Create a ROS workspace

$ mkdir -p ~/catkin\_ws/src

$ cd ~/catkin\_ws/

$ catkin\_make //it will make cmakelist.txt in src folder

**Packages:** Packages are the software organization unit of ROS code. Each package can contain libraries, executables, scripts, or other artifacts.

**Manifests (**[**package.xml**](http://wiki.ros.org/catkin/package.xml)**):** A manifest is a description of a *package*

* [*rospack*](http://wiki.ros.org/rospack) *allows you to get information about packages*

*$ rospack find roscpp*

*o/p-->YOUR\_INSTALL\_PATH/share/roscpp*

* *$ roscd roscpp*

*Now let's print the working directory using the Unix command* [*pwd*](http://ss64.com/bash/pwd.html)*:*

*$ pwd*

*You should see: YOUR\_INSTALL\_PATH/share/roscpp*

* [*rosls*](http://wiki.ros.org/rosbash#rosls) *is part of the* [*rosbash*](http://wiki.ros.org/rosbash) *suite. It allows you to* [*ls*](http://ss64.com/bash/ls.html) *directly in a package by name rather than by absolute path.*

*$ rosls [locationname[/subdir]]*

*Example: $ rosls roscpp\_tutorials*

*would return: cmake launch package.xml srv*

## *Packages in a catkin Workspace*

*The recommended method of working with catkin packages is using a* [*catkin workspace*](http://wiki.ros.org/catkin/workspaces)*, 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*

*Creating a catkin package*

*$ cd ~/catkin\_ws/src*

*$ catkin\_create\_pkg beginner\_tutorials std\_msgs rospy roscpp //*this will create bt package and it depends on std\_msgs,rospy,roscpp

## 

## *Building a catkin workspace and sourcing the setup file*

*$ 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*

* *$ rospack depends beginner\_tutorials (all dependency direct+indirect)*
* *$ rospack depends1 beginner\_tutorials (direct dep.)*

*->rospy roscpp std\_msgs*

* *$ rospack depends1 rospy (indirect dep.)*

*Genpy …….*

### *Building Your Package*

*$ cd ~/catkin\_ws/*

*$ ls src*

* *beginner\_tutorials/ CMakeLists.txt@*

*$ catkin\_make*

*Quick Overview of Graph Concepts*

* [*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 is the first thing you should run when using ROS.*

## *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.*

***Note:*** *When opening a new terminal your environment is reset and your ~/.bashrc file is sourced. If you have trouble running commands like rosnode then you might need to add some environment setup files to your ~/.bashrc or manually re-source them.*

*rosnode displays information about the ROS nodes that are currently running. The rosnode list command lists these active nodes:*

*$ rosnode list*

* *You will see:*
* */rosout*

*The rosnode info command returns information about a specific node.*

*$ rosnode info /rosout*

## *Using rosrun (running a node)*

*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]*

*Then, in a* ***new terminal****:*

*$ rosrun turtlesim turtlesim\_node*

## 

## *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.*

# 

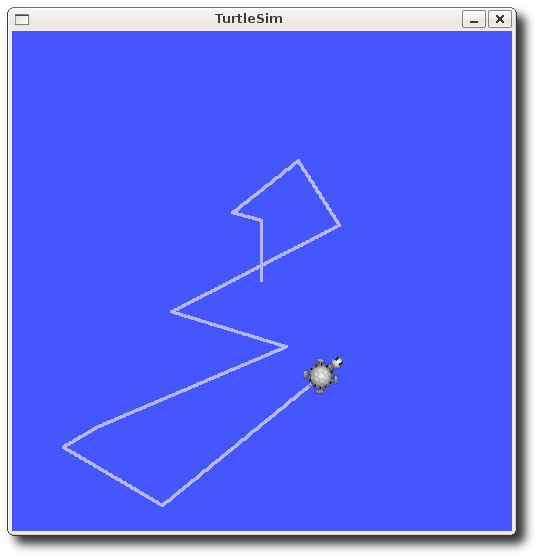
# 

# *Understanding ROS Topics*

*$ roscore*

*$ rosrun turtlesim turtlesim\_node*

*$ rosrun turtlesim turtle\_teleop\_key*

* *(Use arrow keys to move the turtle)*

## *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..*

### *Introducing rostopic*

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

*$ rostopic -h (You can use the help option to get the available sub-commands for rostopic)*

*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*

### *Using rostopic echo*

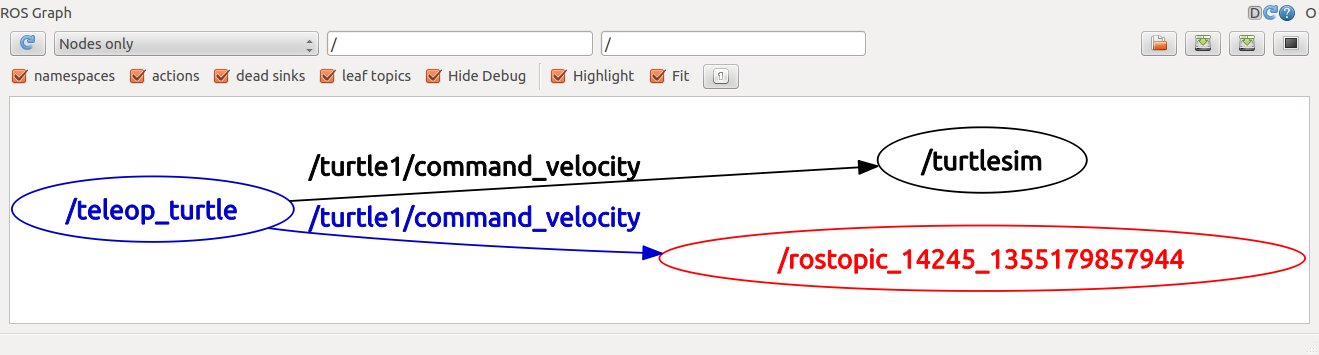
*rostopic echo shows the data published on a topic.*

*Usage:*

*$ rostopic echo [topic]*

*Ex: $ rostopic echo /turtle1/cmd\_vel*

*Now let's look at rqt\_graph again. As you can see rostopic echo, shown here in red, is now also* ***subscribed*** *to the turtle1/command\_velocity topic.*

**

### *publisher (turtle\_teleop\_key) and subscriber (turtlesim\_node) and topic /turtle1/cmd \_vel*

### *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*

*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*

*$ rostopic list -v*

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

*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*

# 

## *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.*

### *Using rostopic type*

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

## *Usage: rostopic type [topic]*

## *Try: $ 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*

## *Now that we know what type of message turtlesim expects, we can publish commands to our turtle..*

### *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, 0.0]' '[0.0, 0.0, 1.8]'*

## *The previous command will send a single message to turtlesim telling it to move with a linear velocity of 2.0, and an angular velocity of 1.8 .*

## *turtle(rostopicpub).png*

## *This is a pretty complicated example, so lets look at each argument in detail.*

## *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 arguments is option.*

## *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:*

## *For ROS Hydro and later,*

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

## *For ROS Groovy and earlier,*

## *$ rostopic pub /turtle1/command\_velocity turtlesim/Velocity -r 1 -- 2.0 -1.8*

## *This publishes the velocity commands at a rate of 1 Hz on the velocity topic.*

## *turtle(rostopicpub)2.png*

## *We can also look at what is happening in rqt\_graph. Press the refresh button in the upper-left. The rostopic pub node (here in red) is communicating with the rostopic echo node (here in green):*

## *rqt_graph_pub.png*

## *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*

## *`~`ROS Services*

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

## *Using rosservice*

*rosservice can easily attach to ROS's client/service framework with services. rosservice has many commands that can be used on services, as shown below:*

*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 list*

*$ rosservice list*

*The list command shows us that the turtlesim node provides nine services:. There are also two services related to the separate rosout node: /rosout/get\_loggers and /rosout/set\_logger\_level.*

*/clear /kill /reset /spawn /teleop\_turtle/get\_loggers /teleop\_turtle/set\_logger\_level /turtle1/set\_pen*

*/turtle1/teleport\_absolute /turtle1/teleport\_relative*

*/turtlesim/get\_loggers /turtlesim/set\_logger\_level*

*/rosout/get\_loggers*

*/rosout/set\_logger\_level*

*Let's look more closely at the clear service using rosservice type:*

### *rosservice type*

*Usage: rosservice type [service]*

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

*$ rosservice type /clear*

* *std\_srvs/Empty*
* *This service is empty, this means when the service call is made it takes no arguments (i.e. it sends no data when making a* ***request*** *and receives no data when receiving a* ***response****). Let's call this service using rosservice call:*

### *rosservice call*

*Usage: rosservice call [service] [args]*

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

*$ rosservice call /clear (it clears the background of the turtlesim\_node.)*

*Let's look at the case where the service has arguments by looking at the information for the service spawn:*

*$ rosservice type /spawn | rossrv show*

*float32 x*

*float32 y*

*float32 theta*

*string name*

*This service lets us spawn a new turtle at a given location and orientation. The name field is optional, so let's not give our new turtle a name and let turtlesim create one for us.*

*$ rosservice call /spawn 2 2 0.2 ""*

*The service call returns with the name of the newly created turtle name: turtle2*

*Using rosparam*

*rosparam allows you to store and manipulate data on the ROS* [*Parameter Server*](http://wiki.ros.org/Parameter%20Server)*. 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:*

*Usage:*

*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*

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

### *rosparam list*

*$ rosparam list*

*Here we can see that the turtlesim node has three parameters on the param server for background color:*

*/background\_b*

*/background\_g*

*/background\_r*

*/rosdistro*

*/roslaunch/uris/host\_57aea0986fef\_\_34309*

*/rosversion /run\_id*

### *rosparam set and rosparam get(Let's change one of the parameter values)*

*Syntax: rosparam get [param\_name]*

*Here will change the red channel of the background color:*

*$ rosparam set /background\_r 150*

1. [*Using rqt\_console and roslaunch*](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch)*This tutorial introduces ROS using* [*rqt\_console*](http://wiki.ros.org/rqt_console) *and* [*rqt\_logger\_level*](http://wiki.ros.org/rqt_logger_level) *for debugging and* [*roslaunch*](http://wiki.ros.org/roslaunch) *for starting many nodes at once. If you use ROS fuerte or ealier distros where* [*rqt*](http://wiki.ros.org/rqt) *isn't fully available, please see this page with* [*this page*](http://wiki.ros.org/ROS/Tutorials/UsingRxconsoleRoslaunch) *that uses old rx based tools.*

### *Using roslaunch*

*roslaunch starts nodes as defined in a launch file.*

*Usage:*

*$ roslaunch [package] [filename.launch]*

*$ roslaunch beginner\_tutorials turtlemimic.launch (tut 8)*

*Tut 9 Using rosed*

*rosed is part of the* [*rosbash*](http://wiki.ros.org/rosbash) *suite. It allows you to directly edit a file within a package by using the package name rather than having to type the entire path to the package.*

*Usage: $ rosed [package\_name] [filename]*

*Example: $ rosed roscpp Logger.msg*

*\Different nodes communicate by using rosmsgs on rostopics*

## *Introduction to msg and srv*

[*msg*](http://wiki.ros.org/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*](http://wiki.ros.org/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.*

*msgs are just simple text files with a field type and field name per line. The field types you can use are:*

* *int8, int16, int32, int64 (plus uint\*)*
* *float32, float64 ,string*
* *time, duration*

*srv files are just like msg files, except they contain two parts: a request and a response. The two parts are separated by a '---' line. Here is an example of a srv file:*

*int64 A*

*int64 B*

*---*

*int64 Sum*

## *Using rosmsg*

*That's all you need to do to create a msg. Let's make sure that ROS can see it using the rosmsg show command.*

*Usage: $ rosmsg show [message type] (Num -- The name of the msg Num)*

*Example: $ rosmsg show beginner\_tutorials/Num (beginner\_tutorials -- the package where You will see: int64 num the message is defined)*

*If you can't remember which Package a msg is in, you can leave out the package name. Try:*

*$ rosmsg show Num*

*You will see: [beginner\_tutorials/Num]:*

* *int64 num*

*ROSSRV*

*rossrv is a command-line tool for displaying information about ROS Service types.*

*Commands:*

*rossrv show Show service description*

*rossrv info Alias for rossrv show*

*rossrv list List all services*

*rossrv md5 Display service md5sum*

*rossrv package List services in a package*

### *Using rossrv*

*That's all you need to do to create a srv. Let's make sure that ROS can see it using the rossrv show command.*

*Usage: $ rossrv show <service type>*

*Example: $ rossrv show beginner\_tutorials/AddTwoInts*

*You will see: int64 a*

*int64 b*

*---*

*int64 sum*