A Gentle Introduction to ROS

Chapter: Writing ROS programs

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

This is version 2.1.5 (5378703), generated on April 12, 2018.

Typeset by the author using LaTeX and memoir.cls.

ISBN 978-14-92143-23-9

Chapter 3

Writing ROS programs

In which we write ROS programs to publish and subscribe to messages.

So far we've introduced a few core ROS features, including packages, nodes, topics, and messages. We also spent a bit of time exploring some existing software built on those features. Now it's finally time to begin creating your own ROS programs. This chapter describes how to set up a development workspace and shows three short programs, including the standard "hello world" example, and two that show how to publish and subscribe to messages.

3.1 Creating a workspace and a package

We saw in Section 2.4 that all ROS software, including software we create, is organized into packages. Before we write any programs, the first steps are to create a workspace to hold our packages, and then to create the package itself.

Creating a workspace Packages that you create should live together in a directory called a **workspace**.[1] For example, the author's workspace is a directory called `/home/jokane-/ros`, but you can name your workspace whatever you like, and store the directory anywhere in your account that you prefer. Use the normal `mkdir` command to create a directory. We'll refer to this new directory as your **workspace directory**.

[1] `http://wiki.ros.org/catkin/Tutorials/create_a_workspace`

⏭ *For many users, there's no real need to use more than one ROS workspace. However, ROS's* `catkin` *build system, which we'll introduce in Section 3.2.2, attempts to build all of the packages in a workspace at once. Therefore, if you're working on many packages or have several distinct projects, it may be useful to maintain several independent workspaces.*

One final step is needed to set up the workspace. Create a subdirectory called `src` inside the workspace directory. As you might guess, this subdirectory will contain the source code for your packages.

Creating a package The command to create a new ROS package, which should be run from the `src` directory of your workspace, looks like this:[2]

```
catkin_create_pkg package-name
```

Actually, this package creation command doesn't do much: It creates a directory to hold the package and creates two configuration files inside that directory.

- ☞ The first configuration file, called `package.xml`, is the manifest discussed in Section 2.4.
- ☞ The second file, called `CMakeLists.txt`, is a script for an industrial-strength cross-platform build system called `CMake`. It contains a list of build instructions including what executables should be created, what source files to use to build each of them, and where to find the include files and libraries needed for those executables. `CMake` is used internally by `catkin`.

In the coming sections, we'll see a few edits you'll need to make to each of these files to configure your new package. For now, it's enough to understand that `catkin_create_pkg` doesn't do anything magical. Its job is simply to make things a bit more convenient by creating both the package directory and default versions of these two configuration files.

This three-layered directory structure—a workspace directory, containing a `src` *directory, containing a package directory—might seem to be overkill for simple projects and small workspaces, but the* `catkin` *build system requires it.*

[2] `http://wiki.ros.org/ROS/Tutorials/CreatingPackage`

⏭ *ROS package names follow a naming convention that allows only lowercase letters, digits, and underscores. The convention also requires that the first character be a lowercase letter. A few ROS tools, including* `catkin`*, will complain about packages that do not follow this convention.*

All of the examples in this book belong to a package called `agitr`, named after the initials of the book's title. If you'd like to recreate this package yourself, you can create a package with this name running this command from your workspace's `src` directory:

```
catkin_create_pkg agitr
```

An alternative to creating the `agitr` package yourself is to download the archive of this package from the book's website, and expand it from within your workspace directory.

Editing the manifest After creating your package, you may want to edit its `package.xml`, which contains some metadata describing the package. The default version installed by `catkin_create_pkg` is liberally commented and largely self-explanatory. Note, however, that most of this information is not utilized by ROS, neither at build time nor at run time, and only becomes really important if you release your package publicly. In the spirit of keeping documentation in sync with actual functionality, a reasonable minimum might be to fill in the `description` and `maintainer` fields. Listing 3.1 shows the manifest from our `agitr` package.

3.2 Hello, ROS!

Now that we've created a package, we can start writing ROS programs.

3.2.1 A simple program

Listing 3.2 shows a ROS version of the canonical "Hello, world!" program. This source file, named `hello.cpp`, belongs in your package folder, right next to `package.xml` and `CMakeLists.txt`.

Some online tutorials suggest creating a `src` *directory within your package directory to contain C++ source files. This additional organization might be helpful, especially for larger packages with many types of files, but it isn't strictly necessary.*

```
<?xml version="1.0"?>
<package>
  <name>agitr</name>
  <version>0.0.1</version>
  <description>
    Examples from A Gentle Introduction to ROS
  </description>
  <maintainer email="jokane@cse.sc.edu">
    Jason O'Kane
  </maintainer>
  <license>TODO</license>
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>geometry_msgs</build_depend>
  <run_depend>geometry_msgs</run_depend>
  <build_depend>turtlesim</build_depend>
  <run_depend>turtlesim</run_depend>
</package>
```

Listing 3.1: The manifest (that is, package.xml) for this book's agitr package.

We'll see how to compile and run this program momentarily, but first let's examine the code itself.

☞ The header file ros/ros.h includes declarations of the standard ROS classes. You'll want to include it in every ROS program that you write.

☞ The ros::init function initializes the ROS client library. Call this once at the beginning of your program.[3] The last parameter is a string containing the default name of your node.

> ▶▶| *This default name can be overridden by a launch file (see page 87) or by a* rosrun *command line parameter (see page 23).*

☞ The ros::NodeHandle object is the main mechanism that your program will use to interact with the ROS system.[4] Creating this object registers your program as a

[3] http://wiki.ros.org/roscpp/Overview/Initialization and Shutdown
[4] http://wiki.ros.org/roscpp/Overview/NodeHandles

```
1  // This is a ROS version of the standard "hello, world"
2  // program.
3
4  // This header defines the standard ROS classes.
5  #include <ros/ros.h>
6
7  int main(int argc, char **argv) {
8    // Initialize the ROS system.
9    ros::init(argc, argv, "hello_ros");
10
11   // Establish this program as a ROS node.
12   ros::NodeHandle nh;
13
14   // Send some output as a log message.
15   ROS_INFO_STREAM("Hello, ROS!");
16 }
```

Listing 3.2: A trivial ROS program called hello.cpp.

node with the ROS master. The simplest technique is to create a single `NodeHandle` object to use throughout your program.

⏭ *Internally, the* `NodeHandle` *class maintains a reference count, and only registers a new node with the master when the first* `NodeHandle` *object is created. Likewise, the node is only unregistered when all of the* `NodeHandle` *objects have been destroyed. This detail has two impacts: First, you can, if you prefer, create multiple* `NodeHandle` *objects, all of which refer to the same node. There are occasionally reasons that this would make sense. An example of one such situation appears on page 129. Second, this means that it is not possible, using the standard* `roscpp` *interface, to run multiple distinct nodes within a single program.*

☞ The `ROS_INFO_STREAM` line generates an informational message. This message is sent to several different locations, including the console screen. We'll see more details about this kind of log message in Chapter 4.

3.2.2 Compiling the Hello program

How can you compile and run this program? This is handled by ROS's build system, called catkin. There are four steps.[5]

Declaring dependencies First, we need to declare the other packages on which ours depends. For C++ programs, this step is needed primarily to ensure that catkin provides the C++ compiler with the appropriate flags to locate the header files and libraries that it needs.

To list dependencies, edit the CMakeLists.txt in your package directory. The default version of this file has this line:

```
find_package(catkin REQUIRED)
```

Dependencies on other catkin packages can be added in a COMPONENTS section on this line:

```
find_package(catkin REQUIRED COMPONENTS package-names)
```

For the hello example, we need one dependency on a package called roscpp, which provides the C++ ROS client library. The required find_package line, therefore, is:

```
find_package(catkin REQUIRED COMPONENTS roscpp)
```

We should also list dependencies in the package manifest (package.xml), using the build_depend and run_depend elements:

```
<build_depend>package-name</build_depend>
<run_depend>package-name</run_depend>
```

In our example, the hello program needs roscpp both at build time and at run time, so the manifest should contain:

```
<build_depend>roscpp</build_depend>
<run_depend>roscpp</run_depend>
```

However, dependencies declared in the manifest are not used in the build process; if you omit them here, you likely won't see any error messages until you distribute your package to others who try to build it without having the required packages installed.

[5] http://wiki.ros.org/ROS/Tutorials/BuildingPackages

Declaring an executable Next, we need to add two lines to CMakeLists.txt declaring the executable we would like to create. The general form is

```
add_executable(executable-name source-files)
target_link_libraries(executable-name ${catkin_LIBRARIES})
```

The first line declares the name of the executable we want, and a list of source files that should be combined to form that executable. If you have more than one source file, list them all here, separated by spaces. The second line tells CMake to use the appropriate library flags (defined by the find_package line above) when linking this executable. If your package contains more than one executable, copy and modify these two lines for each executable you have.

In our example, we want an executable called hello, compiled from a single source file called hello.cpp, so we would add these lines to CMakeLists.txt:

```
add_executable(hello hello.cpp)
target_link_libraries(hello ${catkin_LIBRARIES})
```

For reference, Listing 3.3 shows a short CMakeLists.txt that suffices for our example. The default version of CMakeLists.txt created by catkin_create_pkg contains some commented-out guidance for a few other purposes; for many simple programs, something similar to the simple version shown here is enough.

Building the workspace Once your CMakeLists.txt is set up, you can build your workspace—including compiling all of the executables in all of its packages—using this command:

```
catkin_make
```

Because it's designed to build all of the packages in your workspace, this command must be run from your workspace directory. It will perform several configuration steps (especially the first time you run it) and create subdirectories called devel and build within your workspace. These two new directories contain build-related files like automatically-generated makefiles, object code, and the executables themselves. If you like, the devel and build subdirectories can safely be deleted when you've finished working on your package.

If there are compile errors, you'll see them here. After correcting them, you can catkin_make again to complete the build process.

If you see errors from catkin_make *that the header* ros/ros.h *cannot be found, or "undefined reference" errors on* ros::init *or other ROS functions, the most likely*

```
1  # What version of CMake is needed?
2  cmake_minimum_required(VERSION 2.8.3)
3
4  # Name of this package.
5  project(agitr)
6
7  # Find the catkin build system, and any other packages on
8  # which we depend.
9  find_package(catkin REQUIRED COMPONENTS roscpp)
10
11 # Declare our catkin package.
12 catkin_package()
13
14 # Specify locations of header files.
15 include_directories(include ${catkin_INCLUDE_DIRS})
16
17 # Declare the executable, along with its source files.  If
18 # there are multiple executables, use multiple copies of
19 # this line.
20 add_executable(hello hello.cpp)
21
22 # Specify libraries against which to link.  Again, this
23 # line should be copied for each distinct executable in
24 # the package.
25 target_link_libraries(hello ${catkin_LIBRARIES})
```

Listing 3.3: The CMakeLists.txt to build hello.cpp.

reason is that your CMakeLists.txt *does not correctly declare a dependency on* roscpp.

Sourcing setup.bash The final step is to execute a script called setup.bash, which is created by catkin_make inside the devel subdirectory of your workspace:

```
source devel/setup.bash
```

This automatically-generated script sets several environment variables that enable ROS to find your package and its newly-generated executables. It is analogous to the global setup.bash from Section 2.2, but tailored specifically to your workspace. Unless the direc-

tory structure changes, you only need to do this only once in each terminal, even if you modify the code and recompile with `catkin_make`.

3.2.3 Executing the `hello program`

When all of those build steps are complete, your new ROS program is ready to execute using `rosrun` (Section 2.6), just like any other ROS program. In our example, the command is:

```
rosrun agitr hello
```

The program should produce output that looks something like this:

```
[ INFO] [1416432122.659693753]: Hello, ROS!
```

Don't forget to start `roscore` first: This program is a node, and nodes need a master to run correctly. By the way, the numbers in this output line represent the time—measured in seconds since January 1, 1970—when our `ROS_INFO_STREAM` line was executed.

This `rosrun`*, along with some other ROS commands, may generate an error that looks like this:*

```
[rospack] Error: stack/package package-name not found
```

Two common causes of this error are (a) misspelling the package name, and (b) failing to run the `setup.bash` *for your workspace.*

3.3 A publisher program

The `hello` program from the previous section showed how to compile and run a simple ROS program. That program was useful as an introduction to `catkin`, but, like all "Hello, World!" programs, it didn't do anything useful. In this section, we'll look at a program that interacts with ROS a bit more.[6] Specifically, we'll see how to send randomly-generated velocity commands to a `turtlesim` turtle, causing it to wander aimlessly. The brief C++ source code for the program, called `pubvel`, appears as Listing 3.4. This program shows all of the elements needed to publish messages from code.

[6] `http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(c++)`

```
1  // This program publishes randomly-generated velocity
2  // messages for turtlesim.
3  #include <ros/ros.h>
4  #include <geometry_msgs/Twist.h>  // For geometry_msgs::Twist
5  #include <stdlib.h> // For rand() and RAND_MAX
6
7  int main(int argc, char **argv) {
8    // Initialize the ROS system and become a node.
9    ros::init(argc, argv, "publish_velocity");
10   ros::NodeHandle nh;
11
12   // Create a publisher object.
13   ros::Publisher pub = nh.advertise<geometry_msgs::Twist>(
14     "turtle1/cmd_vel", 1000);
15
16   // Seed the random number generator.
17   srand(time(0));
18
19   // Loop at 2Hz until the node is shut down.
20   ros::Rate rate(2);
21   while(ros::ok()) {
22     // Create and fill in the message.  The other four
23     // fields, which are ignored by turtlesim, default to 0.
24     geometry_msgs::Twist msg;
25     msg.linear.x = double(rand())/double(RAND_MAX);
26     msg.angular.z = 2*double(rand())/double(RAND_MAX) - 1;
27
28     // Publish the message.
29     pub.publish(msg);
30
31     // Send a message to rosout with the details.
32     ROS_INFO_STREAM("Sending␣random␣velocity␣command:"
33       << "␣linear=" << msg.linear.x
34       << "␣angular=" << msg.angular.z);
35
36     // Wait until it's time for another iteration.
37     rate.sleep();
38   }
39 }
```

Listing 3.4: A program called pubvel.cpp that publishes randomly generated movement commands for a turtlesim turtle.

3.3.1 Publishing messages

The main differences between pubvel and hello all stem from the need to publish messages.

Including the message type declaration You'll likely recall from Section 2.7.2 that every ROS topic is associated with a message type. Each message type has a corresponding C++ header file. You'll need to #include this header for every message type used in your program, with code like this:

#include <*package_name*/*type_name*.h>

Note that the package name should be the name of the package containing the message type, and not (necessarily) the name of your own package. In pubvel, we want to publish messages of type geometry_msgs/Twist—a type named Twist owned by a package named geometry_msgs—so we need this line:

#include <geometry_msgs/Twist.h>

The purpose of this header is to define a C++ class that has the same data members as the fields of the given message type. This class is defined in a namespace named after the package. The practical impact of this naming is that when referring to message classes in C++ code, you'll use the double colon (::)—also called the **scope resolution operator**—to separate the package name from the type name. In our pubvel example, the header defines a class called geometry_msgs::Twist.

Creating a publisher object The work of actually publishing the messages is done by an object of class ros::Publisher.[17] A line like this creates the object we need:

ros::Publisher pub = *node_handle*.advertise<*message_type*>(
topic_name, *queue_size*);

Let's have a look at each part of this line.

- The *node_handle* is an object of class ros::NodeHandle, one that you created near the start of your program. We're calling the advertise method of that object.
- The *message_type* part inside the angle brackets—formally called the template parameter—is the data type for the messages we want to publish. This should be the name of the class defined in the header discussed above. In the example, we use the geometry_msgs::Twist class.

[17] http://wiki.ros.org/roscpp/Overview/Publishers and Subscribers

☞ The *topic_name* is a string containing the name of the topic on which we want to publish. It should match the topic names shown by `rostopic list` or `rqt_graph`, but (usually) without the leading slash (`/`). We drop the leading slash to make the topic name a **relative name**; Chapter 5 explains the mechanics and purposes of relative names. In the example, the topic name is `"turtle1/cmd_vel"`.

Be careful about the difference between the topic name and the message type. If you accidentally swap these two, you'll get lots of potentially confusing compile errors.

☞ The last parameter to `advertise` is an integer representing the size of the **message queue** for this publisher. In most cases, a reasonably large value, say 1000, is suitable. If your program rapidly publishes more messages than the queue can hold, the oldest unsent messages will be discarded.

▶▶| *This parameter is needed because, in most cases, the message must be transmitted to another node. This communication process can be time consuming, especially compared to the time needed to create messages. ROS mitigates this delay by having the* `publish` *method (see below) store the message in an "outbox" queue and return right away. A separate thread behind the scenes actually transmits the message. The integer value given here is the number of messages—and not, as you might guess, the number of bytes—that the message queue can hold.*

Interestingly, the ROS client library is smart enough to know when the publisher and subscriber nodes are part of the same underlying process. In these cases, the message is delivered directly to the subscriber, without using any network transport. This feature is very important for making ***nodelets***[8] *— that is, multiple nodes that can be dynamically loaded into a single process—efficient.*

If you want to publish messages on multiple topics from the same node, you'll need to create a separate `ros::Publisher` object for each topic.

[8] `http://wiki.ros.org/nodelet`

Be mindful of the lifetime of your ros::Publisher *objects. Creating the publisher is an expensive operation, so it's a usually bad idea to create a new* ros::Publisher *object each time you want to publish a message. Instead, create one publisher for each topic, and use that publisher throughout the execution of your program. In* pubvel*, we accomplish this by declaring the publisher outside of the* while *loop.*

Creating and filling in the message object Next, we create the message object itself. We already referred to the message class when we created the ros::Publisher object. Objects of that class have one publicly accessible data member for each field in the underlying message type.

We used rosmsg show (Section 2.7.2) to see that the geometry_msgs/Twist message type has two top-level fields (linear and angular), each of which contains three sub-fields (x, y, and z). Each of these sub-fields is a 64-bit floating point number, called a double by most C++ compilers. The code in Listing 3.4 creates a geometry_msgs::Twist object and assigns pseudo-random numbers to two of these data members:

```
geometry_msgs::Twist msg;
msg.linear.x = double(rand())/double(RAND_MAX);
msg.angular.z = 2*double(rand())/double(RAND_MAX) - 1;
```

This code sets the linear velocity to a number between 0 and 1, and the angular velocity to a number between −1 and 1. Because turtlesim ignores the other four fields (msg.linear.y, msg.linear.z, msg.angular.x, and msg.angular.y), we leave them with their default value, which happens to be zero.

Of course, most message types have fields with types other than float64. Fortunately, the mapping from ROS field types to C++ types works precisely the way you might expect.[9] One fact that may not be obvious is that fields with array types—shown with square brackets by rosmsg show—are realized as STL vectors in C++ code.

Publishing the message After all of that preliminary work, it is very simple to actually publish the message, using the publish method of the ros::Publisher object. In the example, it looks like this:

```
pub.publish(msg);
```

This method adds the given msg the publisher's outgoing message queue, from which it will be sent as soon as possible to any subscribers of the corresponding topic.

[9] http://wiki.ros.org/msg

Formatting the output Although it's not directly related to publishing our velocity commands, the ROS_INFO_STREAM line in Listing 3.4 is worth a look. This is a more complete illustration of what ROS_INFO_STREAM can do, because it shows the ability to insert data other than strings—in this case, the specific randomly generated message fields—into the output. Section 4.3 has more information about how ROS_INFO_STREAM works.

3.3.2 The publishing loop

The previous section covered the details of message publishing. Our `pubvel` example repeats the publishing steps inside a `while` loop to publish many different messages as time passes. The program uses a two additional constructs to form this loop.

Checking for node shutdown The condition of `pubvel`'s while loop is:

```
ros::ok()
```

Informally, this function checks whether our program is still in "good standing" as a ROS node. It will return `true`, until the node has some reason to shut down. There are a few ways to get `ros::ok()` to return `false`:

- ☞ You can use `rosnode kill` on the node.
- ☞ You can send an interrupt signal (Ctrl-C) to the program.

  ▶▶| *Interestingly,* `ros::init()` *installs a handler for this signal, and uses it to initiate a graceful shutdown. The impact is that* Ctrl-C *can be used to make* `ros::ok()` *return* `false`*, but does not immediately terminate the program. This can be important if there are clean-up steps—Writing log files, saving partial results, saying goodbye,* etc*—that should happen before the program exits.*

- ☞ You can call, somewhere in the program itself,

  ```
  ros::shutdown()
  ```

  This function can be a useful way to signal that your node's work is complete from deep within your code.
- ☞ You can start another node with the same name. This usually happens if you start a new instance of the same program.

Controlling the publishing rate The last new element of `pubvel` is its use of a `ros::Rate` object:[10]

```
ros::Rate rate(2);
```

This object controls how rapidly the loop runs. The parameter in its constructor is in units of Hz, that is, in cycles per second. This example creates a rate object designed to regulate a loop that executes two iterations per second. Near the end of each loop iteration, we call the `sleep` method of this object:

```
rate.sleep();
```

Each call to the this method causes a delay in the program. The duration of the delay is calculated to prevent the loop from iterating faster than the specified rate. Without this kind of control, the program would publish messages as fast as the computer allows, which can overwhelm publish and subscribe queues and waste computation and network resources. (On the author's computer, an unregulated version of this program topped out around 6300 messages per second.)

You can confirm that this regulation is working correctly, using `rostopic hz`. For `pubvel`, the results should look similar to this:

```
average rate: 2.000
    min: 0.500s max: 0.500s std dev: 0.00006s window: 10
```

We can see that our messages are being published at a rate of two per second, with very little deviation from this schedule.

You might be thinking of an alternative to `ros::Rate` *that uses a simple, fixed delay—perhaps generated by* `sleep` *or* `usleep`*—in each loop iteration. The advantage of a* `ros::Rate` *object over this approach is that* `ros::Rate` *can account for the time consumed by other parts of the loop. If there is nontrivial computation to be done in each iteration (as we would expect from a real program), the time consumed by this computation is subtracted from the delay. In extreme cases, in which the real work of the loop takes longer than the requested rate, the delay induced by* `sleep()` *can be reduced to zero.*

[10] `http://wiki.ros.org/roscpp/Overview/Time`

3.3.3 Compiling pubvel

The process of building pubvel is mostly the same as for hello: Modify CMakeLists.txt and package.xml, and then use catkin_make to build your workspace. There is, however, one important difference from hello.

Declaring message type dependencies Because pubvel uses a message type from the geometry_msgs package, we must declare a dependency on that package. This takes the same form as the roscpp dependency discussed in Section 3.2.2. Specifically, we must modify the find_package line in CMakeLists.txt to mention geometry_msgs in addition to roscpp:

```
find_package(catkin REQUIRED COMPONENTS roscpp geometry_msgs)
```

Note that we are modifying the existing find_package line, rather than creating a new one. In package.xml, we should add elements for the new dependency:

```
<build_depend>geometry_msgs</build_depend>
<run_depend>geometry_msgs</run_depend>
```

If you skip (or forget) this step, then catkin_make *may not be able to find the header file* geometry_msgs/Twist.h*. When you see errors about missing header files, it's a good idea to verify the dependencies of your package.*

3.3.4 Executing pubvel

At last, we're ready to run pubvel. As usual, rosrun can do the job.

```
rosrun agitr pubvel
```

You'll also want to run a turtlesim simulator, so that you can see the turtle respond to the motion commands that pubvel publishes:

```
rosrun turtlesim turtlesim_node
```

Figure 3.1 shows an example of the results.

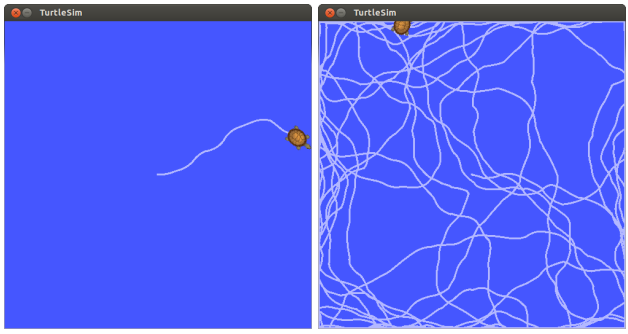

Figure 3.1: A turtlesim turtle responding to random velocity commands from pubvel.

3.4 A subscriber program

So far, we've seen an example program that publishes messages. This is, of course, only half of the story when it comes to communicating with other nodes via messages. Let's take a look now at a program that *subscribes* to messages published by other nodes.[11]

Continuing to use turtlesim as a test platform, we'll subscribe to the /turtle1/pose topic, on which turtlesim_node publishes.[1] Messages on this topic describe the **pose**—a term referring to position and orientation—of the turtle. Listing 3.5 shows a short program that subscribes to those messages and summarizes them for us via ROS_INFO_STREAM. Although some parts of this program should be familiar by now, there are three new elements.

Writing a callback function One important difference between publishing and subscribing is that a subscriber node doesn't know when messages will arrive. To deal with this fact, we must place any code that responds to incoming messages inside a **callback function**, which ROS calls once for each arriving message. A subscriber callback function looks like this:

```
void function_name(const package_name::type_name &msg) {
    ...
}
```

The *package_name* and *type_name* are the same as for publishing: They refer to the message class for the topic to which we plan to subscribe. The body of the callback func-

[1] How do we know that turtlesim_node publishes on this topic? One way to find out is to start that node and then use rostopic list, rosnode info, or rqt_graph to see the topics being published. See Section 2.7.1.

[11] http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(c++)

```
1  // This program subscribes to turtle1/pose and shows its
2  // messages on the screen.
3  #include <ros/ros.h>
4  #include <turtlesim/Pose.h>
5  #include <iomanip> // for std::setprecision and std::fixed
6
7  // A callback function.  Executed each time a new pose
8  // message arrives.
9  void poseMessageReceived(const turtlesim::Pose& msg) {
10   ROS_INFO_STREAM(std::setprecision(2) << std::fixed
11     << "position=(" <<  msg.x << "," << msg.y << ")"
12     << "␣direction=" << msg.theta);
13 }
14
15 int main(int argc, char **argv) {
16   // Initialize the ROS system and become a node.
17   ros::init(argc, argv, "subscribe_to_pose");
18   ros::NodeHandle nh;
19
20   // Create a subscriber object.
21   ros::Subscriber sub = nh.subscribe("turtle1/pose", 1000,
22     &poseMessageReceived);
23
24   // Let ROS take over.
25   ros::spin();
26 }
```

Listing 3.5: A ROS program called subpose.cpp that subscribes to pose data published by a turtlesim robot.

tion then has access to all of the fields in the received message, and can store, use, or discard that data as it sees fit. As always, we must include the appropriate header that defines this class.

In the example, our callback accepts messages of type turtlesim::Pose, so the header we need is turtlesim/Pose.h. (We can learn that this is the correct message type using rostopic info; recall Section 2.7.2.) The callback simply prints out some data from the message, including its x, y, and theta data members, via ROS_INFO_STREAM. (We can learn what data fields the message type has using rosmsg show, again from Section 2.7.2.) A real program would, of course, generally do some meaningful work with the message.

Notice that subscriber callback functions have a `void` return type. A bit of thought should confirm that this makes sense. Since it's ROS's job to call this function, there's no place in our program for any non-`void` return value to go.

Creating a subscriber object To subscribe to a topic, we create a `ros::Subscriber` object:[12]

```
ros::Subscriber sub = node_handle.subscribe(topic_name,
    queue_size, pointer_to_callback_function);
```

This line has several moving parts (most of which have analogues in the declaration of a `ros::Publisher`):

- The *node_handle* is the same node handle object that we've seen several times already.
- The *topic_name* is the name of the topic to which we want to subscribe, in the form of a string. This example uses `"turtle1/pose"`. Again, we omit the leading slash to make this string a relative name.
- The *queue_size* is the integer size of the message queue for this subscriber. Usually, you can use a large value like 1000 without worrying too much about the queuing process.

> *When new messages arrive, they are stored in a queue until ROS gets a chance to execute your callback function. This parameter establishes a maximum number of messages that ROS will store in that queue at one time. If new messages arrive when the queue is full, the oldest unprocessed messages will be dropped to make room. This may seem, on the surface, to be very similar to the technique used for publishing messages—See page 50—but differs in an important way: The rate at which ROS can empty a publishing queue depends on the time taken to actually transmit the messages to subscribers, and is largely out of our control. In contrast, the speed with which ROS empties a subscribing queue depends on how quickly we process callbacks. Thus, we can reduce the likelihood of a subscriber queue overflowing by (a) ensuring that we allow callbacks to occur, via* `ros::spin` *or* `ros::spinOnce`*, frequently, and (b) reducing the amount of time consumed by each callback.*

[12] `http://wiki.ros.org/roscpp/Overview/Publishers and Subscribers`

☞ The last parameter is a pointer to the callback function that ROS should execute when messages arrive. In C++, you can get a pointer to a function using the ampersand (&, "address-of") operator on the function name. In our example, it looks like this:

```
&poseMessageReceived
```

Don't make the common mistake of writing `()` *(or even* `(msg)`*) after the function name. Those parentheses (and arguments) are needed only when you actually want to call a function, not when you want to get a pointer to a function without calling it, as we are doing here. ROS supplies the required arguments when it calls your callback function.*

⏭ *Comment on C++ syntax: The ampersand is actually optional, and many programs omit it. The compiler can tell that you want a pointer to the function, rather than the value returned from executing the function, because the function name is not followed by parentheses. The author's suggestion is to include it, because it makes the fact that we're dealing with a pointer more obvious to human readers.*

You might notice that, while creating a `ros::Subscriber` object, we do not explicitly mention the message type anywhere. In fact, the `subscribe` method is templated, and the C++ compiler infers the correct message type based on the data type of the callback function pointer we provide.

If you use the wrong message type as the argument to your callback function, the compiler will not be able to detect this error. Instead, you'll see run-time error messages complaining about the type mismatch. These errors could, depending on the timing, come from either the publisher or subscriber nodes.

One potentially counterintuitive fact about `ros::Subscriber` objects is that it is quite rare to actually call any of their methods. Instead, the *lifetime* of that object is the most

relevant part: When we construct a ros::Subscriber, our node establishes connections with any publishers of the named topic. When the object is destroyed—either by going out of scope, or by a delete of an object created by the new operator—those connections are dropped.

Giving ROS control The final complication is that ROS will only execute our callback function when we give it explicit permission to do so.[13] There are actually two slightly different ways to accomplish this. One version looks like this:

```
ros::spinOnce();
```

This code asks ROS to execute all of the pending callbacks from all of the node's subscriptions, and *then return control back to us*. The other option looks like this:

```
ros::spin();
```

This alternative to ros::spinOnce() asks ROS to wait for and execute callbacks *until the node shuts down*. In other words, ros::spin() is roughly equivalent to this loop:

```
while(ros::ok()) {
    ros::spinOnce();
}
```

The question of whether to use ros::spinOnce() or ros::spin() comes down to this: Does your program have any repetitive work to do, other than responding to callbacks? If the answer is "No," then use ros::spin(). If the answer is "Yes," then a reasonable option is to write a loop that does that other work and calls ros::spinOnce() periodically to process callbacks. Listing 3.5 uses ros::spin() because that program's only job is to receive and summarize incoming pose messages.

A common error in subscriber programs is to mistakenly omit both ros::spinOnce *and* ros::spin. *In this case, ROS never has an opportunity to execute your callback function. Omitting* ros::spin *will likely cause your program to exit shortly after it starts. Omitting* ros::spinOnce *might make it appear as though no messages are being received.*

[13] http://wiki.ros.org/roscpp/Overview/Callbacks and Spinning

```
1 [ INFO] [1370972120.089584153]: position=(2.42,2.32) direction=1.93
2 [ INFO] [1370972120.105376510]: position=(2.41,2.33) direction=1.95
3 [ INFO] [1370972120.121365352]: position=(2.41,2.34) direction=1.96
4 [ INFO] [1370972120.137468325]: position=(2.40,2.36) direction=1.98
5 [ INFO] [1370972120.153486499]: position=(2.40,2.37) direction=2.00
6 [ INFO] [1370972120.169468546]: position=(2.39,2.38) direction=2.01
7 [ INFO] [1370972120.185472204]: position=(2.39,2.39) direction=2.03
```

Listing 3.6: Sample output from `subpose`, showing gradual changes in the robot's pose.

3.4.1 Compiling and executing `subpose`

This program can be compiled and executed just like the first two examples we've seen.

Don't forget to ensure that your package has a dependency on `turtlesim`*, which is needed because we're using the* `turtlesim/Pose` *message type. See Section 3.3.3 for a reminder of how to declare this dependency.*

A sample of the program's output, from when both `turtlesim_node` and `pubvel` were also running, appears as Listing 3.6.

3.5 Looking forward

This chapter's intent was to show how to write, compile, and execute a few simple programs, including programs that perform the core ROS operations of publishing and subscribing. Each of these programs used a macro called `ROS_INFO_STREAM` to generate informational log messages. In the next chapter, we'll examine ROS's logging system, of which `ROS_INFO_STREAM` is just a small part, more completely.