ROS Guide for Walking Machine

Maxime St-Pierre

Mars 2017

Contents

Chapter 1

Introduction

Welcome to the Walking Machine ROS guide. This guide was build to help new recruits learn ROS. The bases of our robot is ROS which stands for "Robot Operating System" which is develop by the open-source robotics foundation. At the time of writing the current Long Term Release is Kinetic Kame.

Our Robot S.A.R.A. was build at École de Technologie Supérieur at Montréal Québec Canada by Walking Machine. The acronym S.A.R.A. means "Système d'Assitance Robotique Autonome".

Chapter 2

Installing ROS

The system requirements for ROS Kinect are available on the ROS wiki. For this tutorial, we will be focusing on installing ROS on ubuntu 16.04LTS.

2.1 Installing Ubuntu

To install Ubuntu you will need to get a ISO from the ubuntu website. The next step is to follow the setup step from ubuntu.

2.2 Installing ROS

The installation step is explain on the ROS wiki.

2.2.1 Manual Install (Recommended Method)

The recommended method is to install ros by hand the first time. This will help you learn the configuration needed for ROS. The process to install ROS is always the same.

1. Add ROS repo
2. Add ROS build key to the keyring
3. update the source list
4. install ros-desktop-full packages
5. Add ROS entry in the .bashrc
6. Initialize rosdep

Add ROS repo

See the following command :

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(
   lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list
   '
```

This command will add the repo for ROS inside the sources list of ubuntu.

Add ROS build key

See the following command:

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

This command will add will add the build server key to your keychain.

Update and Install ROS

See the following command:

```
$ sudo apt-get update && sudo apt-get install ros-kinetic-desktop-
   full
```

This command will update the source list and install ros-kinetic-desktop-full

Add ROS entry to .bashrc

See the following command:

```
$ echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc && source
   ~/.bashrc
```

This command will add the environment variable for ROS

Initialize rosdep

See the following command:

```
$ sudo rosdep init
$ rosdep update
```

This command will update rosdep with the latest information (DO NOT sudo rosdep update this will break your computer)

Add build dependencies

See the following command:

```
$ sudo apt-get install python-rosinstall python-rosinstall-
   generator python-wstool build-essential
```

This command will install the build-essential for building ROS packages

2.3 Configure ROS

In this section we are going to see how to configure a ROS workspace to put your work inside.

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

These command will create a new directory, initialize a workspace and make the whole workspace. This will also be the workspace for you to practice with ROS.

2.4 Conclusion

In this chapter, we saw how to install Ubuntu and also the step to install and configure ROS on your computer.

Chapter 3

Get Started with ROS

In this chapter, we are going to explore what are nodes and topic inside of ROS.

3.1 Introduction to ROS

The Robot Operating System or ROS use a distributed publisher and subscriber system to pass message between nodes. The nodes can also be on another computer or local. By using this architecture, we are able to scale on multiple computers over our network. This also helps use to debug the robot by hooking our laptop to the robot network.

Another important part of ROS is the tools. The tools like RQT, RVIZ and roswtf, help us debug whats going on with the robot in real-time. This is the avantage by using a pre-build ecosystem to build our robot with.

3.2 Introduction to Nodes

The nodes are the building blocks of any ROS based system. For our robot S.A.R.A. we use around 200 nodes when we are running fully autonomously.

3.3 Introduction to Packages

Each nodes must live inside of a directory that is call a packages. Each driver, state machine, and library.

3.4 Navigating the ROS Filesystem

3.4.1 Prerequisites

For this tutorial we will inspect a package in ros-tutorials, please install it using

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

3.4.2 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...

rospack

rospack allows you to get information about packages. In this tutorial, we are only going to cover the find option, which returns the path to package.

```
$ rospack [packages_name]
```

Example:

```
$ rospack roscpp
```

Would return

```
/opt/ros/kinetic/share/roscpp
```

roscd

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

```
$ roscd [packages_name]
```

Example:

```
roscd roscpp
```

After we check what the current directory is:

```
pwd
```

This command will return

```
/opt/ros/kinetic/share/roscpp
```

rosls

rosls is part of the rosbash suite. It allows you to ls 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
```

review

You may have noticed a pattern with the naming of the ROS tools:

- rospack = ros + packages
- roscd = ros + cd
- rosls = ros + ls

This naming pattern holds for many of the ROS tools.Now that you can get around in ROS, let's create a package.

3.5 Create a ros packages

In this section, we are going to learn how to create a package. For this tutorial, we are going to be using the catkin_ws we created before.

3.5.1 Basic of anatomy of a ROS 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.
- The package must contain a CMakeLists.txt which uses catkin.
- Each package must have its own folder

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

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

3.5.2 Creating a catkin package

To create the package you must be inside you catkin_ws

```
$ cd ~/catkin_ws/src
```

To create the package there is already a script inside of ros.

```
$ 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, which have been partially filled out with the information you gave catkin_create_pkg.

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]
```

3.5.3 Building a catkin workspace

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

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

This will build the whole workspace.

3.5.4 Customizing Your Package

This part of the tutorial will look at each file generated by catkin_create_pkg and describe, line by line, each component of those files and how you can customize them for your package.

Customizing the package.xml

The generated package.xml should be in your new package. Now lets go through the new package.xml and touch up any elements that need your attention.

First update the description tag:

```
<description>The beginner_tutorials package</description>
```

Change the description to anything you like, but by convention the first sentence should be short while covering the scope of the package. If it is hard to describe the package in a single sentence then it might need to be broken up.

Next comes the maintainer tag:

```
<!-- One maintainer tag required, multiple allowed, one person per
    tag -->
<!-- Example:  -->
<!-- <maintainer email="jane.doe@example.com">Jane Doe</maintainer
   > -->
<maintainer email="user@todo.todo">user</maintainer>
```

This is a required and important tag for the package.xml because it lets others know who to contact about the package. At least one maintainer is required, but you can have many if you like. The name of the maintainer goes into the body of the tag, but there is also an email attribute that should be filled out:

```
<maintainer email="you@yourdomain.tld">Your Name</maintainer>
```

Next is the license tag, which is also required:

```
<!-- 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>
```

You should choose a license and fill it in here. Some common open source licenses are BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, and LGPLv3.

The one that we are going to use the most is Apache2.0 (`https://www.apache.org/licenses/LICENSE-2.0`)

So for this package, we are going to use Apache2.0.

```
<license>Apache2.0</license>
```

The next set of tags describe the dependencies of your package. The dependencies are split into build_depend, buildtool_depend, run_depend, test_depend.

```
<!-- The *_depend tags are used to specify dependencies -->
<!-- Dependencies can be catkin packages or system dependencies -->
<!-- Examples: -->
<!-- Use build_depend for packages you need at compile time: -->
<!--   <build_depend>genmsg</build_depend> -->
<!-- Use buildtool_depend for build tool packages: -->
<!--   <buildtool_depend>catkin</buildtool_depend> -->
<!-- Use run_depend for packages you need at runtime: -->
<!--   <run_depend>python-yaml</run_depend> -->
<!-- Use test_depend for packages you need only for testing: -->
<!--   <test_depend>gtest</test_depend> -->
<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>
```

All of our listed dependencies have been added as a build_depend for us, in addition to the default buildtool_depend on catkin. In this case we want all of our specified dependencies to be available at build and run time, so we'll add a run_depend tag for each of them as well:

```
 <buildtool_depend>catkin</buildtool_depend>

 <build_depend>roscpp</build_depend>
 <build_depend>rospy</build_depend>
 <build_depend>std_msgs</build_depend>

 <run_depend>roscpp</run_depend>
 <run_depend>rospy</run_depend>
 <run_depend>std_msgs</run_depend>
```

As you can see the final package.xml, without comments and unused tags, is much more concise:

```
<?xml version="1.0"?>
<package>
<name>beginner_tutorials</name>
<version>0.1.0</version>
<description>The beginner_tutorials package</description>

<maintainer email="you@yourdomain.tld">Your Name</maintainer>
<license>BSD</license>
<url type="website">http://wiki.ros.org/beginner_tutorials</url>
<author email="you@yourdomain.tld">Jane Doe</author>

<buildtool_depend>catkin</buildtool_depend>

<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>

<run_depend>roscpp</run_depend>
<run_depend>rospy</run_depend>
<run_depend>std_msgs</run_depend>

</package>
```

3.5.5 Building your Package

Using catkin_make

catkin_make is a command line tool which adds some convenience to the standard catkin workflow. You can imagine that catkin_make combines the calls to cmake and make in the standard CMake workflow.

Usage:

```
# In a catkin workspace
$ catkin_make [make_targets] [-DCMAKE_VARIABLES=...]
```

catkin projects can be built together in workspaces. Building zero to many catkin packages in a workspace follows this work flow:

```
# In a catkin workspace
$ catkin_make
```

By doing this inside of your workspace, you now have a build packages.

3.6 Understanding ROS node

3.6.1 Prerequisites

For this tutorial, we will use a lightweight simulator, please install it using

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

3.6.2 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
- Topic: Nodes can publish messages to a topic as well as subscibe 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

3.6.3 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.

3.6.4 Client Libraries

Ros client libraries allow nodes written in different programming languages to communicate:

- rospy = python client library
- roscpp = c++ client library

3.6.5 roscore

roscore is the first thing you should run when using ROS.

Please run:

```
$ roscore
```

You will see somethiong similar to:

```
... logging to ~/.ros/log/9cf88ce4-b14d-11df-8a75-00251148e8cf/
   roslaunch-machine_name-13039.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://machine_name:33919/
ros_comm version 1.4.7

SUMMARY
======

PARAMETERS
* /rosversion
* /rosdistro

NODES

auto-starting new master
process[master]: started with pid [13054]
```

```
ROS_MASTER_URI=http://machine_name:11311/

setting /run_id to 9cf88ce4-b14d-11df-8a75-00251148e8cf
process[rosout-1]: started with pid [13067]
started core service [/rosout]
```

3.6.6 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
```

you will see:

```
/rosout
```

This showed us that here is only one node runnning: rosout. This is always running as it collects and logs nodes debugging output.

3.6.7 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
```

You will see the turtlesim window.

In a new terminal:

```
rosnode list
```

you will see something similar to:

```
/rosout
/turtlesim
```

3.7 Understanding ROS Topics

3.7.1 turtle keyboard teleoperation

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

```
$ rosrun turtlesim turtlesim_node
```

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.

3.7.2 ROS Topics

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

Using rqt_graph

rqt_graph creates a dynamic graph of what's going on in the system. rqt_graph is part or the rqt package. Unless you already have it installed, run:

```
$ sudo apt-get install ros-kinetic-rqt
$ sudo apt-get install ros-kinetic-rqt-common-plugins
```

In a new terminal:

```
$ rosrun rqt_graph rqt_graph
```

from this view you can see how the nodes communicate with each other.

3.8 Introducing rostopic

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

You can use the help option to get the available sub-commands 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
```

Using rostopic echo

rostopic echo shows the data published on a topic.

Usage:

```
rostopic echo [topic]
```

Let's look at the command velocity data published by the turtle_teleop_key node.

```
$ 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.

You should now see the following when you press the up key:

```
linear:
x: 2.0
y: 0.0
z: 0.0
angular:
x: 0.0
y: 0.0
z: 0.0
---
linear:
x: 2.0
y: 0.0
z: 0.0
angular:
x: 0.0
y: 0.0
z: 0.0
---
```

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-commands 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.

```
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
```

3.9 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.

3.9.1 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
```

You will get this definition:

```
geometry_msgs/Vector3 linear
float64 x
float64 y
float64 z
geometry_msgs/Vector3 angular
float64 x
float64 y
float64 z
```

3.10 rostopic continued

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

3.10.1 Using rostopic pub

rostopic pub publishes data on to a topic currently advertised.

Usage:

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

example:

```
$ 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 an linear velocity of 2.0, and an angular velocity of 1.8 .

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
```

the 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.

```
'[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
```

3.10.2 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
```

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
```

Now we can tell that the turtlesim is publishing data about our turtle at the rate of 60 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
```

3.10.3 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.

Pressing the minus button shows a menu that allows you to hide the specified topic from the plot. Hiding both the topics you just added and adding /turtle1/pose/theta will result in the plot shown in the next figure.

3.11 Understanding ROS Services and Parameters

3.11.1 ROS Services

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

3.11.2 Using rosservice

rosservice can easily attach to ROS's client/service framework with services. rosservice has many commands that can be used on topics, as show 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: reset, clear, spawn, kill, turtle1/set_pen, /turtle1/teleport_absolute, /turtle1/teleport_relative, turtlesim/get_loggers, and turtlesim/set_logger_level. There are also two services related to the separate rosout node: /rosout/get_loggers and /rosout/set_logger_level.

```
/clear
/kill
/reset
/rosout/get_loggers
/rosout/set_logger_level
/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
```

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

rosservice type

Usage:

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

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

```
$ rosservice call /clear
```

This does what we expect, 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
---
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
```

Now we should have two turtles in our turtlesim.

3.11.3 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:

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 the background color:

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

Let's change one of the parameter values using rosparam set:

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

86
```

We can also use rosparam get / to show us the contents of the entire Parameter Server.

```
$ rosparam get /

background_b: 255
background_g: 86
background_r: 150
roslaunch:
uris: {'aqy:51932': 'http://aqy:51932/'}
run_id: e07ea71e-98df-11de-8875-001b21201aa8
```

You may wish to store this in a file so that you can reload it at another time. This is easy using rosparam:

rosparam dump rosparam load

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 a new namespaces, e.g. copy:

```
$ rosparam load params.yaml copy
$ rosparam get /copy/background_b

255
```

Now that you understand how ROS services and params work, let's try using roslaunch.

3.12 roslaunch

roslaunch starts nodes as defined in a launch file.

Usage:

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

First go to the beginner_tutorials package we created and built earlier:

```
$ roscd beginner_tutorials
```

Then let's make a launch directory:

```
$ mkdir launch
$ cd launch
```

3.12.1 The launch file

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>
```

3.12.2 The Launch File Explained

Now, let's break the launch xml down.

```
<launch>
```

Here we start the launch file with the launch tag, so that the file is identified as a launch file.

```
  <group ns="turtlesim1">
<node pkg="turtlesim" name="sim" type="turtlesim_node"/>
</group>

<group ns="turtlesim2">
<node pkg="turtlesim" name="sim" type="turtlesim_node"/>
</group>
```

Here we start two groups with a namespace tag of turtlesim1 and turtlesim2 with a turtlesim node with a name of sim. This allows us to start two simulators without having name conflicts.

```
<node pkg="turtlesim" name="mimic" type="mimic">
        <remap from="input" to="turtlesim1/turtle1"/>
        <remap from="output" to="turtlesim2/turtle1"/>
</node>
```

Here we start the mimic node with the topics input and output renamed to turtlesim1 and turtlesim2. This renaming will cause turtlesim2 to mimic turtlesim1.

```
</launch>
```

This closes the xml tag for the launch file.

3.12.3 roslaunching

Now let's roslaunch the launch file:

```
$ 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]'
```

You will see the two turtlesims start moving even though the publish command is only being sent to turtlesim1.

3.13 conclusion

Chapter 4

Coding in ROS

4.1 Introduction to 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.

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
- other msg files
- variable-length array[] and fixed-length array[C]

There is also a special type in ROS: Header, the header contains a timestamp and coordinate frame information that are commonly used in ROS. You will frequently see the first line in a msg file have Header header.

Here is an example of a msg that uses a Header, a string primitive, and two other msgs :

```
Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

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
```

In the above example, A and B are the request, and Sum is the response.

4.1.1 Using msg

Creating a msg

Let's define a new msg in the package that was created in the previous tutorial.

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

The example .msg file above contains only 1 line. You can, of course, create a more complex file by adding multiple elements, one per line, like this:

```
string first_name
string last_name
uint8 age
uint32 score
```

There's one more step, though. We need to make sure that the msg files are turned into source code for C++, Python, and other languages:

Open package.xml, and make sure these two lines are in it and uncommented:

```
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
```

Note that at build time, we need "message_generation", while at runtime, we only need "message_runtime".

Open CMakeLists.txt

Add the message_generation dependency to the find_package call which already exists in your CMakeLists.txt so that you can generate messages. You can do this by simply adding message_generation to the list of COMPONENTS such that it looks like this:

```
# Do not just add this to your CMakeLists.txt, modify the existing
   text to add message_generation before the closing parenthesis
find_package(catkin REQUIRED COMPONENTS
roscpp
rospy
std_msgs
message_generation
)
```

You may notice that sometimes your project builds fine even if you did not call find_package with all dependencies. This is because catkin combines all your projects into one, so if an earlier project calls find_package, yours is configured with the same values. But forgetting the call means your project can easily break when built in isolation.

Also make sure you export the message runtime dependency.

```
catkin_package(
...
CATKIN_DEPENDS message_runtime ...
...)
```

Find the following block of code:

```
# add_message_files(
#   FILES
#   Message1.msg
#   Message2.msg
# )
```

Uncomment it by removing the # symbols and then replace the stand in Message*.msg files with your .msg file, such that it looks like this:

```
add_message_files(
FILES
Num.msg
)
```

By adding the .msg files manually, we make sure that CMake knows when it has to reconfigure the project after you add other .msg files.

Now we must ensure the generate_messages() function is called.

You need to uncomment these lines:

```
# generate_messages(
#   DEPENDENCIES
#   std_msgs
# )
```

so it looks like:

```
generate_messages(
DEPENDENCIES
std_msgs
)
```

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]
```

Example:

```
$ rosmsg show beginner_tutorials/Num
```

You will see:

```
int64 num
```

4.1.2 Using srv

Creating a srv

Let's use the package we just created to create a srv:

```
$ roscd beginner_tutorials
$ mkdir srv
```

Instead of creating a new srv definition by hand, we will copy an existing one from another package.

For that, roscp is a useful commandline tool for copying files from one package to another.

Usage:

```
$ roscp [package_name] [file_to_copy_path] [copy_path]
```

Now we can copy a service from the rospy_tutorials package:

```
$ roscp rospy_tutorials AddTwoInts.srv srv/AddTwoInts.srv
```

Also you need the same changes to package.xml for services as for messages, so look above for the additional dependencies required.

Remove # to uncomment the following lines:

```
#   FILES
#   Service1.srv
#   Service2.srv
# )
```

And replace the placeholder Service*.srv files for your service files:

```
add_service_files(
FILES
AddTwoInts.srv
)
```

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
```

4.1.3 Building the package

To make all the ros message and service:

```
# In your catkin workspace
$ roscd beginner_tutorials
$ cd ../..
$ catkin_make install
$ cd -
```

4.2 Writing a Simple Publisher and Subscriber (Python)

4.2.1 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 beginner_tutorials package, you created in the earlier tutorial:

```
$ roscd beginner_tutorials
```

The code

First lets create a 'scripts' folder to store our Python scripts in:

```
$ mkdir scripts
$ cd scripts
```

Create a new python script called talker.py and put this inside:

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def talker():
        pub = rospy.Publisher('chatter', String, queue_size=10)
        rospy.init_node('talker', anonymous=True)
        rate = rospy.Rate(10) # 10hz
        while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()

if __name__ == '__main__':
        try:
                talker()
        except rospy.ROSInterruptException:
                pass
```

The Code Explained

Now, let's break the code down.

```
#!/usr/bin/env python
```

Every Python ROS Node will have this declaration at the top. The first line makes sure your script is executed as a Python script.

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

You need to import rospy if you are writing a ROS Node. The std_msgs.msg import is so that we can reuse the std_msgs/String message type (a simple string container) for publishing.

```
pub = rospy.Publisher('chatter', String, queue_size=10)
rospy.init_node('talker', anonymous=True)
```

This section of code defines the talker's interface to the rest of ROS. pub = rospy.Publisher("chatter", String, queue_size=10) declares that your node is publishing to the chatter topic using the message type String. String here is actually the class std_msgs.msg.String. The queue_size argument limits the amount of queued messages if any subscriber is not receiving the them fast enough.

The next line, rospy.init_node(NAME, ...), is very important as it tells rospy the name of your node – until rospy has this information, it cannot start communicating with the ROS Master. In this case, your node will take on the name talker. NOTE: the name must be a base name, i.e. it cannot contain any slashes "/".

anonymous = True ensures that your node has a unique name by adding random numbers to the end of NAME.

```
rate = rospy.Rate(10) # 10hz
```

This line creates a Rate object rate. With the help of its method sleep(), it offers a convenient way for looping at the desired rate. With its argument of 10, we should expect to go through the loop 10 times per second (as long as our processing time does not exceed 1/10th of a second!)

```
    while not rospy.is_shutdown():
                hello_str = "hello world %s" % rospy.get_time()
                rospy.loginfo(hello_str)
                pub.publish(hello_str)
                rate.sleep()
```

This loop is a fairly standard rospy construct: checking the rospy.is_shutdown() flag and then doing work. You have to check is_shutdown() to check if your program should exit (e.g. if there is a Ctrl-C or otherwise). In this case, the "work" is a call to pub.publish(hello_str) that publishes a string to our chatter topic. The loop calls rate.sleep(), which sleeps just long enough to maintain the desired rate through the loop.

This loop also calls rospy.loginfo(str), which performs triple-duty: the messages get printed to screen, it gets written to the Node's log file, and it gets written to rosout.

std_msgs.msg.String is a very simple message type, so you may be wondering what it looks like to publish more complicated types. The general rule of thumb is that constructor args are in the same order as in the .msg file. You can also pass in no arguments and initialize the fields directly, e.g.

```
msg = String()
msg.data = str
```

or you can initialize some of the fields and leave the rest with default values:

```
String(data=str)
```

You may be wondering about the last little bit:

```
try:
        talker()
except rospy.ROSInterruptException:
        pass
```

In addition to the standard Python __main__ check, this catches a rospy.ROSInterruptException exception, which can be thrown by rospy.sleep() and rospy.Rate.sleep() methods when Ctrl-C is pressed or your Node is otherwise shutdown. The reason this exception is raised is so that you don't accidentally continue executing code after the sleep().

Now we need to write a node to receive the messages.

4.2.2 Wrting the Subscriber Node

The code

Create a new python script called listener.py and put this inside:

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def callback(data):
        rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.
           data)

def listener():

        # In ROS, nodes are uniquely named. If two nodes with the
           same
        # node are launched, the previous one is kicked off. The
        # anonymous=True flag means that rospy will choose a unique
        # name for our 'listener' node so that multiple listeners
           can
        # run simultaneously.
        rospy.init_node('listener', anonymous=True)
```

```
        rospy.Subscriber("chatter", String, callback)

        # spin() simply keeps python from exiting until this node
           is stopped
        rospy.spin()

if __name__ == '__main__':
        listener()
```

This declares that your node subscribes to the chatter topic which is of type std_msgs.msgs.String. When new messages are received, callback is invoked with the message as the first argument.

We also changed up the call to rospy.init_node() somewhat. We've added the anonymous=True keyword argument. ROS requires that each node have a unique name. If a node with the same name comes up, it bumps the previous one. This is so that malfunctioning nodes can easily be kicked off the network. The anonymous=True flag tells rospy to generate a unique name for the node so that you can have multiple listener.py nodes run easily.

The final addition, rospy.spin() simply keeps your node from exiting until the node has been shutdown. Unlike roscpp, rospy.spin() does not affect the subscriber callback functions, as those have their own threads.

Also never to forget to chmod +x the script this will tell linux that they are executable.

4.2.3 Building your nodes

We use CMake as our build system and, yes, you have to use it even for Python nodes. This is to make sure that the autogenerated Python code for messages and services is created.

Go to your catkin workspace and run catkin_make:

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

4.2.4 Running the Nodes

Running the Publisher

Make sure that a roscore is up and running

```
$ roscore
```

In the last tutorial, we made a publisher called "talker". Let's run it:

```
$ rosrun beginner_tutorials talker.py
```

You will see something similar to:

```
[INFO] [WallTime: 1314931831.774057] hello world 1314931831.77
[INFO] [WallTime: 1314931832.775497] hello world 1314931832.77
[INFO] [WallTime: 1314931833.778937] hello world 1314931833.78
[INFO] [WallTime: 1314931834.782059] hello world 1314931834.78
[INFO] [WallTime: 1314931835.784853] hello world 1314931835.78
[INFO] [WallTime: 1314931836.788106] hello world 1314931836.79
```

The publisher node is up and running. Now we need a subscriber to receive messages from the publisher.

Runnning the Subscriber

In the last tutorial, we made a subscriber called "listener". Let's run it:

```
$ rosrun beginner_tutorials listener.py
```

You will see something similar to:

```
[INFO] [WallTime: 1314931969.258941] /listener_17657_13149319687951
    heard hello world 1314931969.26
[INFO] [WallTime: 1314931970.262246] /listener_17657_13149319687951
    heard hello world 1314931970.26
[INFO] [WallTime: 1314931971.266348] /listener_17657_13149319687951
    heard hello world 1314931971.26
[INFO] [WallTime: 1314931972.270429] /listener_17657_13149319687951
    heard hello world 1314931972.27
[INFO] [WallTime: 1314931973.274382] /listener_17657_13149319687951
    heard hello world 1314931973.27
[INFO] [WallTime: 1314931974.277694] /listener_17657_13149319687951
    heard hello world 1314931974.28
[INFO] [WallTime: 1314931975.283708] /listener_17657_13149319687951
    heard hello world 1314931975.28
```

4.3 Writing a Simple Service and Client (Python)

Here we'll create the service ("add_two_ints_server") node which will receive two ints and return the sum.

Change directory into the beginner_tutorials package, you created in the earlier tutorial, creating a package:

```
$ roscd beginner_tutorials
```

4.3.1 The Code

Writing a Service Node

Create the scripts/add_two_ints_server.py file within the beginner_tutorials package and paste the following inside it:

```
#!/usr/bin/env python

from beginner_tutorials.srv import *
import rospy

def handle_add_two_ints(req):
        print "Returning [%s + %s = %s]"%(req.a, req.b, (req.a +
           req.b))
        return AddTwoIntsResponse(req.a + req.b)

def add_two_ints_server():
        rospy.init_node('add_two_ints_server')
        s = rospy.Service('add_two_ints', AddTwoInts,
           handle_add_two_ints)
        print "Ready to add two ints."
        rospy.spin()

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

Don't forget to make the node executable:

```
chmod +x scripts/add_two_ints_server.py
```

The Code Explained

Now, let's break the code down.

There's very little to writing a service using rospy. We declare our node using init_node() and then declare our service:

```
s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
```

This declares a new service named add_two_ints with the AddTwoInts service type. All requests are passed to handle_add_two_ints function. handle_add_two_ints is called with instances of AddTwoIntsRequest and returns instances of AddTwoIntsResponse.

Just like with the subscriber example, rospy.spin() keeps your code from exiting until the service is shutdown.

Writing the Client Node

Create the scripts/add_two_ints_client.py file within the beginner_tutorials package and paste the following inside it:

```
#!/usr/bin/env python

import sys
import rospy
from beginner_tutorials.srv import *

def add_two_ints_client(x, y):
        rospy.wait_for_service('add_two_ints')
        try:
        add_two_ints = rospy.ServiceProxy('add_two_ints',
           AddTwoInts)
        resp1 = add_two_ints(x, y)
        return resp1.sum
        except rospy.ServiceException, e:
        print "Service call failed: %s"%e

def usage():
        return "%s [x y]"%sys.argv[0]

if __name__ == "__main__":
        if len(sys.argv) == 3:
                x = int(sys.argv[1])
                y = int(sys.argv[2])
        else:
                print usage()
                sys.exit(1)
        print "Requesting %s+%s"%(x, y)
        print "%s + %s = %s"%(x, y, add_two_ints_client(x, y))
```

Don't forget to make the node executable:

```
chmod +x scripts/add_two_ints_client.py
```

The Code Explained

Now, let's break the code down.

The client code for calling services is also simple. For clients you don't have to call init_node().

4.3.2 Building your nodes

Go to your catkin workspace and run catkin_make.

```
# In your catkin workspace
$ cd ~/catkin_ws
$ catkin_make
```

4.4 Examining the Simple Service and Client

4.4.1 Running the Service

Let's start by running the service:

```
$ rosrun beginner_tutorials add_two_ints_server.py
```

You should see something similar to:

```
Ready to add two ints.
```

4.4.2 Running the client

Now let's run the client with the necessary arguments:

```
$ rosrun beginner_tutorials add_two_ints_client.py 1 3
```

You should see something similar to:

```
Requesting 1+3
1 + 3 = 4
```

Now that you've successfully run your first server and client, let's learn how to record and play back data.

Chapter 5

ROS Tools

5.1 Recording and playing back data

5.1.1 Recording data(creating a bag file)

This section of the tutorial will instruct you how to record topic data from a running ROS system. The topic data will be accumulated in a bag file.

First, execute the following commands in separate terminals:

Terminal 1:

```
roscore
```

Terminal 2:

```
rosrun turtlesim turtlesim_node
```

Terminal 3:

```
rosrun turtlesim turtle_teleop_key
```

This will start two nodes - the turtlesim visualizer and a node that allows for the keyboard control of turtlesim using the arrows keys on the keyboard. If you select the terminal window from which you launched turtle_keyboard, you should see something like the following:

```
Reading from keyboard
---------------------------
Use arrow keys to move the turtle.
```

Pressing the arrow keys on the keyboard should cause the turtle to move around the screen. Note that to move the turtle you must have the terminal from which you launched turtlesim selected and not the turtlesim window.

Recording all published topics

First lets examine the full list of topics that are currently being published in the running system. To do this, open a new terminal and execute the command:

```
rostopic list -v
```

This should yield the following output:

```
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
```

The list of published topics are the only message types that could potentially be recorded in the data log file, as only published messages are recorded. The topic /turtle1/cmd_vel is the command message published by teleop_turtle that is taken as input by the turtlesim process. The messages /turtle1/color_sensor and /turtle1/pose are output messages published by turtlesim.

We now will record the published data. Open a new terminal window. In this window run the following commands:

```
$ mkdir ~/bagfiles
$ cd ~/bagfiles
$ rosbag record -a
```

Here we are just making a temporary directory to record data and then running rosbag record with the option -a, indicating that all published topics should be accumulated in a bag file.

Move back to the terminal window with turtle_teleop and move the turtle around for 10 or so seconds.

In the window running rosbag record exit with a Ctrl-C. Now examine the contents of the directory /bagfiles. You should see a file with a name that begins with the year, date, and time and the suffix .bag. This is the bag file that contains all topics published by any node in the time that rosbag record was running.

5.1.2 Examining and playing the bag file

Now that we've recorded a bag file using rosbag record we can examine it and play it back using the commands rosbag info and rosbag play. First we are going to see what's recorded in the bag

file. We can do the info command – this command checks the contents of the bag file without playing it back. Execute the following command from the bagfiles directory:

```
$ rosbag info <your bagfile>
```

You should see something like:

```
path:        2014-12-10-20-08-34.bag
version:     2.0
duration:    1:38s (98s)
start:       Dec 10 2014 20:08:35.83 (1418270915.83)
end:         Dec 10 2014 20:10:14.38 (1418271014.38)
size:        865.0 KB
messages:    12471
compression: none [1/1 chunks]
types:       geometry_msgs/Twist [9f195f881246fdfa2798d1d3eebca84a]
rosgraph_msgs/Log   [acffd30cd6b6de30f120938c17c593fb]
turtlesim/Color     [353891e354491c51aabe32df673fb446]
turtlesim/Pose      [863b248d5016ca62ea2e895ae5265cf9]
topics:      /rosout                   4 msgs    : rosgraph_msgs/
   Log   (2 connections)
/turtle1/cmd_vel         169 msgs    : geometry_msgs/Twist
/turtle1/color_sensor   6149 msgs    : turtlesim/Color
/turtle1/pose           6149 msgs    : turtlesim/Pose
```

This tells us topic names and types as well as the number (count) of each message topic contained in the bag file. We can see that of the topics being advertised that we saw in the rostopic output, four of the five were actually published over our recording interval. As we ran rosbag record with the -a flag it recorded all messages published by all nodes.

The next step in this tutorial is to replay the bag file to reproduce behavior in the running system. First kill the teleop program that may be still running from the previous section - a Ctrl-C in the terminal where you started turtle_teleop_key. Leave turtlesim running. In a terminal window run the following command in the directory where you took the original bag file:

```
$ rosbag play <your bagfile>
```

In this window you should immediately see something like:

```
[ INFO] [1418271315.162885976]: Opening 2014-12-10-20-08-34.bag

Waiting 0.2 seconds after advertising topics... done.

Hit space to toggle paused, or 's' to step.
```

In its default mode rosbag play will wait for a certain period (.2 seconds) after advertising each message before it actually begins publishing the contents of the bag file. Waiting for some duration allows any subscriber of a message to be alerted that the message has been advertised and that

messages may follow. If rosbag play publishes messages immediately upon advertising, subscribers may not receive the first several published messages. The waiting period can be specified with the -d option.

Eventually the topic /turtle1/cmd_vel will be published and the turtle should start moving in turtlesim in a pattern similar to the one you executed from the teleop program. The duration between running rosbag play and the turtle moving should be approximately equal to the time between the original rosbag record execution and issuing the commands from the keyboard in the beginning part of the tutorial. You can have rosbag play not start at the beginning of the bag file but instead start some duration past the beginning using the -s argument. A final option that may be of interest is the -r option, which allows you to change the rate of publishing by a specified factor. If you execute:

```
$ rosbag play -r 2 <your bagfile>
```

You should see the turtle execute a slightly different trajectory - this is the trajectory that would have resulted had you issued your keyboard commands twice as fast.

5.1.3 Recording a subset of the data

When running a complicated system, such as the pr2 software suite, there may be hundreds of topics being published, with some topics, like camera image streams, potentially publishing huge amounts of data. In such a system it is often impractical to write log files consisting of all topics to disk in a single bag file. The rosbag record command supports logging only particular topics to a bag file, allowing a user to only record the topics of interest to them.

If any turtlesim nodes are running exit them and relaunch the keyboard teleop launch file:

```
$ rosrun turtlesim turtlesim_node
$ rosrun turtlesim turtle_teleop_key
```

In your bagfiles directory, run the following command:

```
$ rosbag record -O subset /turtle1/cmd_vel /turtle1/pose
```

The -O argument tells rosbag record to log to a file named subset.bag, and the topic arguments cause rosbag record to only subscribe to these two topics. Move the turtle around for several seconds using the keyboard arrow commands, and then Ctrl-C the rosbag record.

Now check the contents of the bag file (rosbag info subset.bag). You should see something like this, with only the indicated topics:

```
path:        subset.bag
version:     2.0
duration:    12.6s
start:       Dec 10 2014 20:20:49.45 (1418271649.45)
end:         Dec 10 2014 20:21:02.07 (1418271662.07)
size:        68.3 KB
```

```
messages:    813
compression: none [1/1 chunks]
types:       geometry_msgs/Twist [9f195f881246fdfa2798d1d3eebca84a]
turtlesim/Pose      [863b248d5016ca62ea2e895ae5265cf9]
topics:      /turtle1/cmd_vel    23 msgs    : geometry_msgs/Twist
/turtle1/pose      790 msgs    : turtlesim/Pose
```

5.1.4 The limitations of rosbag record/play

In the previous section you may have noted that the turtle's path may not have exactly mapped to the original keyboard input - the rough shape should have been the same, but the turtle may not have exactly tracked the same path. The reason for this is that the path tracked by turtlesim is very sensitive to small changes in timing in the system, and rosbag is limited in its ability to exactly duplicate the behavior of a running system in terms of when messages are recorded and processed by rosrecord, and when messages are produced and processed when using rosplay. For nodes like turtlesim, where minor timing changes in when command messages are processed can subtly alter behavior, the user should not expect perfectly mimicked behavior.

Now that you've learned how to record and play back data, let's learn how to troubleshoot with roswtf.

5.2 Getting started with roswtf

Before you start this part, make sure your roscore is NOT running.

5.2.1 Checking your installation

roswtf examines your system to try and find problems. Let's try it out:

```
$ roscd
$ roswtf
```

You should see (detail of the output varies):

```
Stack: ros
================================================================
Static checks summary:

No errors or warnings
================================================================

Cannot communicate with master, ignoring graph checks
```

If your installation ran correctly you should output similar to the above. The output is telling you:

- "Stack: ros": roswtf uses whatever your current directory is to determine what checks it does. This is telling us that you started roswtf in the ros stack.
- "Static checks summary": this is a report on any filesystem issues. It's telling us that there were no errors.
- "Cannot communicate with master, ignoring graph checks": the roscore isn't running, so roswtf didn't do any online checks.