ROS Task Documentation
- Swaminathan S K (AGV Software Team)

Resources:
http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment
http://wiki.ros.org/ROS/Tutorials/NavigatingTheFilesystem

Filesystem Tools:

rospack:

```
$ rospack find [package_name]
```

Example:

```
$ rospack find roscpp
```

- Code is spread across many ROS packages. Navigating with command-line tools such as ls and cd can be very tedious which is why ROS provides tools to help you.

```
swaminathan@swaminathan-ASUS-TUF-Dash-F15-FX517ZM-FX517ZM:~/catkin_ws$ rospack find roscpp
/opt/ros/noetic/share/roscpp
```

roscd:

```
$ roscd <package-or-stack>[/subdir]
```

Example:

```
$ roscd roscpp
```

- roscd is part of the rosbash suite. It allows you to change directory (cd) directly to a package or a stack.

```
swaminathan@swaminathan-ASUS-TUF-Dash-F15-FX517ZM-FX517ZM:~/catkin_ws$ roscd roscpp
swaminathan@swaminathan-ASUS-TUF-Dash-F15-FX517ZM-FX517ZM:/opt/ros/noetic/share/roscpp$ pwd
/opt/ros/noetic/share/roscpp
```

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
```

rosls:

```
$ rosls <package-or-stack>[/subdir]
```

Example:

```
$ rosls roscpp_tutorials
```

- rosls is part of the rosbash suite. It allows you to ls directly in a package by name rather than by absolute path.

```
swamlnathan@swamlnathan-ASUS-TUF-Dash-F15-FX517ZM-FX517ZM:/opt/ros/noetic/share/roscpp$ rosls r
oscpp_tutorials
cmake  launch  package.xml  srv
```

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`:

```
turtle_actionlib/  turtlesim/        turtle_tf/
```

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:

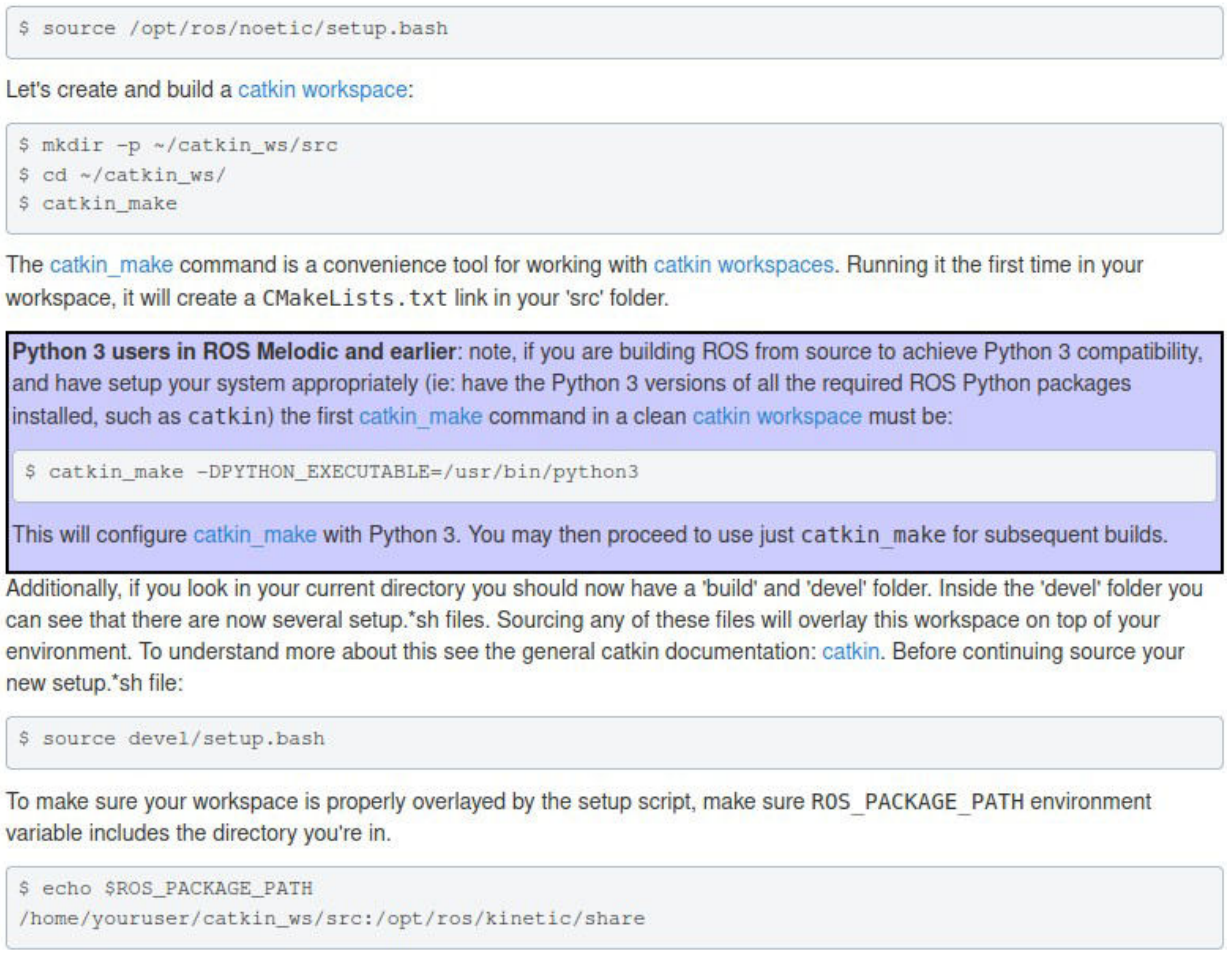
```
$ rosls <<< now push the TAB key twice >>>
```

Creating a ROS package:

http://wiki.ros.org/ROS/Tutorials/CreatingPackage

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.



```
$ source /opt/ros/noetic/setup.bash
```

Let's create and build a catkin workspace:

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

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.

Python 3 users in ROS Melodic and earlier: note, if you are building ROS from source to achieve Python 3 compatibility, and have setup your system appropriately (ie: have the Python 3 versions of all the required ROS Python packages installed, such as `catkin`) the first catkin_make command in a clean catkin workspace must be:

```
$ catkin_make -DPYTHON_EXECUTABLE=/usr/bin/python3
```

This will configure catkin_make with Python 3. You may then proceed to use just `catkin_make` for subsequent builds.

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

First change to the source space directory of the catkin workspace you created in the Creating a Workspace for catkin tutorial:

Now use the `catkin_create_pkg` script to create a new package called 'beginner_tutorials' which depends on std_msgs, roscpp, and rospy:

```
swaminathan@swaminathan-ASUS-TUF-Dash-F15-FX517ZM-FX517ZM:~$ cd catkin_ws
swaminathan@swaminathan-ASUS-TUF-Dash-F15-FX517ZM-FX517ZM:~/catkin_ws$ ls
build  devel  src
swaminathan@swaminathan-ASUS-TUF-Dash-F15-FX517ZM-FX517ZM:~/catkin_ws$ cd src
swaminathan@swaminathan-ASUS-TUF-Dash-F15-FX517ZM-FX517ZM:~/catkin_ws/src$ catkin_create_pkg be
ginner_tutorials std_msgs rospy roscpp
Created file beginner_tutorials/package.xml
Created file beginner_tutorials/CMakeLists.txt
Created folder beginner_tutorials/include/beginner_tutorials
Created folder beginner_tutorials/src
Successfully created files in /home/swaminathan/catkin_ws/src/beginner_tutorials. Please adjust
 the values in package.xml.
```

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

```
# This is an example, do not try to run this
# catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
```

4. Building a catkin workspace and sourcing the setup file

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

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

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

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

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

First Order Dependencies:

We provided a few dependencies earlier. These can be checked for by using rospack depends1 {package-name}

```
swaminathan@swaminathan-ASUS-TUF-Dash-F15-FX517ZM-FX517ZM:~/catkin_ws$ rospack depends1 beginne
r_tutorials
roscpp
rospy
std_msgs
```

Note: in the package.xml file, we have the dependencies tags:
dependencies tags

The next set of tags describe the dependencies of your package. The dependencies are split into build_depend, buildtool_depend, exec_depend, test_depend. Since we passed std_msgs, roscpp, and rospy as arguments to catkin_create_pkg, the dependencies will look like this:

Toggle line numbers

```
27  <!-- The *_depend tags are used to specify dependencies -->
28  <!-- Dependencies can be catkin packages or system dependencies -->
29  <!-- Examples: -->
30  <!-- Use build_depend for packages you need at compile time: -->
31  <!--   <build_depend>genmsg</build_depend> -->
32  <!-- Use buildtool_depend for build tool packages: -->
33  <!--   <buildtool_depend>catkin</buildtool_depend> -->
34  <!-- Use exec_depend for packages you need at runtime: -->
35  <!--   <exec_depend>python-yaml</exec_depend> -->
36  <!-- Use test_depend for packages you need only for testing: -->
37  <!--   <test_depend>gtest</test_depend> -->
38  <buildtool_depend>catkin</buildtool_depend>
39  <build_depend>roscpp</build_depend>
40  <build_depend>rospy</build_depend>
41  <build_depend>std_msgs</build_depend>
```

Building Packages:

```
# In a catkin workspace
$ catkin_make
$ catkin_make install  # (optionally)
```

The above commands will build any catkin projects found in the `src` folder. This follows the recommendations set by REP128. If your source code is in a different place, say `my_src` then you would call catkin_make like this:

Note: If you run the below commands it will not work, as the directory `my_src` does not exist.

```
# In a catkin workspace
$ catkin_make --source my_src
$ catkin_make install --source my_src  # (optionally)
```

ROS Nodes:
http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes
https://www.youtube.com/watch?v=aL7zLnaEdAg

Some terms and definitions:

- 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)

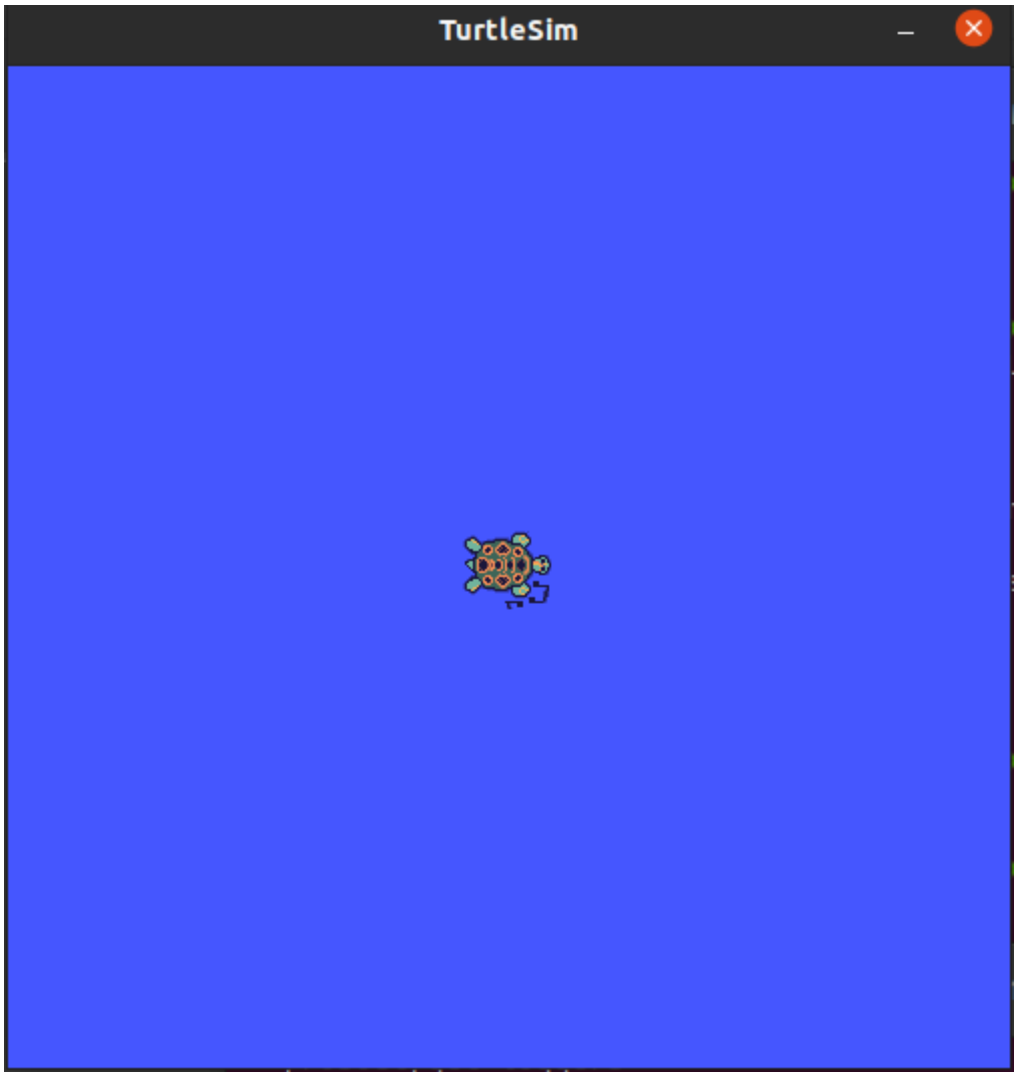
roscore -> creates the master and a node called /rosout. After initializing open a **new terminal** and type:

```
$ rosnode list
```

This will list the present nodes. Running a process or a node:

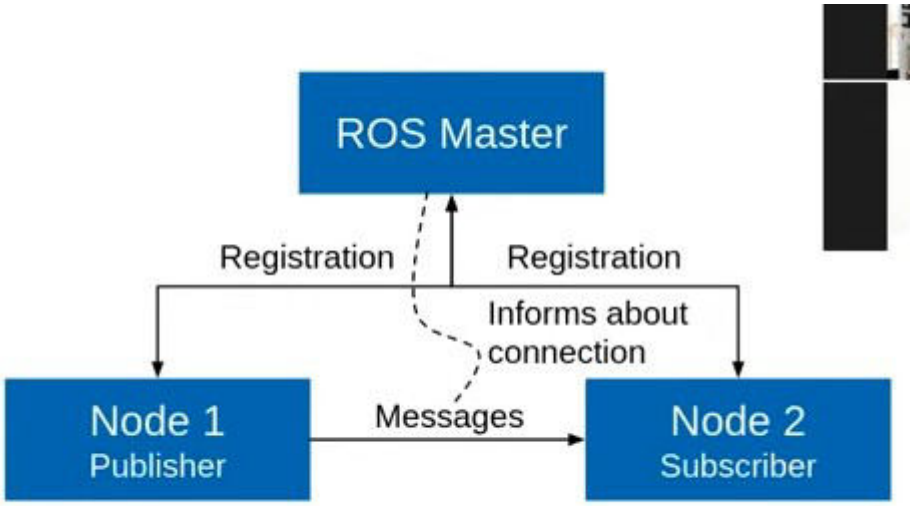
```
$ rosrun turtlesim turtlesim_node
```

This opens a new window to display the turtlesim (which is again a node). Now, typing rosnode list in the terminal would give two nodes. We can close the process or the node by closing the window or using Ctrl + C in terminal.



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

Some more details:

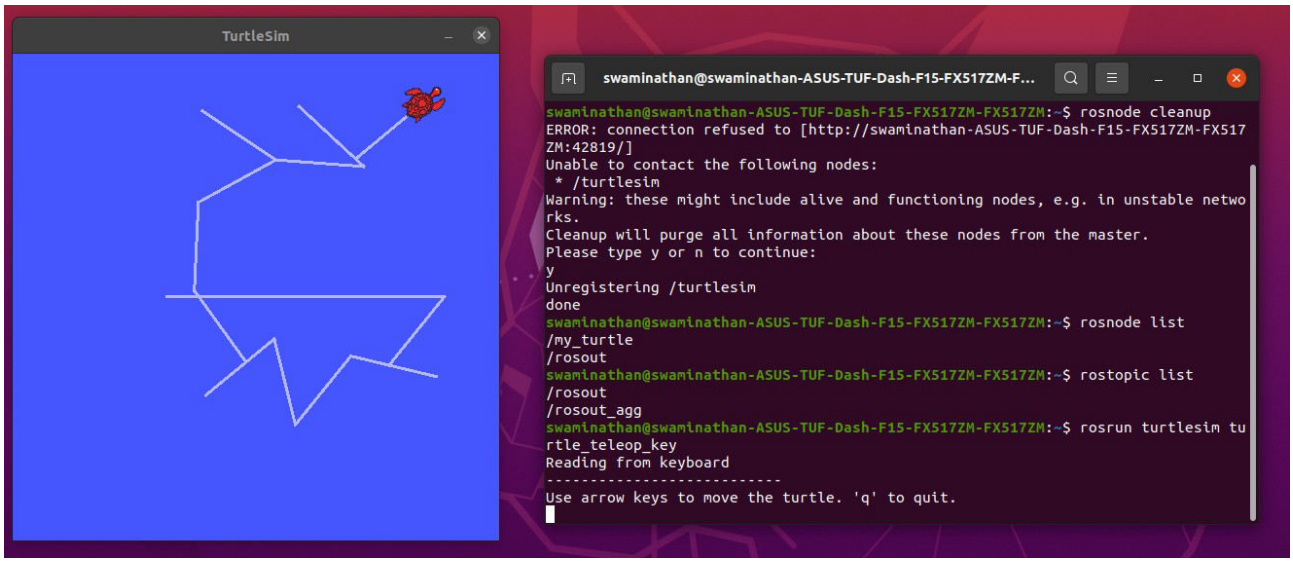


Nodes communicate over topics. Nodes can be a subscriber or publisher. Topic is a name for a stream of messages.

Understanding ROS Topics:
http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics

Run in Separate Terminals:

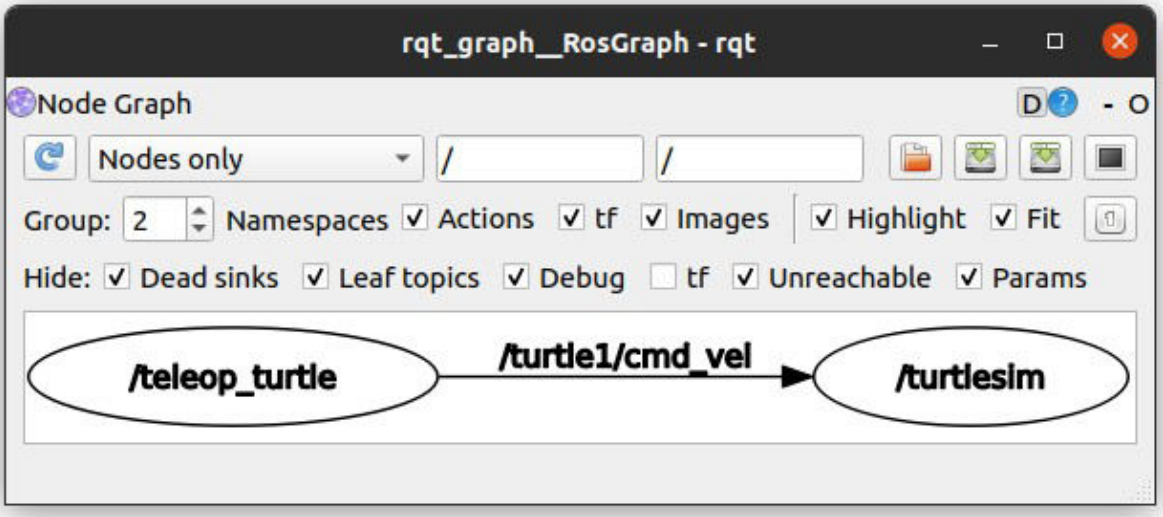
```
$ rosrun turtlesim turtlesim_node
$ rosrun turtlesim turtle_teleop_key

These are both different nodes
```



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.

Visualizing using rqt_graph:

```
$ rosrun rqt_graph rqt_graph
```

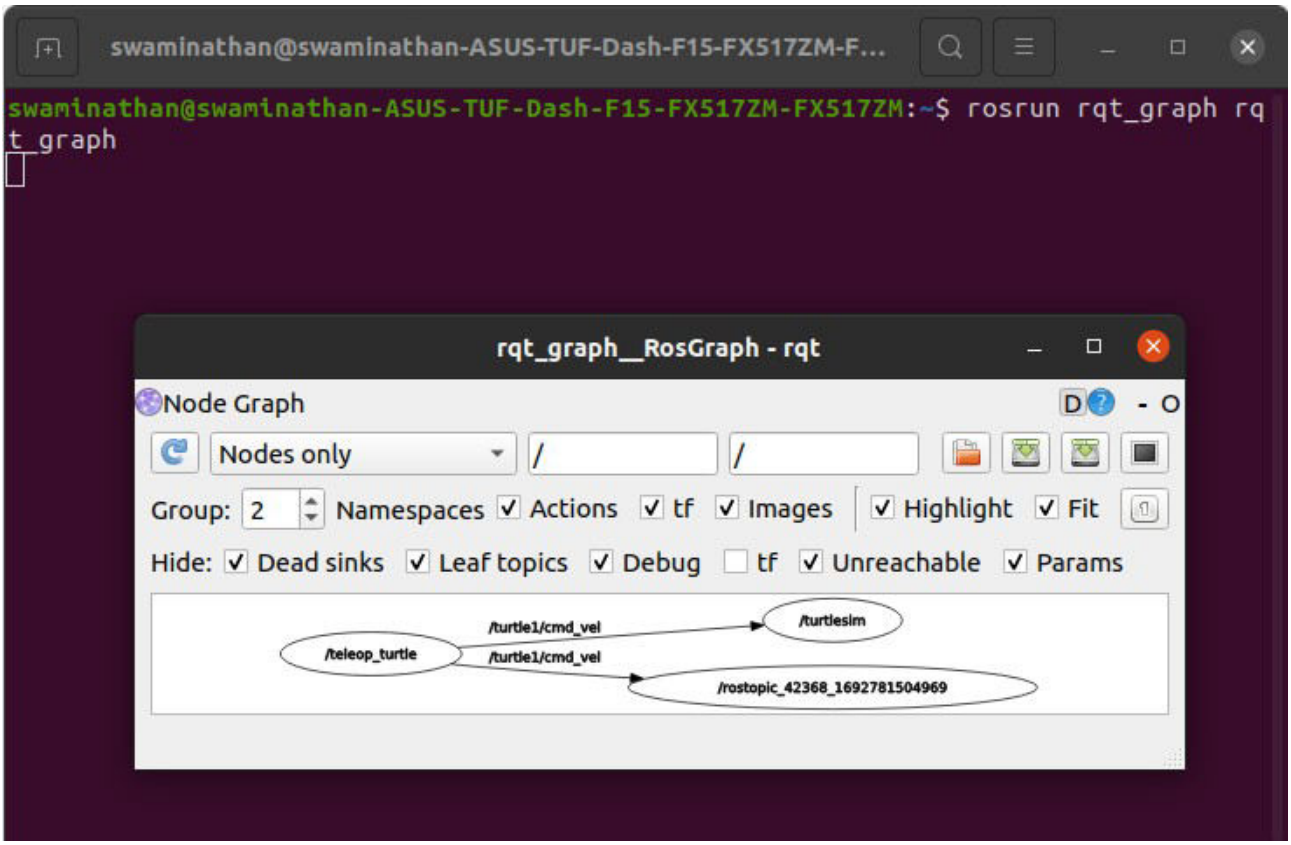


The turtlesim_node and the turtle_teleop_key nodes are communicating on the topic named /turtle1/command_velocity.

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

`rostopic echo` shows the data published on a topic.
`$ rostopic echo /turtle1/cmd_vel`

We wont see anything unless the keys are recorded or any messages are published.



Here the rostopic echo node can also be seen. (Conclusion that rostopic echo also creates a new node).

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.

For ROS Hydro and later,

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

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

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

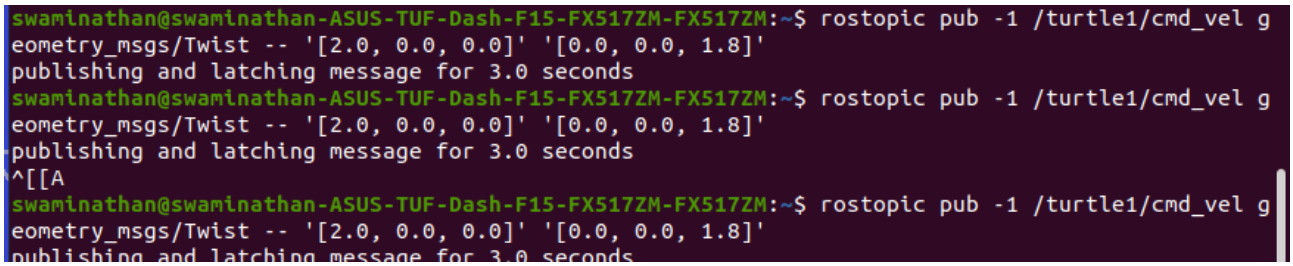
published on it. The type of the message sent on a topic can be determined using rostopic type.

```
swaminathan@swaminathan-ASUS-TUF-Dash-F15-FX517ZM-FX517ZM:~$ rostopic type /turtle1/cmd_vel
geometry_msgs/Twist
swaminathan@swaminathan-ASUS-TUF-Dash-F15-FX517ZM-FX517ZM:~$ 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
```

Publishing to a Topic

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

```
$ rostopic pub [topic] [msg_type] [args]
```



- 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.

```
'[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:

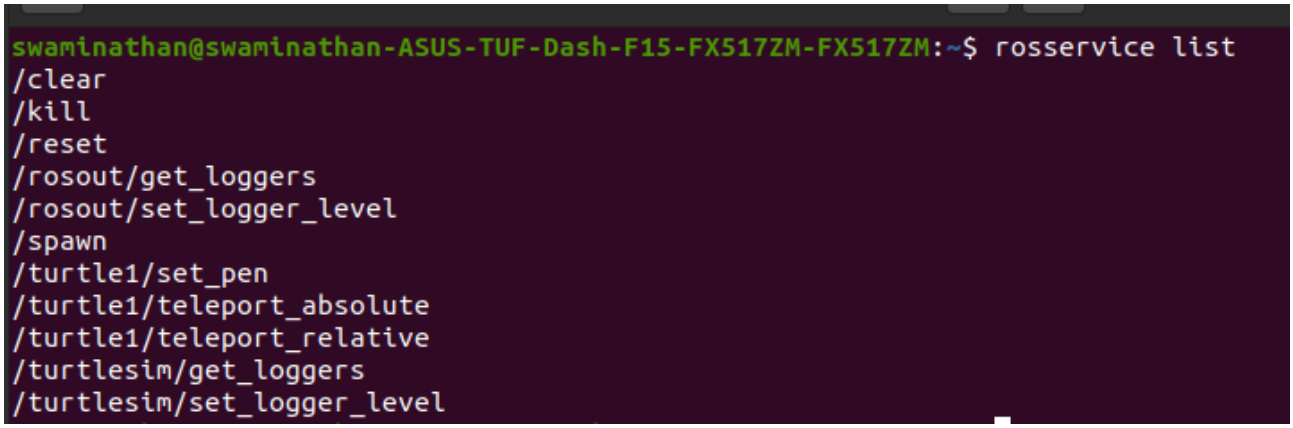
```
$ rostopic pub /turtle1/cmd_vel geometry_msgs/Twist -
r 1 -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, -1.8]'
```

Understanding ROS services and parameters:
http://wiki.ros.org/ROS/Tutorials/UnderstandingServicesParams

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
```



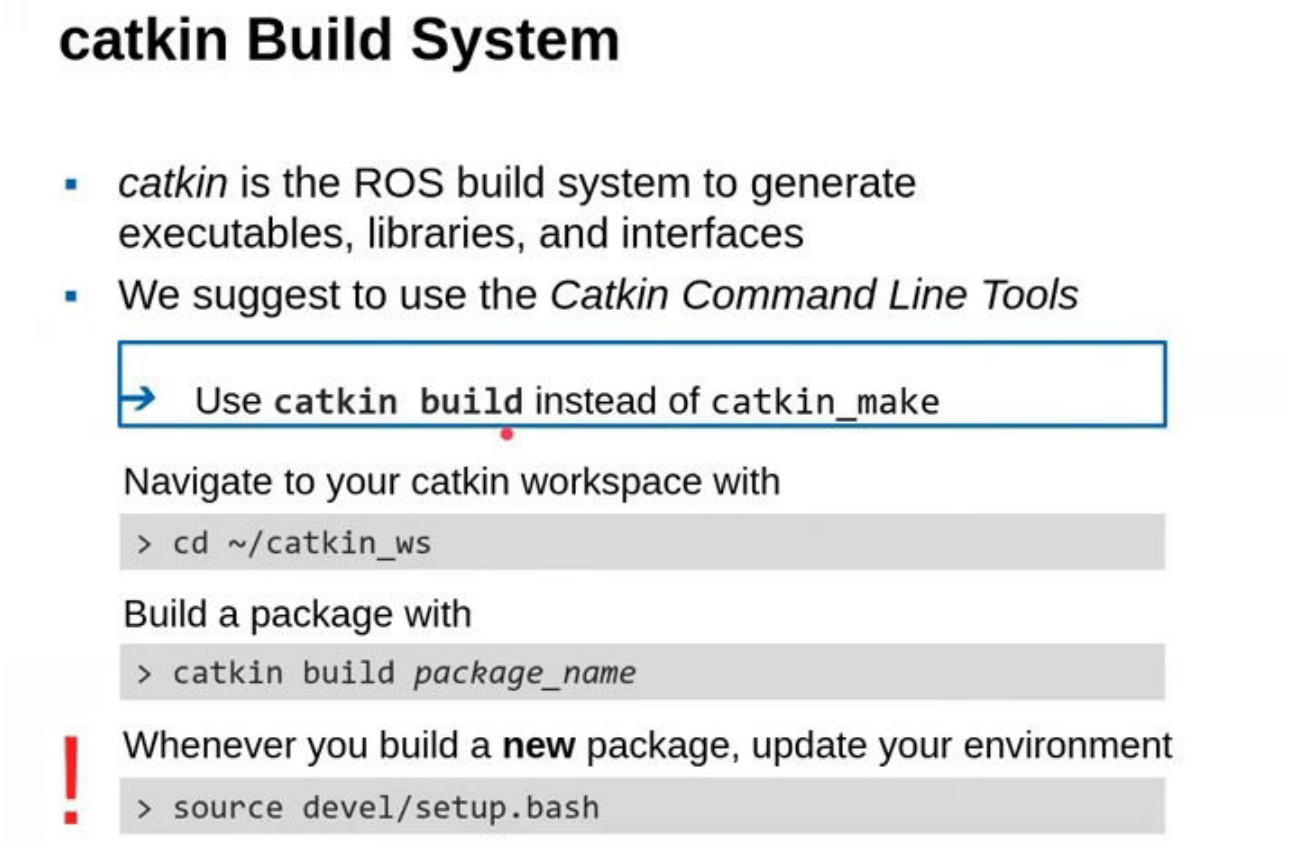

```
rosservice call [service] [args]
```

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 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
```

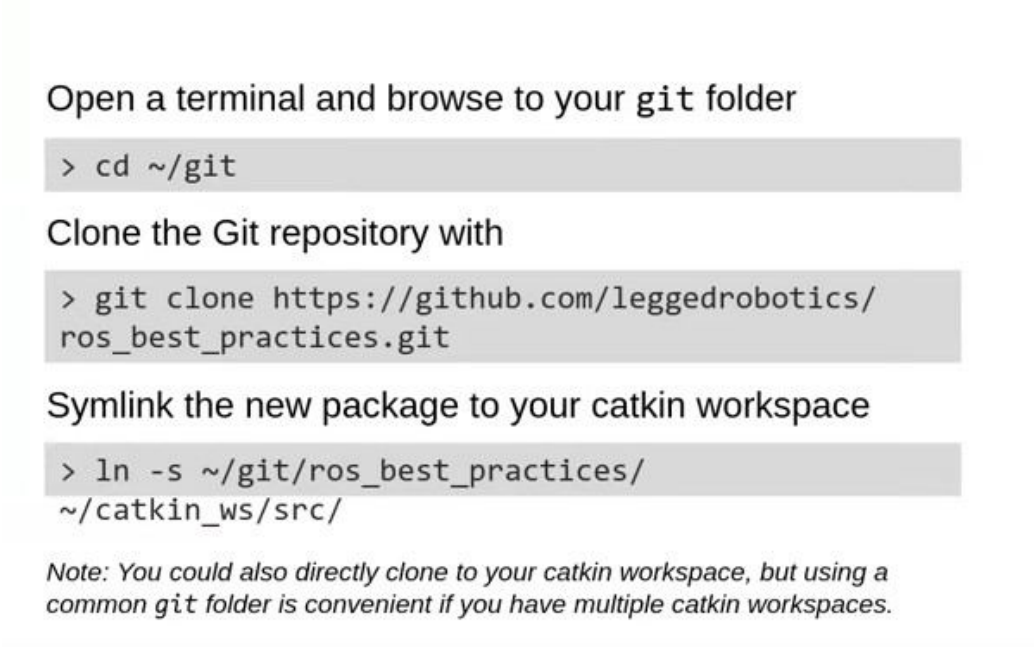
We can use rosparam get / to show the contents of all the params. We can also store it in a file. By using, rosparam dump [file_name] [namespace].

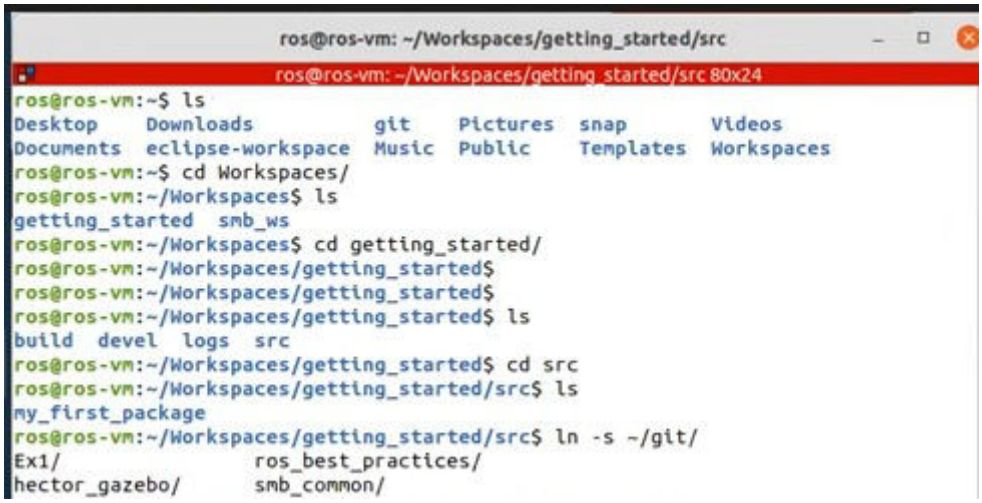
ROS Workspace Environment
Defines the context we are working on or with. Such as which packages should be used. (For example, when we have different versions of packages installed).



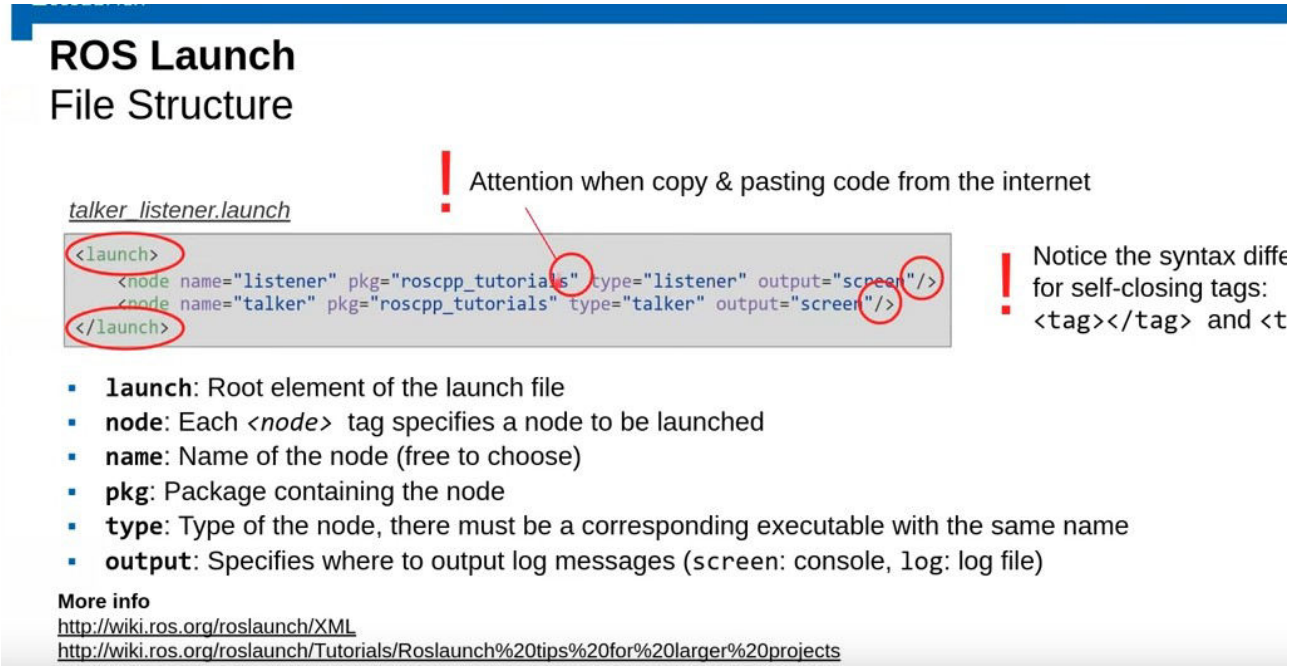
We work with the src directory under catkin workspace. The rest are recommended to not be tinkered with. If necessary, clean the build and devel space with

```
$ catkin clean
```



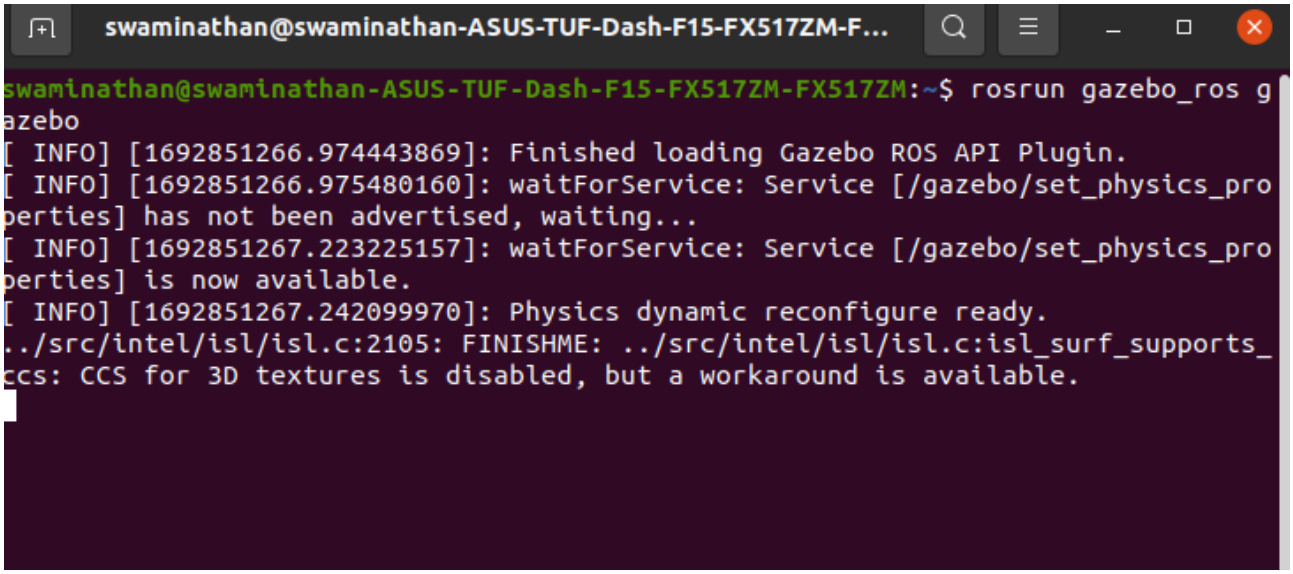


ROS Launch:
Tool for launching multiple nodes as well as setting parameters. Are written in XML as *.launch files. Launch automatically starts a roscore.



Create reusable launch files with <arg> tag, which works like a parameter. Use arguments in launch file with `$(arg arg_name)`. Assign arg values in the terminal with roslaunch `launch_file.launch arg_name:=value`.

Gazebo Simulator:



Tool to simulate 3-d rigid body dynamics