CHAPTER 2
Preliminaries

Before we look at how to write code in ROS, we're going to take a moment to disucss some of the key concepts that underly the framework. ROS systems are organized as a *computation graph*, where a number of independent programs each perform some piece of computation, passing data and results to other programs, arranged in a pre-defined network. In this chapter we'll discuss this graph architecture, and look at the command-line tools that you're going to be using to interact with it. We'll also discuss the details of the naming schemes and namespaces used by ROS, and how these can be tailored to promote reuse of your code.

The ROS Graph

As mentioned in the previous chapter, one of the original "challenge problems" which motivated the design of ROS was fondly referred to as the "fetch a stapler" problem. Imagine a relatively large and complex robot with several cameras and laser scanners, a manipulator arm, and a wheeled base. In the "fetch a stapler" problem, the robot's task is to navigate a typical home or office environment, find a stapler, and deliver it to a person who needs one. Exactly why a person would be working in a building equipped with large, complex robots but lacking a sufficient stockpile of staplers is the logical first question, but will be ignored for the purposes of this discussion. There are several key insights which can be gained from the "fetch a stapler" problem:

- the task can be decomposed into many independent subsystems, such as navigation, computer vision, grasping, and so on;
- it should be possible to re-purpose these subsystems for other tasks, such as doing security patrols, cleaning, delivering mail, and so on; and
- with proper hardware and geometry abstraction layers, the vast majority of the software should be able to run on *any* robot.

These principles can be illustrated by the fundamental rendering of a ROS system: its *graph*. A ROS system is made up of many different programs, running simultaneously, which communicate with each other by passing *messages*. Visualizing this as a *graph*, the programs are the *nodes*, and programs that communicate with each other are connected by edges, as shown in Figure 2-1. We can visualize any ROS system, large or small, in this way. In fact, this visualization is so useful that we actually call all ROS programs *nodes*, to help remember that each program is just one piece of a much larger system.

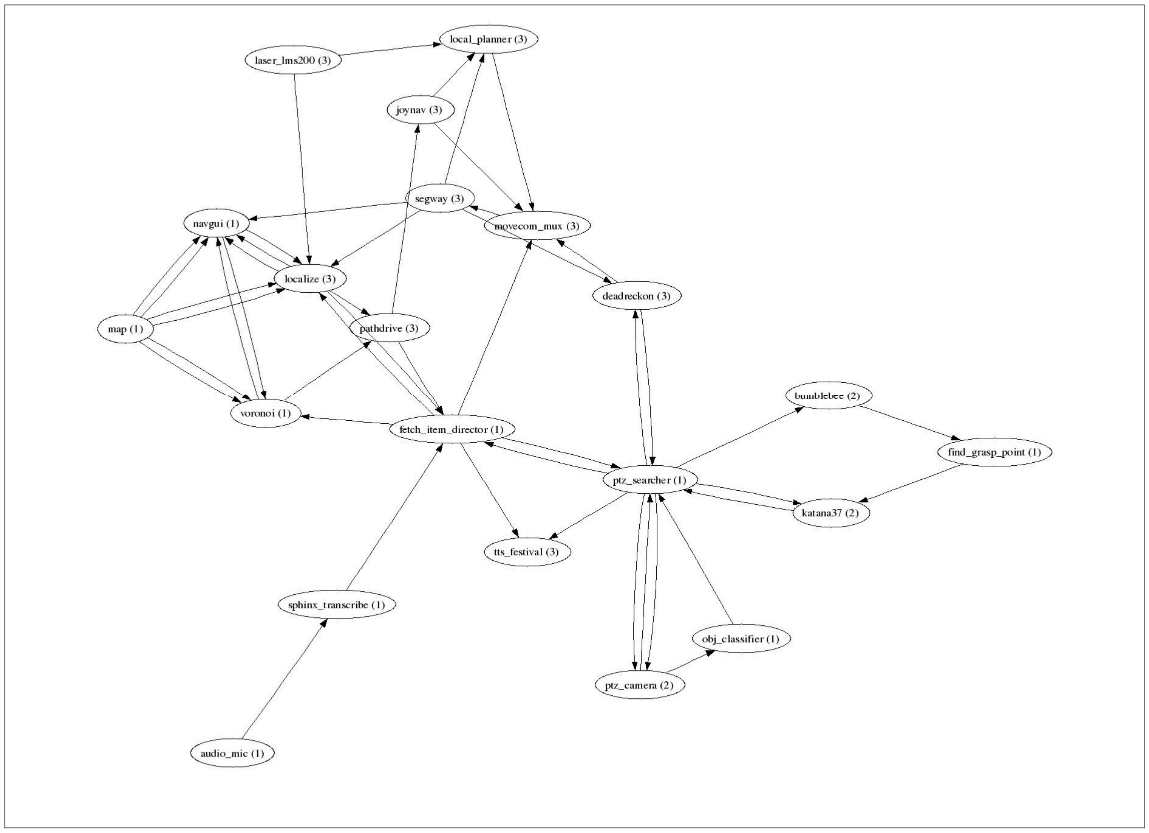

Figure 2-1. An hypothetical ROS graph for a fetch-an-item robot. Nodes in the graph represent individual programs; edges represent message streams.

To reiterate: a ROS graph *node* represents a software module that is sending or receiving messages, and a ROS graph *edge* represents a stream of messages between two nodes. Although things can get more complex, typically nodes are POSIX processes and edges are TCP connections.

To get started, consider the canonical first program, whose task is to just print "Hello, world!" to the console. It is instructive to think about how this is implemented. In UNIX, every program has a stream called "standard output," or `stdout`. When an interactive terminal runs a “hello, world” program, its `stdout` stream is received by the terminal program, which (nowadays) renders it in a *terminal emulator* window.

In ROS, this concept is extended so that programs can have an arbitrary number of streams, connected to an arbitrary number of other programs, any of which can start up or shut down at any time. To re-create the minimal “Hello, world!” system in ROS requires two *nodes* with one stream of text messages between them. `talker` will periodically send “Hello, world!” as a text message. Meanwhile, `listener` is diligently awaiting new text messages, and whenever one arrives, it simply prints it to the console. Whenever both of these programs are running, ROS would connect them as shown in world graph.

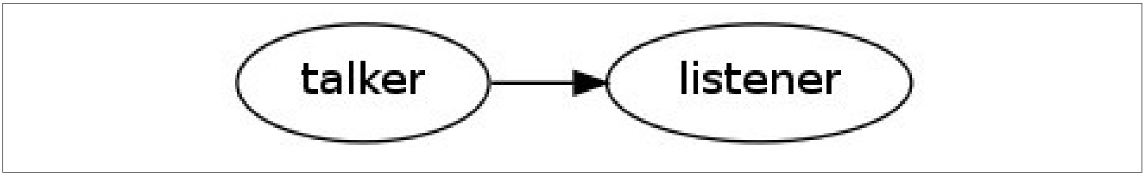

Figure 2-2. The simplest possible ROS graph: one program is sending messages to another program.

Indeed, we have created a gloriously complex “Hello, world!” system. It really doesn’t do much! But we can now demonstrate some benefits of the architecture. Imagine that you wanted to create a log file of these important “Hello, world!” messages. In ROS, nodes have **no idea** who they are connected to, where their messages are coming from, or where they are going. We can thus create a generic `logger` program which writes all its incoming strings to disk, and tie that to `talker` as well, as shown in Figure 2-3.

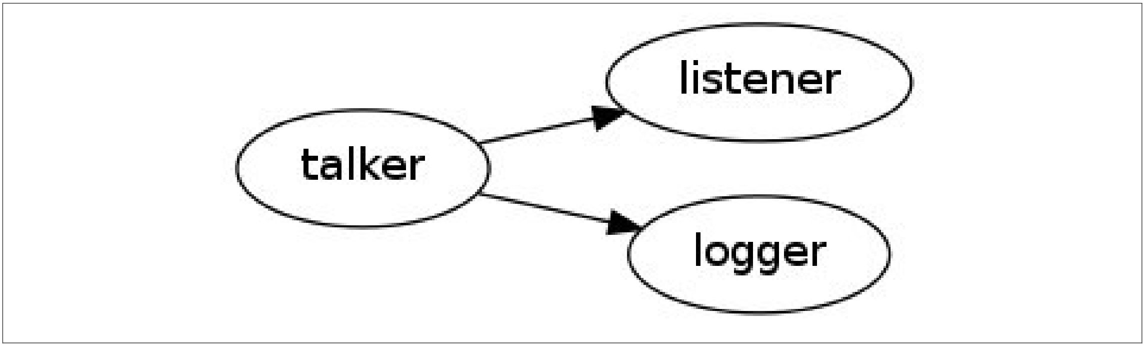

Figure 2-3. Hello, world! with a logging node.

Perhaps we wanted to run “Hello, world!” on two different computers, and have a single node receive both of their messages. Without having to modify any source code, we could just start `talker` twice, calling them “talker1” and “talker2,” respectively, and ROS will connect them as shown in Figure 2-4.

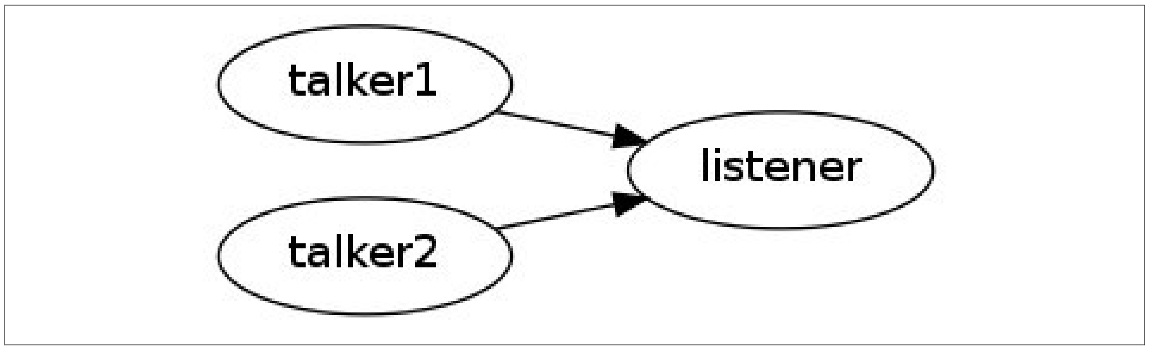

Figure 2-4. Instantiating two Hello, world! programs and routing them to the same receiver.

Perhaps we want to simultaneously log and print both of those streams? Again this can be accomplished without modifying any source code; we can command ROS to route the streams as shown in Figure 2-5.

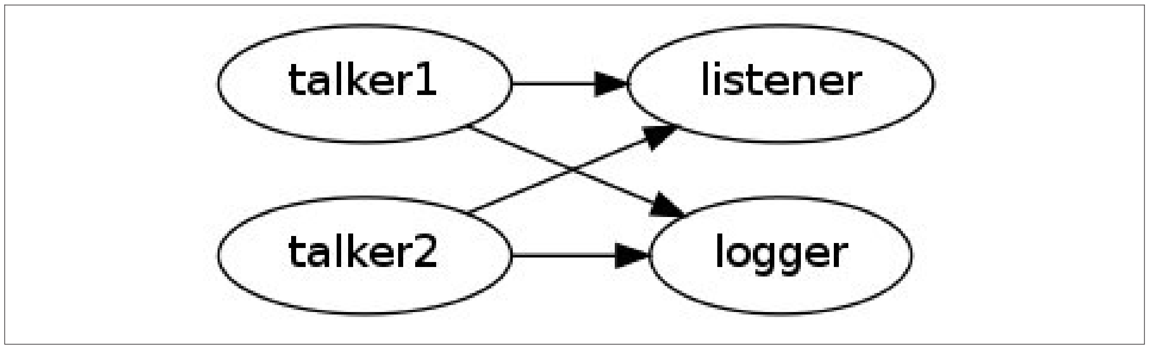

Figure 2-5. Now, there are two Hello, world! programs with two listeners. No source code has been modified.

“Hello, world!” programs are fun to write, but of course a typical robot is more complicated. For example, the “fetch a stapler” problem described at the beginning of this chapter was implemented in the early days of ROS using the exact graph previously shown as Figure 2-1. This system included 22 programs was running on four computers.

The navigation system is the upper half of Figure 2-1, and the vision and grasping systems are in the lower-right corner. It is interesting to note that this graph is *sparse*, meaning that most nodes connect to a very small number of other nodes. This property is commonly seen in ROS graphs, and can serve as a check on a software architecture: if the ROS graph starts looking like a star, where most nodes are streaming data to or from a central node, it is often worthwhile to re-assess the flow of data and separate functions into smaller pieces. The goal is to create small, manageable functional units, which ideally can be re-used in other applications on other robots.

Often, each graph node is running in its own POSIX process. This offers additional fault tolerance: a software fault will only take down its own process. The rest of the graph will stay up, passing messages as functioning as normal. The circumstances leading up to the crash can often be re-created by logging the messages entering a node, and simply playing them back at a later time inside a debugger.

However, perhaps the greatest benefit of a loosely-coupled, graph-based architecture is the ability to rapid-prototype complex systems with little or no software "glue" required for experimentation. Single nodes, such as the object-recognition node in the "fetch a stapler" example, can trivially be swapped by simply launching an entirely different process that accepts images and outputs labeled objects. Not only can a single node be swapped, but entire chunks of the graph can be torn down and replaced, even at runtime, with other large subgraphs. Real-robot hardware drivers can be replaced with simulators, navigation subsystems can be swapped, algorithms can be tweaked and recompiled, and so on. Since ROS is creating all the required network backend on-the-fly, the entire system is interactive and designed to encourage experimentation.

A ROS system is a graph of nodes, each doing some computation and sending messages to a small number of other nodes. Some nodes will be connected to sensors, and be sources of data for our system. Some will be connected to actuators, and will cause the robot to move. However, up until now, we have conveniently avoided the natural question: how do nodes find each other, so they can start passing messages? The answer lies in a program called `roscore`.

roscore

`roscore` is a broker that provides connection information to nodes so that they can transmit messages to each other. Nodes register details of the messages they provide, and those that they want to subscribe to, and this is stored in `roscore`. When a new node appears, `roscore` provides it with the information that it needs to form a direct peer-to-peer connection with other nodes with which it wants to swap messages. Every ROS system needs a running `roscore`. Without it, nodes cannot find other nodes to get messages from or send messages to.

When a ROS node starts up, it expects its process to have a POSIX environment variable named `ROS_MASTER_URI`. This is expected to be of the form `http://hostname:11311/` , which must point to a running instance of `roscore` somewhere on the network. Port 11311 was chosen as the default port for `roscore` simply because it was a palindromic prime. Different ports can be specified to allow multiple ROS systems to co-exist on a single LAN. With knowledge of the location of `roscore` on the network, nodes *register* themselves at startup with `roscore`, and can then query `roscore` to find other nodes and data streams by name. Each ROS node tells `roscore` which messages it provides, and which it would like to subscribe to. `roscore` then provides the addresses of the relevant message producers and consumers in Viewed in a graph form, every node in the graph can periodically call on services provided by `roscore` to find their peers. This is represented by the dashed lines shown in Figure 2-6.

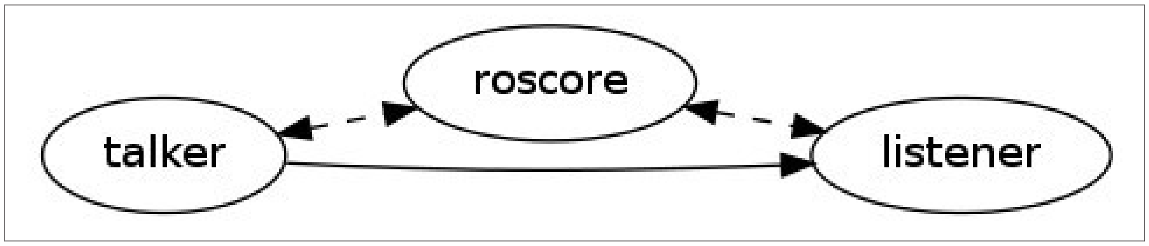

Figure 2-6. roscore connects only ephemerally to the other nodes in the system.

`roscore` also holds a *parameter server* which is used extensively by ROS nodes for configuration. The parameter server allows nodes to store and retrieve arbitrary data structures, such as descriptions of robots, parameters for algorithms, and so on. Like everything in ROS, there is a simple command-line tool to interact with the parameter server: `rosparam`, which will be used throughout the book.

We'll see examples of how to use `roscore` soon. For now, all you really need to remember about it is that it's a program that allows nodes to find other nodes. That brings is to the question of how you start a node in ROS, and to `rosrun`.

rosrun

Up until now, all command-line examples have assumed that the shell's working directory include the program of interest. Of course, this is not always the case, and it would be tiresome to have to continually `cd` all around the filesystem to chase down ROS programs. Since ROS has a large, distributed community, its software is organized into *packages*. The concept of a ROS *package* will be described in greater detail in subsequent chapters, but a package can be thought of as a collection of resources that are built and distributed together.

For example, the `talker` program lives in a packaged named `roscpp_tutorials`. ROS provides a command-line utility called `rosrun` which will search a package for the requested program, and pass it any and all other parameters supplied on the command line. The syntax is as follows:

```
$ rosrun PACKAGE EXECUTABLE [ARGS]
```

To run the `talker` program in the `rospy_tutorials` package, no matter where one happens to be in the filesystem, one could type:

```
$ rosrun rospy_tutorials talker
```

This is great for starting single ROS nodes, but real robot systems often consist of tens or hundreds of nodes, all running at the same time. Since it wouldn't be practical to call `rosrun` on each of these nodes, ROS includes a tool for starting collections of nodes, called `roslaunch`. However, before looking at `roslaunch`, we need to first talk about how things are named in ROS.

Names, Namespaces, and Remapping

Names are a fundamental concept in ROS. Nodes, data streams, and parameters must all have unique names. For example, the camera on a robot could be named `camera` and it could output a data stream named `image`, and read a parameter named `frame_rate` to know how fast to send images.

So far, so good. But, what happens when a fancy robot has two cameras? We wouldn't want to have to write a separate program for each camera, nor would we want the output of both cameras to be interleaved on the `image` topic.

More generally, namespace collisions are extremely common in robotic systems, which often contain several identical hardware or software modules that are used in different contexts. ROS provides two mechanisms to handle these situations: *namespaces* and *remapping*.

Namespaces are a fundamental concept throughout computer science. Following the convention of UNIX paths and Internet URI's, ROS uses the forward slash `/` to delimit namespaces. Just like how two files named `readme.txt` can exist in separate paths, such as `/home/username/readme.txt` and `/home/username/foo/readme.txt`, ROS can launch identical nodes into separate namespaces to avoid clashes.

In the previous example, a robot with two cameras could launch two camera drivers in separate namespaces, such as `left` and `right`, which would result in image streams named `left/image` and `right/image`.

This avoids a topic name clash, but how could we send these data streams to another program, which was still expecting to receive on topic `image` ? One answer would be to launch this other program in the same namespace, but perhaps this program needs to "reach into" more than one namespace. Enter *remapping*.

In ROS, any string in a program that defines a name can be *remapped* at run-time. As one example, there is a commonly-used program in ROS called `image_view` that renders a live video window of images being sent on the `image` topic. At least, that is what is written in the source code of the `image_view` program. Using remapping, we can instead cause the `image_view` program to render the `right/image` topic, or the `left/image` topic, without having to modify the source code of `image_view`!

The need to *remap* names is so common in ROS that there is a command-line shortcut for it. If the working directory contains the `image_view` program, one could type the following to map `image` to `right/image`:

```
$ ./image_view image:=right/image
```

This command-line *remapping* would produce the graph shown in Figure 2-7

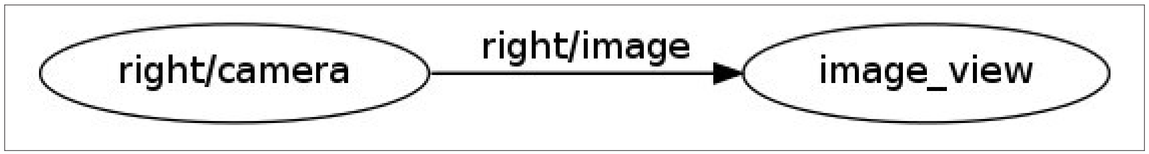

Figure 2-7. The image topic has been renamed right/image using command-line remapping.

Pushing a node into a namespace can be accomplished with a special `__ns` remapping syntax (note the double underscore). For example, if the working directory contains the `camera` program, the following shell command would launch `camera` into the namespace `right`:

```
$ ./camera __ns:=right
```

Just as for filesystems, web URLs, and countless other domains, ROS names must be unique. If the same node is launched twice, `roscore` directs the older node to exit, to make way for the newer instance of the node. Earlier in this chapter, a graph was shown that had two nodes, `talker1` and `talker2`, sending data to a node named `listener`. To change the *name* of a node on the command line, the special `__name` remapping syntax can be used (again, note the double underscore). The following two shell commands would launch two instances of `talker`, one named `talker1` and one named `talker2`:

```
$ ./talker __name:=talker1
```

```
$ ./talker __name:=talker2
```

The previous examples demonstrated that ROS topics can be remapped quickly and easily on the command line. This is useful for debugging and for initially hacking systems together when experimenting with various ideas. However, after typing long command-line strings a few times, it's time to automate them! The `roslaunch` tool was created for this purpose.

roslaunch

`roslaunch` is a command-line tool designed to automate the launching of collections of ROS nodes. On the surface, it looks a lot like `rosrun`, needing a package name and a file name:

```
roslaunch PACKAGE LAUNCH_FILE
```

However, `roslaunch` operates on *launch files* which are XML files describing a combination of nodes, topic remappings, and parameters. By convention, these files have a suffix of `.launch`. For example, here is `talker_listener.launch` in the `rospy_tutorials` package:

```
<launch>
  <node name="talker" pkg="rospy_tutorials"
        type="talker.py" output="screen" />
  <node name="listener" pkg="rospy_tutorials"
        type="listener.py" output="screen" />
</launch>
```

Each `<node>` tag includes attributes declaring the ROS graph name of the node, the package in which it can be found, and the *type* of node, which is simply the filename of the executable program. In this example, the `output="screen"` attributes indicate that the `talker` and `listener` nodes should dump their console outputs to the current console, instead of only to log files. This is a commonly-used setting for debugging; once things start working, it is often convenient to remove this attribute so that the console has less noise.

`roslaunch` has many other important features, such as the ability to launch programs on other computers across the network via `ssh`, automatically re-spawning nodes that crash, and so on. These features will be described throughout the book as they are necessary to accomplish various tasks. One of its most useful features is that it tears down the subgraph it launched when Ctrl+C is pressed in the console containing `roslaunch`. For example, the following command would cause `roslaunch` to spawn two nodes to form a talker-listener subgraph, as described in the `talker_listener.launch` file listed previously:

```
$ roslaunch rospy_tutorials talker_listener.launch
```

And, equally importantly, pressing Ctrl+C would tear down the subgraph! Virtually every time you use ROS, you will be calling `roslaunch` and pressing Ctrl+C in `roslaunch` terminals, to create and destroy subgraphs.

`roslaunch` will automatically instantiate a `roscore` if one does not exist when `roslaunch` is invoked. However, this `roscore` will exit when Ctrl+C is pressed in the `roslaunch` window. If one has more than one terminal open when launching ROS programs, it is often easier to remember to launch a `roscore` in a separate shell, which is left open the entire ROS session. Then, one can `roslaunch` and Ctrl+C with abandon in all other consoles, without risk of losing the `roscore` tying the whole system together.

Before we're start to look at writing some code with ROS, there's one more thing to cover, that will save you time and heartache as you try to remember the names of packages, nodes, and launch files: tab-completion.

The [TAB] key

The ROS command-line tools have tab-completion enabled. When running `rosrun`, for example, hitting the `[TAB]` key in the middle of typing that package name will auto-complete it for you, or present you with a list of possible completions. As with many other Linux commands, using tab-completion with ROS will save you a massive amount of time and avoid errors, especially when trying to type long package or message names. For example, typing

```
$ rosrun rospy_tutorials ta[TAB]
```

will auto-complete to

```
$rosrun rospy_tutorials talker
```

since no other programs in the `rospy_tutorials` package begin with `ta`. Additionally, `rosrun` (like virtually all ROS core tools) will auto-complete package names. For example, typing:

```
$ rosrun rospy_tu[TAB]
```

will auto-complete to

```
$ rosrun rospy_tutorials
```

since no other packages currently loaded begin with `rospy_tu`.

Summary

In this chapter, we looked at the ROS graph architecture, and introduced you to the tools that you're going to be using to interact with it, starting and stopping nodes. We also introduced the ROS namespace conventions, and showed how namespaces can be remapped to avoid collisions.

Now that you understand the underlying architecture of a ROS system, it's time to look at what sort of messages the nodes in this system might send to each other, how these messages are composed, sent, and received, and to think about some of the computations that the nodes might be doing. That brings us to *topics*, the fundamental communication method in ROS.

CHAPTER 3
Topics

As we saw in the previous chapter, ROS systems consist of a number of independent *nodes* that comprise a *graph*. These nodes by themselves are typically not very useful. Things only get interesting when nodes communicate with each other, exchanging information and data. The most common way to do that is through *topics*. A topic is a name for a stream of messages with a defined. For example, the data from a laser range-finders might be send on a topic called `scan`, with a message type of `LaserScan`, while the data from a camera might be sent over a topic called `image`, with a message type of `Image`.

Before they can start to transmit data over topics, nodes must first announce, or *advertize*, both the topic name and the type of messages that are going to be sent. Then they can start to send, or *publish*, the actual data on the topic. Nodes that want to receive messages on a topic can *subscribe* to that topic by making a request to `roscore`. After subscribing, all messages on the topic are delivered to the node that make the request. Topics implement a *publish/subcribe communications mechanism*, one of the more common ways to exchange data in a distributed system. One of the main advantages to using ROS is that `roscore` and the ROS client libraries handle all the messy details of setting up the necessary connections when nodes advertise or subscribe to topics, so that you don't have to worry about it.

In ROS, all messages on the same topic **must** be of the same data type. Although ROS does not enforce it, topic names often describe the messages that are sent over them. For example, on the PR2 robot, the topic `/wide_stereo/right/image_color` is used for color images from the rightmost camera of the wide-angle stereo pair.

We'll start off by looking at how a node advertizes a topic and publishes data on it.