

A Gentle Introduction to ROS

Chapter: Getting started

Jason M. O’Kane

Jason M. O’Kane
University of South Carolina
Department of Computer Science and Engineering
315 Main Street
Columbia, SC 29208

<http://www.cse.sc.edu/~jokane>

©2014, Jason Matthew O’Kane. All rights reserved.

This is version 2.1.6 (ab984b3), generated on April 24, 2018.

Typeset by the author using \LaTeX and `memoir.cls`.

ISBN 978-14-92143-23-9

Chapter 2

Getting started

In which we install ROS, introduce some basic ROS concepts, and interact with a working ROS system.

Before jumping into the details of how to write software that uses ROS, it will be valuable to get ROS up and running and to understand a few of the basic ideas that ROS uses. This chapter lays that groundwork. After a quick discussion of how to install ROS and configure your user account to use it, we'll have a look at a working ROS system (specifically, an instance of the turtlesim simulator) and learn how to interact with that system using some command line tools.

2.1 Installing ROS

Before you can do anything with ROS, naturally you must ensure that it is installed on your computer. (If you are working with a computer on which someone else has already installed ROS—including the `ros-indigo-turtlesim` package—you can skip directly to Section 2.2.) The installation process is well documented and mostly straightforward.¹² Here's a summary of the necessary steps.

Adding the ROS repository As root, create a file called

```
/etc/apt/sources.list.d/ros-latest.list
```

¹<http://wiki.ros.org/ROS/Installation>

²<http://wiki.ros.org/indigo/Installation/Ubuntu>

containing a single line:

```
deb http://packages.ros.org/ros/ubuntu trusty main
```



This line is specific to Ubuntu 14.04, whose codename is trusty. If you are using Ubuntu 13.10 instead, you can substitute saucy for trusty.

▶▶ *Other versions of Ubuntu—both older and newer—are not supported by the pre-compiled packages for the indigo distribution of ROS. However, for Ubuntu versions newer than 14.04, installing ROS from its source³ may be a reasonable option.*

If you are unsure of which Ubuntu version you're using, you can find out using this command:

```
lsb_release -a
```

The output should show both a codename and a release number.

Installing the package authentication key Before installing the ROS packages, you must acquire their package authentication key. First, download the key:

```
wget https://raw.githubusercontent.com/ros/rosdistro/master/ros.key
```

If this works correctly, you'll have a small binary file called `ros.key`. Next, you should configure the apt package management system to use this key:

```
sudo apt-key add ros.key
```

After completing this step (apt-key should say "OK"), you can safely delete `ros.key`.

Downloading the package lists Once the repositories are configured, you can get the latest lists of available packages in the usual way:

³<http://wiki.ros.org/indigo/Installation/Source>

```
sudo apt-get update
```

Note that this will update *all* of the repositories configured on your system, not just the newly added ROS repositories.

Installing the ROS packages Now we can actually install the ROS software. The simplest approach is to perform a complete install of the core ROS system:

```
sudo apt-get install ros-indigo-desktop-full
```

If you have plenty of free disk space—a few GB should suffice—this package is almost certainly the best choice. If you need them, there are also some more compact alternatives, including `ros-indigo-desktop` and `ros-indigo-ros-base`, that omit some packages and tools in favor of reduced disk space requirements.

Installing turtlesim In this book we'll refer many times to a simple “simulator” called `turtlesim` to illustrate how things work. If you plan to follow along with any of the examples—Recommended answer: Yes—you'll need to install `turtlesim`. Use a command like this:

```
sudo apt-get install ros-indigo-turtlesim
```

Setting up rosdep systemwide After installing the ROS packages, you'll need to execute this command:

```
sudo rosdep init
```

This is a one-time initialization step; once ROS is working correctly, many users will not need to revisit `rosdep`.

►► *As its name suggests, the purpose of this command is to initialize `rosdep`, which is a tool for checking and installing package dependencies in an OS-independent way.⁴ On Ubuntu, for example, `rosdep` acts as a front end to `apt-get`. We won't use `rosdep` directly, but we will use a few tools that invoke it behind the scenes. Those tools will be very unhappy if `rosdep` is not set up correctly.*

⁴<http://wiki.ros.org/rosdep>



The online documentation occasionally mentions a tool called `rosinstall`, whose job is to install ROS software from source.⁵ ⁶ The software that we'll need in this book is all available in Ubuntu `deb` packages, which do not require `rosinstall`.

2.2 Configuring your account

Whether you install ROS yourself or use a computer on which ROS is already installed, there are two important configuration steps that must be done within the account of every user that wants to use ROS.

Setting up `roscdep` in a user account First, you must initialize the `roscdep` system in your account, using this command:

```
roscdep update
```

This command stores some files in your home directory, in a subdirectory called `.ros`. It generally only needs to be done once.



Note that, unlike `roscdep init` above, the `roscdep update` command should be run using your normal user account, not using `sudo`.

Setting environment variables ROS relies on a few environment variables to locate the files it needs. To set these environment variables, you'll need to execute the `setup.bash` script that ROS provides, using this command:⁷

```
source /opt/ros/indigo/setup.bash
```

You can then confirm that the environment variables are set correctly using a command like this:

```
export | grep ROS
```

⁵<http://wiki.ros.org/rosinstall>

⁶<http://www.ros.org/doc/independent/api/rosinstall/html/>

⁷<http://wiki.ros.org/rosbash>

If everything has worked correctly, you should see a handful of values (showing values for environment variables like `ROS_DISTRO` and `ROS_PACKAGE_PATH`) as the output from this command. If `setup.bash` has not been run, then the output of this command will usually be empty.



If you get “command not found” errors from the ROS commands introduced later in this chapter, the most likely reason is that `setup.bash` has not been run in your current shell.

Note, however, that the steps listed above apply only to the current shell. It would work perfectly well to simply execute that `source` command each time you start a new shell in which you'd like to execute ROS commands. However, this is both annoying and remarkably easy to forget, especially when you consider that the modular design of many ROS systems often calls for several different commands to execute concurrently, each in a separate terminal.

Thus, you'll want to configure your account to run the `setup.bash` script automatically, each time you start a new shell. To do this, edit the file named `.bashrc` in your home directory, and put the above `source` command at the bottom.

►► *In addition to setting environment variables, this `setup.bash` script also defines bash functions to implement a few commands, including `roscd` and `rosls`, which are introduced below. These functions are defined in the `rosbash` package.⁸*

2.3 A minimal example using turtlesim

Before we begin to examine the details of how ROS works, let's start with an example. This quick exercise will serve a few different purposes: It will help you confirm that ROS is installed correctly, it will introduce the `turtlesim` simulator⁹ that is used in many online tutorials and throughout this book, and it will provide a working (albeit quite simple) system that we'll refer back to in the rest of this chapter.

⁸<http://wiki.ros.org/rosbash>

⁹<http://wiki.ros.org/turtlesim>

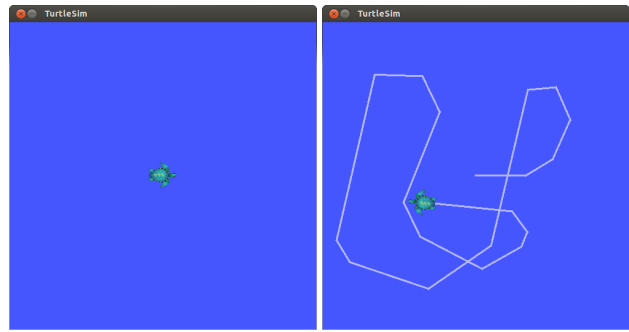


Figure 2.1: The turtlesim window, before and after some drawing.

Starting turtlesim In three separate terminals, execute these three commands:

```
roscore
roslaunch turtlesim turtlesim_node
roslaunch turtlesim turtle_teleop_key
```

The separate terminals are intended to allow all three commands to execute simultaneously. If everything works correctly, you should see a graphical window similar to the left part of Figure 2.1. This window shows a simulated, turtle-shaped robot that lives in a square world. (The appearance of your turtle may differ. The simulator selects from a collection of “mascot” turtles for each of the historical distributions of ROS.) If you give your third terminal (the one executing the `turtle_teleop_key` command) the input focus and press the Up, Down, Left, or Right keys, the turtle will move in response to your commands, leaving a trail behind it.



If the turtle does not move in response to your key presses, make sure that the `turtle_teleop_key` window has the input focus, for example by clicking inside it. You may need to arrange the windows carefully to focus this terminal while the simulation window is still visible.

Making virtual turtles draw lines is not, in itself, particularly exciting.¹ However, this example already has enough happening behind the scenes to illustrate many of the main ideas on which more interesting ROS systems are based.

You should keep these three terminals open, because the examples in the following sections will show some additional ways to interact with this system.

¹The author, for example, first started making turtles draw on computer screens sometime around 1987.

2.4 Packages

All ROS software is organized into **packages**. A ROS package is a coherent collection of files, generally including both executables and supporting files, that serves a specific purpose. In the example, we used two executables called `turtlesim_node` and `turtle_teleop_key`, both of which are members of the `turtlesim` package.



Be careful of the difference between ROS packages and the packages used by your operating system's package manager, such as the `deb` packages used by Ubuntu. The concepts are similar, and installing a `deb` package may add one or more ROS packages to your installation, but the two are not equivalent.

It is not an overstatement to say that *all* ROS software is part of one package or another. Importantly, this includes new programs that you create. We'll see how to create new packages in Section 3.1. In the meantime, ROS provides several commands for interacting with installed packages.

Listing and locating packages You can obtain a list of all of the installed ROS packages using this command:¹⁰¹¹

```
rospack list
```

On the author's system, this produces a list of 188 packages.

Each package is defined by a **manifest**, which is a file called `package.xml`. This file defines some details about the package, including its name, version, maintainer, and dependencies. The directory containing `package.xml` is called the **package directory**. (In fact, this is the *definition* of a ROS package: Any directory that ROS can find that contains a file named `package.xml` is a package directory.) This directory stores most of the package's files.

►► *An important exception is that, for most packages—specifically, those that have been updated to use the new catkin build system—compiled executables are not stored in the package directory, but in a separate standardized directory hierarchy.*

¹⁰<http://wiki.ros.org/ROS/Tutorials/NavigatingTheFilesystem>

¹¹<http://wiki.ros.org/rospack>

*For packages installed by apt-get, this hierarchy is rooted at /opt/ros/indigo. Executables are stored in the lib subdirectory under this root. Similarly, automatically generated include files are stored inside the include subdirectory under this root. When it needs them, ROS finds these files by searching in the directories listed in the CMAKE_PREFIX_PATH environment variable, which is set automatically by setup.bash. This sort of **out-of-source compilation** is one of the primary changes introduced by catkin in the groovy distribution of ROS, compared to fuerte and older distributions. Generally, though, all of this happens behind the scenes, and we can rely on ROS to find the files it needs.*

To find the directory of a single package, use the rospack find command:

```
rospack find package-name
```

Of course, there may be times when you don't know (or can't remember) the complete name of the package that you're interested in. In these cases, it's quite convenient that rospack supports **tab completion** for package names. For example, you could type

```
rospack find turtle
```

and, before pressing Enter, press the Tab key twice to see a list of all of the installed ROS packages whose names start with turtle.

In fact, most ROS commands support this kind of tab completion, not just for package names, but in nearly every place in which it makes sense. In the command above, you could also use Tabs to complete both the command name rospack and the "sub-command" find.



Frequent use of tab completion can go a long way toward reducing the number of things you'll need to remember, including the full names of packages, nodes, topics, message types, and services. Computers are quite good at storing and recalling these kinds of things. Unsolicited advice: Let your computer do that job for you.

Inspecting a package To view the files in a package directory, use a command like this:

```
rosls package-name
```

If you'd like to "go to" a package directory, you can change the current directory to a particular package, using a command like this:

```
1 $ rosls turtlesim
2 cmake
3 images
4 msg
5 package.xml
6 srv
7 $ rosls turtlesim/images
8 box-turtle.png
9 diamondback.png
10 electric.png
11 fuerte.png
12 groovy.png
13 hydro.png
14 hydro.svg
15 indigo.png
16 indigo.svg
17 palette.png
18 robot-turtle.png
19 sea-turtle.png
20 turtle.png
21 $ roscd turtlesim/images/
22 $ eog box-turtle.png
```

Listing 2.1: Using `rosls` and `roscd` to view the turtle images used by `turtlesim`. The `eog` command is the “Eye of GNOME” image viewer.

```
roscd package-name
```

As a simple example, suppose that you wanted to see the collection of turtle images used by `turtlesim`. Listing 2.1 shows an example of how you could use `rosls` and `roscd` to see a list of these images and to view one of them.



*In some parts of the online documentation, you may see references to the concept of a **stack**.¹² A stack is a collection of related packages. Starting with the groovy version of ROS, the stack concept was phased out and replaced by **meta-packages**.¹³ ¹⁴ The biggest difference is a “flattening” of the hierarchy: A meta-package is a package—It has a manifest just like any other package, and no other packages are stored inside its directory—whereas a stack is a container for packages*

stored as subdirectories. There's rarely a reason for new users to interact directly with stacks.

2.5 The master

So far we've talked primarily about *files*, and how they are organized into packages. Let's shift gears and talk now about how to actually *execute* some ROS software.

One of the basic goals of ROS is to enable roboticists to design software as a collection of small, mostly independent programs called **nodes** that all run at the same time. For this to work, those nodes must be able to communicate with one another. The part of ROS that facilitates this communication is called the **ROS master**. To start the master, use this command:

```
roscore
```

We've already seen this in action in the `turtlesim` example. For once, there is no additional complexity to worry about: `roscore` does not take any arguments, and does not need to be configured.

You should allow the master to continue running for the entire time that you're using ROS. One reasonable workflow is to start `roscore` in one terminal, then open other terminals for your "real" work. There are not many reasons to stop `roscore`, except when you've finished working with ROS. When you reach that point, you can stop the master by typing Ctrl-C in its terminal.

▶▶ *Though not many, there are a few reasons that restarting `roscore` might be a good idea. Examples: To switch to a new set of log files (See Chapter 4) or to clear the parameter server (See Chapter 7).*

¹²<http://wiki.ros.org/rosbuild/Stacks>

¹³http://wiki.ros.org/catkin/conceptual_overview

¹⁴<http://wiki.ros.org/catkin/package.xml>



Most ROS nodes connect to the master when they start up, and do not attempt to reconnect if that connection fails later on. Therefore, if you stop roscore, any other nodes running at the time will be unable to establish new connections, even if you restart roscore later.

The roscore command shown here is used to explicitly start the ROS master. In Chapter 6, we'll learn about a tool called roslaunch whose purpose is to start many nodes at once; this tool is smart enough to start a master if none is running, but will also happily use an existing master if there is one.

2.6 Nodes

Once you've started roscore, you can run programs that use ROS. A running instance of a ROS program is called a **node**.^{¶15}

▶▶ *The phrase “running instance of” in this definition is important. If we execute multiple copies of the same program at the same time—taking care to ensure that each uses a different node name—each of those copies is treated as a separate node. We will see this difference in action in Section 2.8.*

In the turtlesim example, we created two nodes. One node is an instance of an executable called turtlesim_node. This node is responsible for creating the turtlesim window and simulating the motion of the turtle. The second node is an instance of an executable called turtle_teleop_key. The abbreviation teleop is a shortened form of the word **teleoperation**, which refers to situations in which a human controls a robot remotely by giving direct movement commands. This node waits for an arrow key to be pressed, converts that key press to a movement command, and sends that command to the turtlesim_node node.

Starting nodes The basic command to create a node (also known as “running a ROS program”) is rosr`un`:^{¶16}

^{¶15}<http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes>

^{¶16}<http://wiki.ros.org/rosbash#rosrun>

```
roslaunch package-name executable-name
```

There are two required parameters to `roslaunch`. The first parameter is a package name. We discussed package names in Section 2.4. The second parameter is simply the name of an executable file within that package.

►► There’s nothing “magical” about `roslaunch`: It’s just a shell script that understands ROS’s file organization well enough to know where to look for executables by their package names. Once it finds the program you ask for, `roslaunch` executes that program normally. For example, if you really wanted to, you could execute `turtlesim__node` directly, just like any other program:

```
/opt/ros/indigo/lib/turtlesim/turtlesim__node
```

The work of registering with the master to become a ROS node happens inside the program, not in `roslaunch`.

Listing nodes ROS provides a few ways to get information about the nodes that are running at any particular time. To get a list of running nodes, try this command:¹⁷

```
rostopic list
```

If you do this after executing the commands from Section 2.3, you’ll see a list of three nodes:

```
/rosout  
/teleop__turtle  
/turtlesim
```

A few things about this list are worthy of note.

☞ The `/rosout` node is a special node that is started automatically by `roscore`. Its purpose is somewhat similar to the standard output (i.e. `std::cout`) that you might use in a console program. We look at `/rosout` more fully in Section 4.4.2.

¹⁷<http://wiki.ros.org/rostopic>

▶▶ The leading `/` in the name `/rosout` indicates that this node's name is in the **global namespace**. ROS has a rich system for naming nodes and other objects. This system, which Chapter 5 discusses in more detail, uses **namespaces** to organize things.¹⁸

☞ The other two nodes should be fairly clear: They're the simulator (`turtlesim`) and the teleoperation program (`teleop_turtle`) we started in Section 2.3.

☞ If you compare the output of `roslaunch list` to the executable names in the `roslaunch` commands from Section 2.3, you'll notice that **node names** are not necessarily the same as the names of the executables underlying those nodes.

▶▶ You can explicitly set the name of a node as part of the `roslaunch` command:

```
roslaunch package-name executable-name __name:=node-name
```

This approach will override the name that the node would normally have given itself, and can be important because ROS insists that every node have a distinct name. (We'll use `__name` in Section 2.8 to construct a slightly larger example system.) Generally, though, if you're assigning names using `__name` on a regular basis, you probably should be using a launch file—See Chapter 6—instead of running nodes individually.

Inspecting a node You can get some information about a particular node using this command:

```
roslaunch info node-name
```

The output includes a list of topics—See Section 2.7.2—for which that node is a publisher or subscriber, the services—See Chapter 8—offered by that node, its Linux process identifier (PID), and a summary of the connections it has made to other nodes.

Killing a node To kill a node you can use this command:

```
roslaunch kill node-name
```

¹⁸<http://wiki.ros.org/Names>

Unlike killing and restarting the master, killing and restarting a node usually does not have a major impact on other nodes; even for nodes that are exchanging messages, those connections would be dropped when the node is killed and reestablished when the node restarts.



You can also kill a node using the usual Ctrl-C technique. However, that method may not give the node a chance to unregister itself from the master. A symptom of this problem is that the killed node may still be listed by `rostopic list` for a while. This is harmless, but might make it more difficult to tell what's going on. To remove dead nodes from the list, you can use this command:

```
rostopic cleanup
```

2.7 Topics and messages

In our `turtlesim` example, it's clear that the teleoperation node and the simulator node must be talking to each other somehow. Otherwise, how would the turtle, which lives in the latter node, know when to move in response to your key presses, which are collected by the former node?

The primary mechanism that ROS nodes use to communicate is to send **messages**. Messages in ROS are organized into named **topics**.¹⁹ The idea is that a node that wants to share information will **publish** messages on the appropriate topic or topics; a node that wants to receive information will **subscribe** to the topic or topics that it's interested in. The ROS master takes care of ensuring that publishers and subscribers can find each other; the messages themselves are sent directly from publisher to subscriber.

2.7.1 Viewing the graph

This idea is probably easiest to see graphically, and the easiest way to visualize the publish-subscribe relationships between ROS nodes is to use this command:

```
rqt_graph
```

In this name, the `r` is for ROS, and the `qt` refers to the Qt GUI toolkit used to implement the program. You should see a GUI, most of which is devoted to showing the nodes in the

¹⁹<http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics>

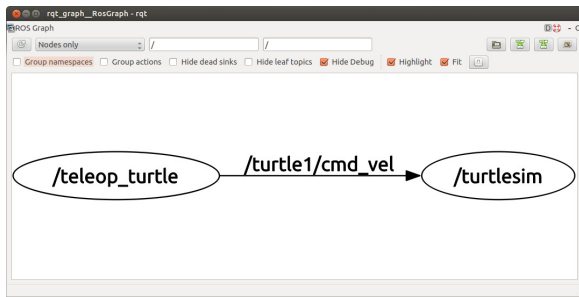


Figure 2.2: The `rqt_graph` interface, showing the graph for the `turtlesim` example. Debug nodes, including `rosout`, are omitted by default.

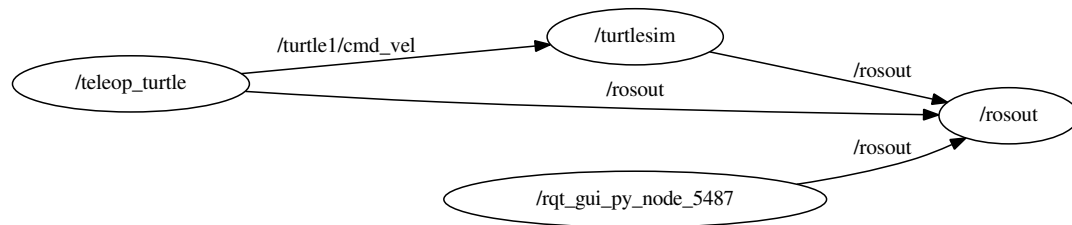



Figure 2.3: The complete `turtlesim` graph, including nodes that `rqt_graph` classifies as debug nodes.

current system. In this case, you will see something like Figure 2.2. In this graph, the ovals represent nodes, and the directed edges represent publisher-subscriber relationships. The graph tells us that the node named `/teleop_turtle` publishes messages on a topic called `/turtle1/cmd_vel`, and that the node named `/turtlesim` subscribes to those messages. (In this context, the name “`cmd_vel`” is short for “command velocity.”)

You might notice that the `rosout` node that we saw in Section 2.6 is missing from this view. By default, `rqt_graph` hides nodes that it thinks exist only for debugging. You can disable this feature by unchecking the “Hide debug” box. Figure 2.3 shows the resulting graph.

 Notice that `rqt_graph` itself appears as a node.

 All of these nodes publish messages on a topic called `/rosout`, to which the node named `/rosout` subscribes. This topic is one mechanism through which nodes can generate textual log messages. Chapter 4 has more about logging in ROS.



The name `/rosout` refers to both a node and a topic. ROS doesn't get confused by these kinds of duplicate names because it's always clear from the context whether we want to talk about the `/rosout` node or the `/rosout` topic.

This view of debug nodes is useful for seeing a true picture of the current state of things, but can also clutter the graph quite a bit with information that is not often very helpful.

The `rqt_graph` tool has several other options for tweaking the way that it shows the graph. The author's personal preference is to change the dropdown from "Nodes only" to "Nodes/Topics (all)", and to disable all of the checkboxes except "Hide Debug." This setup, whose results are shown in Figure 2.4, has the advantage that all of the topics are shown in rectangles, separate from the nodes. One can see, for example, that the `turtlesim` node, in addition to subscribing to velocity commands, also publishes both its current pose and data from a simulated color sensor. When you're exploring a new ROS system, `rqt_graph`, especially with these options, can be a useful way to discover what topics are available for your programs to use to communicate with the existing nodes.



The phenomenon of having topics with no subscribers may seem like a bug, but it's actually very common. The intuition is that ROS nodes are usually designed to publish the useful information that they have, without worrying about whether anyone is subscribing to those messages. This helps to reduce the level of coupling between individual nodes.

Now we can understand at least part of how the `turtlesim` teleoperation system works. When you press a key, the `/teleop_turtle` node publishes messages with those movement commands on a topic called `/turtle1/cmd_vel`. Because it subscribes to that topic, the `turtlesim_node` receives those messages, and simulates the turtle moving with the requested velocity. The important points here are:

- ☞ The simulator doesn't care (or even know) which program publishes those `cmd_vel` messages. Any program that publishes on that topic can control the turtle.
- ☞ The teleoperation program doesn't care (or even know) which program subscribes to the `cmd_vel` messages it publishes. Any program that subscribes to that topic is free to respond to those commands.

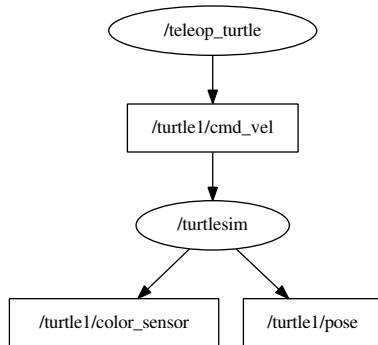


Figure 2.4: The turtlesim graph, showing all topics, including those with no publishers or no subscribers, as distinct objects.

By the way, these topic names begin with `/turtle1` because they are concerned with the default turtle, whose name happens to be “turtle1.” We’ll see, in Chapter 8, how to add additional turtles to a turtlesim window.

2.7.2 Messages and message types

So far we’ve talked about the idea that nodes can send messages to each other, but we’ve been quite vague about what information is actually contained in those messages. Let’s take a closer look at the topics and messages themselves.

Listing topics To get a list of active topics, use this command:²⁰

```
rostopic list
```

In our example, this shows a list of five topics:

```
/rosout
/rosout_agg
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

The topic list should, of course, be the same as the set of topics viewable in `rqt_graph`, but might be more convenient to see in text form.

²⁰<http://wiki.ros.org/rostopic>

Echoing messages You can see the actual messages that are being published on a single topic using the `rostopic` command:

```
rostopic echo topic-name
```

This command will dump any messages published on the given topic to the terminal. Listing 2.2 shows some example output from

```
rostopic echo /turtle1/cmd_vel
```

taken at a time when `/teleop_turtle` was receiving keystrokes. Each `---` line in the output shows the end of one message and the start of another. In this case, there were three messages.

Measuring publication rates There are also two commands for measuring the speed at which messages are published and the bandwidth consumed by those messages:

```
rostopic hz topic-name
rostopic bw topic-name
```

These commands subscribe to the given topic and output statistics in units of messages per second and bytes per second, respectively.



Even if you don't care much about the specific rates, these commands can be useful for debugging, because they provide an easy way to verify that messages are indeed being published regularly on particular topics.

Inspecting a topic You can learn more about a topic using the `rostopic info` command:

```
rostopic info topic-name
```

For example, from this command:

```
rostopic info /turtle1/color_sensor
```

you should see output similar to this:

```
Type: turtlesim/Color
```

```
Publishers:
```

```
* /turtlesim (http://donatello:46397/)
```

```
Subscribers: None
```

```
1 linear :
2   x: 2.0
3   y: 0.0
4   z: 0.0
5 angular :
6   x: 0.0
7   y: 0.0
8   z: 0.0
9 ---
10 linear :
11  x: 0.0
12  y: 0.0
13  z: 0.0
14 angular :
15  x: 0.0
16  y: 0.0
17  z: -2.0
18 ---
19 linear :
20  x: 2.0
21  y: 0.0
22  z: 0.0
23 angular :
24  x: 0.0
25  y: 0.0
26  z: 0.0
27 ---
```

Listing 2.2: Sample output from rostopic echo.

The most important part of this output is the very first line, which shows the **message type** of that topic. In the case of `/turtle1/color_sensor`, the message type is `turtlesim/Color`. The word “type” in this context is referring to the concept of a **data type**. It’s important to understand message types because they determine the *content* of the messages. That is, the message type of a topic tells you what information is included in each message on that topic, and how that information is organized.

Inspecting a message type To see details about a message type, use a command like this:^{¶21}^{¶22}

```
rosmmsg show message-type-name
```

Let's try using it on the message type for `/turtle1/color_sensor` that we found above:

```
rosmmsg show turtlesim/Color
```

The output is:

```
uint8 r
uint8 g
uint8 b
```

The format is a list of **fields**, one per line. Each field is defined by a built-in data type (like `int8`, `bool`, or `string`) and a field name. The output above tells us that a `turtlesim/Color` is a thing that contains three unsigned 8-bit integers called `r`, `g`, and `b`. Every message on any topic with message type `turtlesim/Color` is defined by values for these three fields. (As you might guess, these numbers correspond to the red-green-blue color intensities for the pixel under the center of the simulated turtle.)

Another example, one we'll revisit several times, is `geometry_msgs/Twist`. This is the message type for the `/turtle1/cmd_vel` topic, and it is slightly more complicated:

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

In this case, both `linear` and `angular` are **composite fields** whose data type is `geometry_msgs/Vector3`. The indentation shows that fields named `x`, `y`, and `z` are members within those two top-level fields. That is, a message with type `geometry_msgs/Twist` contains exactly six numbers, organized into two vectors called `linear` and `angular`. Each of these numbers has the built-in type `float64`, which means, naturally, that each is a 64-bit floating point number.

^{¶21}<http://wiki.ros.org/rosmmsg>

^{¶22}<http://wiki.ros.org/msg>

In general, a composite field is simply a combination of one or more sub-fields, each of which may be another composite field or a simple field with a built-in data type. The same idea appears in C++ and other object-oriented languages, in which one object may have other objects as data members.

►► It's worth noting that the data types of composite fields are message types in their own right. For example, it would be perfectly legitimate to have topic with message type `geometry_msgs/Vector3`. Messages on with this type would consist of three top-level fields, namely `x`, `y`, and `z`.

This kind of nesting can be useful to preventing code duplication for systems in which many message types share common elements. A common example is the message type `std_msgs/Header`, which contains some basic sequence, timestamp, and coordinate frame information. This type is included as a composite field called header in hundreds of other message types.

Fortunately, `rosmmsg` show automatically expands composite fields all the way down to the underlying built-in types, using indentation to show the nested structure, so there is often no need to inspect the nested message types directly.

Message types can also contain arrays with fixed or variable length (shown with square brackets `[]`) and constants (generally for interpreting the contents of other, non-constant fields). These features are not used by `turtlesim`. For an example message type that uses these features, have a look at `sensor_msgs/NavSatFix`, which represents a single GPS fix.

Publishing messages from the command line Most of the time, the work of publishing messages is done by specialized programs.² However, you may find it useful at times to publish messages by hand. To do this, use `rostopic`:²³

```
rostopic pub -r rate-in-hz topic-name message-type message-content
```

This command repeatedly publishes the given message on the given topic at the given rate.

The final *message content* parameter should provide values for all of the fields in the message type, in order. Here's an example:

```
rostopic pub -r 1 /turtle1/cmd_vel geometry_msgs/Twist '[2, 0, 0]' '[0, 0, 0]'
```

²Indeed, creating those programs is the primary subject matter of this book!

²³<http://wiki.ros.org/rostopic>

The values are assigned to message fields in the same order that they are shown by `rosmmsg show`. In the case, the first three numbers denote the desired linear velocity and the final three numbers denote the desired angular velocity. We use single quotes (`'...'`) and square brackets (`[...]`) to group the individual subfields into the two top-level composite fields. As you might guess, the messages generated by this example command the turtle to drive straight ahead (along its x -axis), with no rotation.

Likewise, a command like this will command the robot to rotate in place about its z -axis (which is perpendicular to your computer's screen):

```
rostopic pub -r 1 /turtle1/cmd_vel geometry_msgs/Twist '[0, 0, 0]' '[0, 0, 1]'
```



In fact, the two non-zero fields from the last two examples—specifically, `linear.x` and `angular.z`—are the only fields within `geometry_msgs/Twist` that `turtlesim` pays any attention to. Because the other four fields represent motions that the two-dimensional simulator does not allow, `turtlesim` ignores them.

►► The syntax shown above has the distinct disadvantage that you must remember all of the fields of the message type and the order in which they appear. An alternative is to give single parameter specifying all of the fields as a single YAML (a recursive acronym for “YAML Ain't Markup Language”²⁴) dictionary. This command (which does, in fact, contain newline characters) is equivalent to the one above, but it explicitly shows the mapping from field names to values:

```
rostopic pub /turtle1/cmd_vel geometry_msgs/Twist "linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0"
```

There are enough tricky interactions between `bash` and `YAML` that the online documentation has an entire page devoted just to the use of `YAML` on the command line.²⁵ ²⁶ The simplest way to get the syntax correct is to use `tab completion`.

Pressing Tab after entering the message type will insert a fully formed YAML dictionary, with all of the fields in the given message type. The tab-generated message will use default values (zero, false, empty string, etc), but you can edit it to contain the real message content that you want.

There are a few additional options to `rostopic pub` that might be of use.

- ☞ The form shown here uses `-r` to select the “rate mode” of `rostopic pub`, which publishes messages at regular intervals. This command also supports a one-time mode (`-1` “dash one”) and a special “latched” mode (`-l` “dash ell”) that publishes only once but ensures that new subscribers to that topic will receive the message. Latched mode is actually the default.
- ☞ It is also possible to read messages from a file (using `-f`) or from standard input (by omitting both `-f` and the message content from the command). In both cases, the input should be formatted like the output of `rostopic echo`.



Perhaps you have begun to imagine possibilities for using a scripted combination of `rostopic echo` and `rostopic pub` as a way of “recording” and “playing back” messages, for automating testing of your programs. If so, you’ll be interested in the `rosbag` tool (Chapter 9), which is a more complete implementation of this kind of idea.

Understanding message type names Like everything else in ROS, every message type belongs to a specific package. Message type names always contain a slash, and the part before the slash is the name of the containing package:

package-name/type-name

For example, the `turtlesim/Color` message type breaks down this way:

<u>turtlesim</u>	+	<u>Color</u>	⇒	<u>turtlesim/Color</u>
package name		type name		message data type

²⁴<http://www.yaml.org/>

²⁵<http://wiki.ros.org/YAMLOverview>

²⁶<http://wiki.ros.org/ROS/YAMLCommandLine>

This division of message type names serves a few purposes.

- ☞ Most directly, including packages in the message type names helps to prevent **name collisions**. For example, `geometry_msgs/Pose` and `turtlesim/Pose` are distinct message types that contain different (but conceptually similar) data.
- ☞ As we'll see in Chapter 3, when writing ROS programs, we'll need to declare dependencies on other packages that contain message types that we use. Including the package name as part of the message type name makes these dependencies easier to see.
- ☞ Finally, knowing the package that contains a given message type can be useful for figuring out that type's purpose. For example, the type name `ModelState` is quite mysterious in isolation, but the full name `gazebo/ModelState` clarifies that this message type is part of the Gazebo simulator, and likely contains information about one of the models within that simulation.

2.8 A larger example

So far in this chapter, we've seen how to start the ROS master, how to start ROS nodes, and how to investigate the topics those nodes use to communicate with one another. This section wraps up our introduction with an example a little larger than the one from Section 2.3, intended to illustrate a bit more fully the way topics and messages work.

First, stop any nodes that might be currently running. Start `roscore` if it's not already active. Then, in four separate terminals, run these four commands:

```
roslaunch turtlesim turtlesim_node __name:=A
roslaunch turtlesim turtlesim_node __name:=B
roslaunch turtlesim turtle_teleop_key __name:=C
roslaunch turtlesim turtle_teleop_key __name:=D
```

This should start two instances of the `turtlesim` simulator—These should appear in two separate windows—and two instances of the `turtlesim` teleoperation node.

The only element in the example that might be unfamiliar is the `__name` parameter to `roslaunch`. These parameters override the default name that each node tries to assign to itself. They're needed because the ROS master does not allow multiple nodes with the same name.

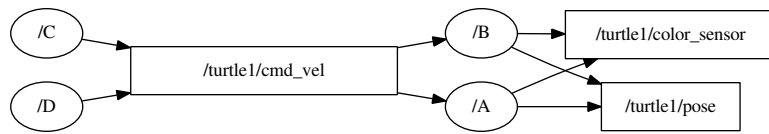


Figure 2.5: A slightly more complex ROS graph, with two turtlesim nodes named A and B and two teleoperation nodes named C and D.

►► If you do attempt to start two nodes with the same name, the new node will start without any problem, but the original node will terminate with a message like this:

```
[ WARN] [1369835799.391679597]: Shutdown request received.
[ WARN] [1369835799.391880002]: Reason given for shutdown:
[new node registered with same name]
```

Even though we're working to avoid it here, this behavior can be useful in general. This is especially true if you are debugging and revising a node, because it ensures that you won't have multiple versions of the same node running by mistake.

Before we discuss this four-node example, you may wish to take a moment to think about how the system will behave. What would the graph, as displayed by `rqt_graph`, look like? Which turtles would move in response to which teleoperation nodes?

Hopefully, you predicted that the graph would look like Figure 2.5, and that *both* turtles would make the same movements in response to key presses sent to *either* teleoperation node. Let's see why.

2.8.1 Communication via topics is many-to-many.

You might have expected each teleoperation node to connect to one simulator, creating two independently controllable simulations.³ Note, however, that these two kinds of nodes publish and subscribe, respectively, on the `/turtle1/cmd_vel` topic. Messages published on this topic, regardless of which node publishes them, are delivered to every subscriber of that topic.

In this example, every message published by teleoperation node C is delivered to both simulation nodes, namely A and B. Likewise, messages published by D are delivered to

³In Chapter 6, we'll see the right way to create these sorts of parallel, independent turtlesim simulations.

both A and B. When these messages arrive, the turtles move accordingly, regardless of which node published them. The main idea here is that topics and messages are used for **many-to-many** communication. Many publishers and many subscribers can share a single topic.

2.8.2 Nodes are loosely coupled.

No doubt you have noticed that we did not need to reprogram the turtlesim simulator to accept movement commands from multiple sources, nor did the teleoperation node need to be designed to drive multiple instances of the simulator at once. In fact, it would be an easy exercise to extend this example to arbitrarily many⁴ nodes of either type.

At the other extreme, consider what would happen if the turtlesim simulator were started in isolation, without any other nodes. In that situation, the simulator would wait for messages on `/turtle1/cmd_vel`, happily oblivious to the fact that there are no publishers for that topic.

The punchline is that our turtlesim nodes specifically—and most well-designed ROS nodes generally—are **loosely coupled**. None of the nodes explicitly know about any of the others; their only interactions are indirect, at the level of topics and messages. This independence of nodes, along with the decomposition it facilitates of larger tasks into smaller reusable pieces, is one of the key design features of ROS.

☞ Software (like `turtle_teleop_key`) that produces information can publish that information, without worrying about how that information is consumed.

☞ Software (like `turtlesim_node`) that consumes information can subscribe to the topic or topics containing the data it needs, without worrying about how those data are produced.

ROS does provide a mechanism, called **services**, for slightly more direct, one-to-one communication. This secondary technique is much less common, but does have its uses. Chapter 8 describes how to create and use services.

⁴...within reason. The author's computer, for example, begins to complain about having too many active X clients after starting about 100 simultaneous instances of `turtlesim_node`.

2.9 Checking for problems

One final (for now) command line tool, which can be helpful when ROS is not behaving the way you expect, is `roswtf`,⁵^{¶27} ^{¶28} which can be run with no arguments:

```
roswtf
```

This command performs a broad variety of sanity checks, including examinations of your environment variables, installed files, and running nodes. For example, `roswtf` checks whether the `rosdep` portions of the install process have been completed, whether any nodes appear to have hung or died unexpectedly, and whether the active nodes are correctly connected to each other. A complete list of checks performed by `roswtf` seems to exist, unfortunately, only in the Python source code for that tool.

2.10 Looking forward

The goal of this chapter was to introduce some basic ROS objects like nodes, messages, and topics, along with some command line tools for interacting with those objects. In the next chapter, we'll move beyond interacting with existing ROS programs and on to the process of writing new programs.

⁵The name of this tool is not explained in its documentation, but the author is pretty sure that it's an acronym for "Why The Failure?"

^{¶27}<http://wiki.ros.org/roswtf>

^{¶28}<http://wiki.ros.org/ROS/Tutorials/Gettingstartedwithroswtf>

