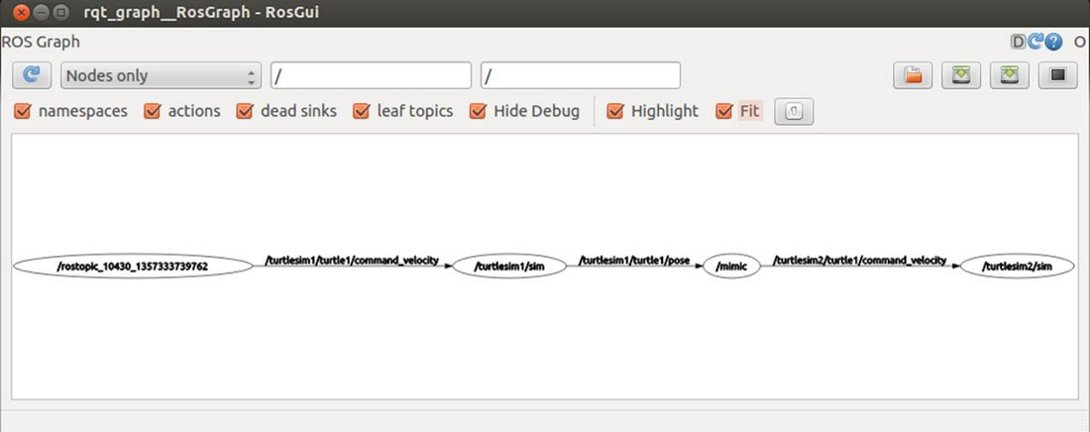
* $ rostopic pub /turtlesim1/turtle1/cmd\_vel geometry\_msgs/Twist -r 1 -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, -1.8]'

We can also use [rqt\_graph](http://wiki.ros.org/rqt_graph) to better understand what our launch file did. Run [rqt](http://wiki.ros.org/rqt)'s main window and select *Plugins > Introspection > Node Graph*:

$ rqt

Or simply:

$ rqt\_graph



## 

## Introduction to msg and srv

<http://wiki.ros.org/ROS/Tutorials/CreatingMsgAndSrv>

INTERMEDIATE

# Creating a ROS package by hand.

<http://wiki.ros.org/ROS/Tutorials/Creating%20a%20Package%20by%20Hand>

# Managing System dependencies

ROS packages sometimes require external libraries and tools that must be provided by the operating system. These required libraries and tools are commonly referred to as *system dependencies*.

### rosdep

rosdep is a tool you can use to install system dependencies required by ROS packages.

Usage:

rosdep install [package]

Download and install the system dependencies for turtlesim:

$ rosdep install turtlesim

$ rosdep update

and now that dependency will be resolved by rosdep.

You can test it with :

$ rosdep resolve my\_dependency\_name

SNAP

A snap is a bundle of your app and its dependencies that works without modification across many different Linux distributions.[Snapcraft](https://snapcraft.io/snapcraft) is a powerful and easy to use command line tool for building [snaps](https://docs.snapcraft.io/t/getting-started/3876).

ACTION LIBRARY

First create a catkin workspace using following commands:

Mkdir catkin\_ws\_…. Or mkdir …..

Cd catkin\_ws\_…. Or cd …

Mkdir src

Catkin\_make

Now make actionlib package

Catkin\_create\_pkg [package name] [depend 1] [depend 2] [depend 3]

Catkin\_create\_pkg actionlib\_tutorials actionlib message\_generation roscpp rospy std\_msgs actionlib\_msgs

Creating the Action Message

First create a file named Fibonacci.action and place it in actionlib\_tutorials/action/Fibonacci.action

And add the following info in it:

#goal definition

int32 order

---

#result definition

Int32[ ] sequence

---

#feedback

Int32[ ] sequence

Also some changes are to be made in CMakeList.txt to automatically generate messages on using catkin\_make

find\_package(catkin REQUIRED COMPONENTS actionlib\_msgs)

add\_action\_files(

DIRECTORY action

FILES Fibonacci.action

)

Comment the other two action files under add\_action\_files (Action1.action , Action2.action)

generate\_messages(

DEPENDENCIES actionlib\_msgs std\_msgs # Or other packages containing msgs

)

catkin\_package(

CATKIN\_DEPENDS actionlib\_msgs

)

And following addition in packge.xml file

<exec\_depend>message\_generation</exec\_depend>

WRITING a SIMPLE SERVER

**Code ….from website**

**FOR SERVER:**

**Add the following lines to your CMakeLists.txt file:**

add\_executable(fibonacci\_server src/fibonacci\_server.cpp)

target\_link\_libraries(

fibonacci\_server

${catkin\_LIBRARIES}

)

add\_dependencies(

fibonacci\_server

${actionlib\_tutorials\_EXPORTED\_TARGETS}

)

**The minimal entire CMakeLists.txt would look like this:**

cmake\_minimum\_required(VERSION 2.8.3)

project(actionlib\_tutorials)

find\_package(catkin REQUIRED COMPONENTS roscpp actionlib actionlib\_msgs)

find\_package(Boost REQUIRED COMPONENTS system)

add\_action\_files(

DIRECTORY action

FILES Fibonacci.action

)

generate\_messages(

DEPENDENCIES actionlib\_msgs std\_msgs

)

catkin\_package(

CATKIN\_DEPENDS actionlib\_msgs

)

include\_directories(include ${catkin\_INCLUDE\_DIRS} ${Boost\_INCLUDE\_DIRS})

add\_executable(fibonacci\_server src/fibonacci\_server.cpp)

target\_link\_libraries(

fibonacci\_server

${catkin\_LIBRARIES}

)

add\_dependencies(

fibonacci\_server

${actionlib\_tutorials\_EXPORTED\_TARGETS}

)

**Keep in mind the order** of **add\_executable** and **add\_dependencies**

**FOR CLIENT**