ROS TUTORIALS

Ros Service: A service that your robot provide

CREATE WORKSPACE:

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/
$ catkin_make
```

Catkin_make is a command just like the command “make” when every code and all other relevant things are in the same directory then you can use “make” command but else you have to use catkin_make command.

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

Src	build	devel
The source space contains the Source code.this is where you can clone create and edit source code For the packages you want to build	This is where CMake is involved to build the packages in the source space	the development space or devel is where built target are placed

You can check your workspace path by using this command:

```
$ echo $ROS_PACKAGE_PATH
```

You will find something like that:

```
/home/youruser/catkin_ws/src:/opt/ros/kinetic/share
```

ROS PACKAGES:

A catkin Package should contains two things

1.package.xml file provides meta information about the package.
2.CMakeList.txt

And Each package must have its own folder .

If you want to create your own packages first got to your src directory
By $cd ~/catkin_ws/src

Then type

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

This will create a beginner_tutorials folder which contains a package.xml and a CMakeLists.txt

Here std_msgs rospy roscpp are the 1st order dependence of the package

Next you have to build the the packages in the catkin workspace by this command:

```
$ cd ~/catkin_ws
$ catkin_make
```

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

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

DEPEDENCES:

1. Roscpp rospy std_msgs this are the 1st order dependences
   You can check 1st order dependences by typing :
   ```
   $ rospack depends1 beginner_tutorials
   ```

2. In many cases, a dependency will also have its own dependencies. For instance, rospy has other dependencies.
   Like this one
   genpy
   roscpp
   rosgraph
   rosgraph_msgs
   roslib
   std_msgs

You have to add the dependence into the package.xml like package.xml,;

ROSNODES:

Nodes are single purpose executable program
Individually compiled executed and managed
Organized in package

Topic:

Topic is a stream of messages that are transferred between nodes once the nodes are registered with the rosmaster then the node 1 can be publisher on that topic and node 2 could be act as subscriber on that topic thus messages flow from subscriber and publisher.

```
                    Rosmaster
                        |
  registered            | registered
  ----------------------------------------------
  |                                            |
Node 1                                       node 2
  |                                            |
  |publisher                  subscriber       |
  --------------------Topic---------------------
```

1st you have to run ROSMASTER it will act as a server by typing: $ roscore

Using rosnode

$ rosnode list
Will give you numbered of node connected to the master
$ rosnode info [/name of the node]
Will give you information about that node.

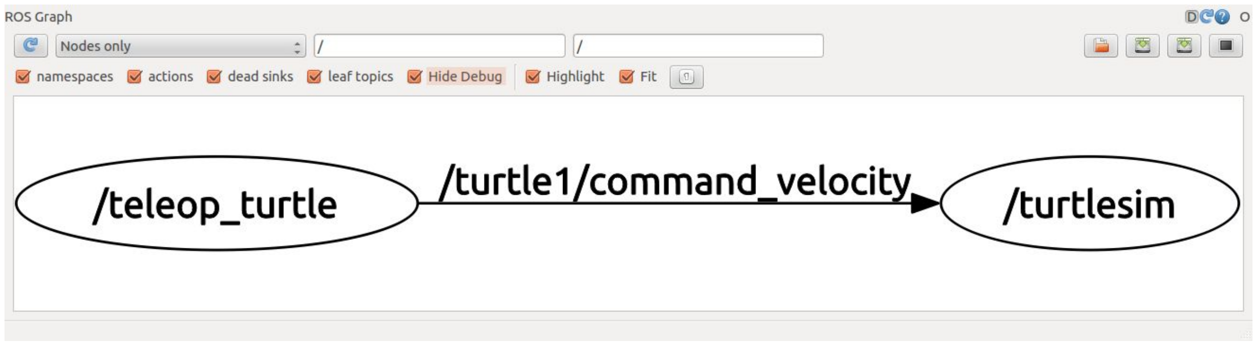
Using rosrun

rosrun allows you to use the package name to directly run a node within a package (without having to know the package path).
$ rosrun [package_name] [node_name]
Ie :$ rosrun turtlesim turtlesim_node
So now turtlesim_node is also connected with rosmaster.

ROS Topics

$ rosrun turtlesim turtlesim_node
$ rosrun turtlesim turtle_teleop_key
The turtlesim_node and the turtle_teleop_key node are communicating with each other over a ROS **Topic**.

RQT GRAPH:
Through rqt_graph you will get a clear picture about the total things that what is happening .
Go to the directory and type:
$ rosrun rqt_graph rqt_graph
You will get something like that



you can use the help option to get the available sub-commands for rostopic
By $ rostopic -h

Then $ rostopic echo [topic]
This command enable us the data published on that topic
Just like in this command you will get to know the linear and rotational motion
Ie: $ rostopic echo /turtle1/cmd_vel

ROS Messages:

The nodes comminute each other through msgs
So the topic type is defined through message type and the type of the messages send through topics are known through :

```
        $ rostopic type [topic]
Ie      $ 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
```

Using rostopic pub

rostopic pub publishes data on to a topic currently advertised.
rostopic pub [topic] [msg_type] [args]
Just an example :

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

Here -1 signifies that rostopic will publish one message than exit;
“/turtle1/cmd_vel” is the topic name
“geometry_msgs/Twist‘ is the message type
2.0 -velocity in x direction
0.0,0.0 are velocity in the y and z direction
And 1.8 is the angular velocity in the z direction;
So without using turtle_tople_key by using this you can control the turtle;

And if you want to rotate the turtle i circle type:

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

ROS Services

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

rosservice 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 type

```
$ rosservice type [service]
$ rosservice type /clear    --------this command will clear all the lines in the background;
```

rosservice call

```
$ rosservice call /spawn 2 2 0.2 ""    ----------------it will make a new turtle on the same screen
and it will be denoted as turtle2
```

rosparam

```
$ rosparam list
```

Through this one could get all the ros_parameter list

```
/background_b
/background_g
/background_r
/rosdistro
/roslaunch/uris/host_57aea0986fef__34309
/rosversion
/run_id
```

Usage:

```
rosparam set [param_name]
rosparam get [param_name]
```

Through this we can change the background colour
$ rosparam set /background_r 150
now we have to call the clear service for the parameter change to take effect:
$ rosservice call /clear
Through this command we can get to know about the colour
$ rosparam get /background_g

ROS msg and srv:

- ***msg**: msg files are simple text files that describe the fields of a ROS message. They are used to generate source code for messages in different languages.*
- ***srv**: an srv file describes a service. It is composed of two parts: a request and a response.*

msg files are stored in the msg directory of a package, and srv files are stored in the srv directory.
You have to create an msg file in beginner tutorials

Just like :
```
$ roscd beginner_tutorials
$ mkdir msg
$ echo "int64 num" > msg/Num.msg
```

And we have to add dependencies in the CMake list file
In CMakeList we have to add also   Num.msg(The name of the .msg file)
```
# add_message_files(
#   FILES
#   Message1.msg
#   Message2.msg
# )
```

And in the package.xml file we have to add :

```
<build_depend>message_generation</build_depend>
  <exec_depend>message_runtime</exec_depend>
```

Writing the Publisher Node

We have to Create a src directory in the beginner_tutorials package directory:
By typing : $ mkdir -p src
In that just write a talker.cpp code:

```
#include "ros/ros.h"                //header necessary to  use most common public spiece in ros
 #include "std_msgs/String.h"           //This includes the std_msgs/String message,

 #include <sstream>

int main(int argc, char **argv)
 {
ros::init(argc, argv, "talker");   // nodes name must be unique
ros::NodeHandle n;    //Create a handle to this process' node.
ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
/*Tell the master that we are going to be publishing a message of type std_msgs/String on the
topic chatter. This lets the master tell any nodes listening on chatter that we are going to publish
data on that topic.The second argument is the size of our publishing queue.*/


  ros::Rate loop_rate(10);       //allows you to specify a frequency that you would like to loop at.
int count = 0;
 while (ros::ok())         // here it means if you press ctrl+c the loop will be ended
{
     std_msgs::String msg;
   std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();

   ROS_INFO("%s", msg.data.c_str());
 chatter_pub.publish(msg);     //Now we actually broadcast the message to anyone connected
   ros::spinOnce();       //if you get callback then you have to do this operaton
  loop_rate.sleep();
   ++count;
 }
 return 0;
 }
```

Writing the Subscriber Node

Create the src/listener.cpp file within the beginner_tutorials package;

```
#include "ros/ros.h"
 #include "std_msgs/String.h"

 void chatterCallback(const std_msgs::String::ConstPtr& msg)
 {
  ROS_INFO("I heard: [%s]", msg->data.c_str());
 }

 int main(int argc, char **argv)
 {
  -   * The ros::init() function needs to see argc and argv so that it can perform
   * any ROS arguments and name remapping that were provided at the command line.
 * For programmatic remappings you can use a different version of init() which takes
 * remappings directly, but for most command-line programs, passing argc and argv is
* the easiest way to do it.  The third argument to init() is the name of the node.
  *
   * You must call one of the versions of ros::init() before using any other
* part of the ROS system.
   */
 ros::init(argc, argv, "listener");

  /**
   * NodeHandle is the main access point to communications with the ROS system.
   * The first NodeHandle constructed will fully initialize this node, and the last
   * NodeHandle destructed will close down the node.
   */
  ros::NodeHandle n;

  /**
   * The subscribe() call is how you tell ROS that you want to receive messages
  * on a given topic.  This invokes a call to the ROS
   * master node, which keeps a registry of who is publishing and who
   * is subscribing.  Messages are passed to a callback function, here
   * called chatterCallback.  subscribe() returns a Subscriber object that you
   * must hold on to until you want to unsubscribe.  When all copies of the Subscriber
   * object go out of scope, this callback will automatically be unsubscribed from
   * this topic.
   *
   * The second parameter to the subscribe() function is the size of the message
   * queue.  If messages are arriving faster than they are being processed, this
   * is the number of messages that will be buffered up before beginning to throw
   * away the oldest ones.
   */
  ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);

  /**
   * ros::spin() will enter a loop, pumping callbacks.  With this version, all
   * callbacks will be called from within this thread (the main one).  ros::spin()
  * will exit when Ctrl-C is pressed, or the node is shutdown by the master.
   */
  ros::spin();

  return 0;
}
```

This is the callback function that will get called when a new message has arrived on the chatter topi
You have to change according to it.

Running ROS across multiple machines

Deploying multiple things into ros system is easy you have to keep this things in your mind:

*. You only need one master. Select one machine to run it on.
- All nodes must be configured to use the same master, via ROS_MASTER_URI.
- There must be complete, bi-directional connectivity between all pairs of machines, on all ports (see ROS/NetworkSetup).
- Each machine must advertise itself by a name that all other machines can resolve (see ROS/NetworkSetup).

Let us assume that there are two machine we have to run the talker and listener.cpp/py in two machine let's called marvin and hal

1. We have to start the rosmaster.
2. Then start any of them at any of two machines;

Ie,

```
$ roscore

$ ssh hal

$ export ROS_MASTER_URI=http://hal:11311

$ rosrun rospy_tutorials listener.py

And

$ ssh marvin

$ export ROS_MASTER_URI=http://hal:11311

$ rosrun rospy_tutorials talker.py
```

Defining custom messages:

*Defi*ning a custom message is quite easy just place .msg file inside the msg folder.

And have to add the dependencies in the package.xml file

Just like it:

<build_depend>name_of_package_containing_custom_msg</build_depend>

<exec_depend>name_of_package_containing_custom_msg</exec_depend>

And in the CMakeList.file you have to do the same;