Programming Robots with ROS, Morgan Quigley, Brian Gerkey & William D. Smart

O'Reilly December 2015

CHAPTER 23

Using C++ in ROS

We chose to use Python for this book for a number of reasons. First, it's an accessible language for people without a lot of computer science background. Second, it has a lot of useful stuff in the core packages, which lets us concentrate on higher-level concepts. Third, ROS has strong support for Python. Fourth, we wanted to pick a single language for all of the examples in the book, and Python seemed like a reasonable choice.

However, sometimes you're going to want to use another language for your ROS development. Maybe some library that you need to use doesn't have Python support. Maybe you're more comfortable developing in another language. Maybe you want the (often slight) speed advantage that a compiled language brings. In this chapter, we're going to look at how the API in C++, one of the other supported languages, differs from the Python API, and how you can translate the examples in this book to C++. All of the idioms and design patterns for C++, and any other language that has a ROS API, will be the same: we're still going to use callbacks, we're still going to pass messages over topics, and so on. However, the syntax and specific data structures will be a little different. Once you learn how to map the Python examples onto your language of choice, then you'll be able to easily translate examples from one language to another.

The two best-supported language APIs in ROS are for Python and C++. In this chapter, we'll concentrate on the C++ API, but many of the things that we talk about will apply to APIs in other languages. Once you figure out the syntax and data structure differences, things will start to look the same, and you'll be able to change languages at will.

When Should You Use C (or Some Other Language)?

When should you use C++, or one of the other supported languages? The short answer is: when it makes your life easier. Since ROS is inherently a distributed system, it's easy to mix nodes written in different languages within the same system, with the messaging system (topics, services, and actions) acting as the glue that holds everything together.

Sometimes you will have a sensor or actuator with an API in C or C++, and it will be much easier to wrap this up into a ROS node if you use C++. Or, if you're new to Python but have years of C++ coding experience, you might just be more efficient writing code in C++. Similarly, if you're making extensive use of code that's written in C++, then it's easier to wrap this up in a C++ node. You might even be forced to use C++ because you're maintaining or extending a package that someone else wrote in C++.

Sometimes, especially if you're doing complex mathematical calculations, you'll want to write a node in C++ to make it faster. Be careful about this, though, since Python libraries like `scipy` are already very well optimized and will most likely be running the same code as your C++ implementation under the hood. Python does introduce some slowness, but you should be objective when you make the decision to imple-

ment something in C++. A C++ node might be faster than a similar Python node, but does the speed increase justify the extra development time of writing and debugging the C++ node?

Whatever your reasons for using C++ in ROS, whether they're driven by program- ming language zealotry or by cold, hard facts, let's look at how to write and build a ROS node with C++.

Building C++ with catkin

The main difference between C++ and Python (for our purposes, at least) is that C++ is a compiled language, while Python is an interpreted one. This means that you' re going to be interacting more with catkin and the ROS build system when you' re using C++. Every time you make a change to your code, you' re going to have to recompile it using catkin_make, and depending on the changes that you' ve made, you might also have to edit some other files.

This need to recompile is, in our opinion, one of the reasons to prefer Python for development. You can iterate on changes faster with Python because you don't have to recompile your code. ROS is a big software system, and if your node is complex and has many dependencies, your compile might take a few minutes. This will inevitably slow down your development process a bit.

Again, you need to add lines like this for each of the executables you build. Once You've got this information in place, then you're ready to build your node.

Putting our biases to one side for the moment, let's look at the files you need to edit when using C++.

package.xml

The *package.xml* file is the place where you declare all of your dependencies. When using C++, you have to declare both a build and a runtime dependency on `roscpp`:

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

You can either do this manually, by editing the file, or have `catkin_create_pkg` do it for you when you create the package:

```
user@hostname$ catkin_create_pkg <package name> roscpp
```

You'll also need to add in dependencies, both build and runtime, for any additional packages that you use in your node, just as you did when using Python.

CMakeLists.txt

You'll also need to add to the *CMakeLists.txt* file, so that the build system knows what you're trying to do and where to find things. In particular, you need to modify the file in the directory where your *src* directory lives (where your *package.xml* file also lives), not the one at the top of your `catkin` workspace. Suppose you're going to build a node called `minimal` from a single source file, *minimal.cpp*. You first have to let the build system know about the executable, and all of the files needed to build it:

```
add_executable(minimal
  src/minimal.cpp
)
```

This tells the build system that you're going to build an executable called `minimal` from the file *minimal.cpp*. If you have more than one executable in your package, you need to add lines like this for each one. If an executable is built from more than a single source file, you need to list these files in the body of `add_executable()`.

You also need to tell the build system about any link dependencies that you have. At a minimum, this will be the set of dependencies that `catkin` has worked out for you, based on the build dependencies in your *package.xml* file:

```
target_link_libraries(minimal
  ${catkin_LIBRARIES}
)
```

Again, you need to add lines like this for each of the executables you build. Once you've got this information in place, then you're ready to build your node.

catkin_make

To build your node, invoke catkin_make from the root of your catkin workspace. This will build your code, and make sure that everything that you depend upon is up to date. To make things easier on you, you should structure your directories according to ROS Enhancement Proposal (REP) 128. Basically, this means that there should be a directory called src in your catkin workspace directory. Individual package directories should live in this src directory. Within a package directory, there should be a package.xml, a CMakeLists.txt, and another src directory (where your source code actually lives):

```
catkin_workspace/
src/
CMakeLists.txt
package_1/
CMakeLists.txt
package.xml
...
package_n/
CMakeLists.txt
package.xml
build/
devel/
```

You invoke `catkin_make` from *catkin_workspace*. This will build your `minimal` executable and place it in *catkin_workspace/devel/lib/<package name>/minimal*.

Now that we've seen how to build a C++ node, let's look at what goes into the node itself, and how to translate from the Python examples in this book to C++.

Translating from Python to C++ (and Back Again)

To understand how to translate from the Python examples in this book to C++, you only really need to know three things: how a node is put together, how the three communication mechanisms are defined, and how to translate the data structures from one language to another. We'll start by looking at how to write a minimal node in C++.

A Simple Node

Example 23-1 shows the code for a minimal C++ node in ROS.

Example 23-1. minimal.cpp

```
#include <ros/ros.h>
int main(int argc, char **argv) {
ros::init(argc, argv, "minimal");
ros::NodeHandle n;
ros::spin();
return 0;
}
```

Include the basic ROS header information.

Initialize the node, and give it a name.

Create a node handle.

Give control over to ROS.

All ROS C++ nodes need to include the ros.h header file. Nodes are initialized by a call to init(), giving the command-line arguments and a name for the node. Then, we create a node handle that allows us to create topics, services, and actions. We didn't have to explicitly create a node handle when using Python, since the language was able to do it for us behind the scenes. This is one of the recurring themes when using C++: things often need to be more explicitly specified.
We need to add both build and runtime dependencies on roscpp to the package.xml file, and modify our CMakeLists.txt to contain the information shown in

Example 23-2.

Example 23-2. CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8.3)
project(cpp)
find_package(catkin REQUIRED roscpp)
add_executable(minimal src/minimal.cpp)
```

```
target_link_libraries(minimal
  ${catkin_LIBRARIES}
)
```

In this example, our package is called `cpp`. Once all of this information is in place, we can `cd` to our top-level `catkin` workspace and invoke `catkin_make`. This will compile our code and make sure all of the dependencies are up to date. Once this is done, we can find the resulting executable in *devel/lib/cpp/minimal*, and we can run it with `rosrun` as usual:

```
user@hostname$ rosrun cpp minimal
```

Topics

Example 23-3 shows how to set up a topic publisher in C++. The basic approach (set up the node, define the publisher, publish in a loop) is the same as in Python, but the details are a little different.

Example 23-3. topic_publisher.cpp

```
#include <ros/ros.h>
#include <std_msgs/Int32.h>  ❶


int main(int argc, char **argv) {
  ros::init(argc, argv, "count_publisher");
  ros::NodeHandle node;

  ros::Publisher pub = node.advertise<std_msgs::Int32>("counter", 10); ❷

  ros::Rate rate(1);  ❸
  int count = 0;

  while (ros::ok()) {  ❹
    std_msgs::Int32 msg; ❺
    msg.data = count;

    pub.publish(msg);  ❻

    ++count;
    rate.sleep();  ❼
  }

  return 0;  ❽
}
```

❶ Include the definition of the message we're going to use.

❷ Create the publisher.

❸ Create a `Rate` instance to control the publishing rate.

❹ Loop while the node is alive.

❺ Create a message and populate its data field.

❻ Publish the message.

❼ Wait for a while.

❽ Return success.

The two notable parts of this code are the creation of the topic publisher, and the loop condition. To create a publisher, we use the syntax:

```
ros::Publisher pub = node.advertise<std_msgs::Int32>("counter", 10);
```

This is a function defined as part of the `NodeHandle` class, templated on the type of message that's being sent. The parameters are the topic name, and the buffer size. The loop condition:

```
while (ros::ok()) {
```

will evaluate to true as long as the node is running and has not received a Ctrl-C to shut it down.

The corresponding topic subscriber node is shown in Example 23-4, and is even simpler.

Example 23-4. topic_subscriber.cpp

```
#include <ros/ros.h>
#include <std_msgs/Int32.h>

#include <iostream>


void callback(const std_msgs::Int32::ConstPtr &msg) {  ❶
  std::cout << msg->data << std::endl;
}


int main(int argc, char **argv) {
  ros::init(argc, argv, "count_subscriber");
  ros::NodeHandle node;

  ros::Subscriber sub = node.subscribe("counter", 10, callback);  ❷

  ros::spin();
}
```

❶ Define the callback function.

❷ Create the subscriber.

As with the publisher, the subscriber is called on the node instance, but this time we don't need a template argument since it can be calculated from the type of the callback parameter. The three arguments are the topic name, the buffer size, and the callback function.

The trickiest part is the callback function:

```
void callback(const std_msgs::Int32::ConstPtr &msg) {
```

This function should have a return type of `void`, and a single argument that is a `const` reference to a `const` pointer to the message type. In this instance, the message type is `std_msgs::Int32`, and this has a type of `ConstPtr` defined within it. In general, the argument for a callback dealing with messages of type *`T`* should have an argument of type `const` *`T`*`::ConstPtr &`. When building the message definition, `catkin` will make sure that the type `ConstPtr` is defined for your message types. Note that `ConstPtr` is a reference-counted smart pointer. You're not expected to call `delete()` on this when you're done with the message.

Although we've used one particular signature for the callback here (using `ConstPtr`), there are actually several that will work just as well (they all resolve to the same underlying types). We suggest that you use signatures like this in your code, but don't be surprised if you see a different, but equivalent, signature in someone else's code.

Note that, when accessing the data from the message, you should use the dereferencing operator `->`:

```
std::cout << msg->data << std::endl;
```

As you can see, the basic structure and idioms of a C++ node are the same as those of a Python node, even if the syntax is a little different. This is also true for services and actions.

Services

Defining and using services is largely the same as defining and using topics. Example 23-5 shows how to define the word counting service from Chapter 4 in C++.

Example 23-5. service_server.cpp

```
#include <ros/ros.h>
#include <cpp/WordCount.h>
```

```
bool count(cpp::WordCount::Request &req,  ❶
           cpp::WordCount::Response &res) {
  l = strlen(req.words);
  if (l == 0)
    count = 0;
  else {
    count = 1;
    for(int i = 0; i < l; ++i)
      if (req.words[i] == ' ')
        ++count;
  }

  res.count = count;

  return true;
}


int main(int argc, char **argv) {
  ros::init(int argc, char **argv, "count_server");
  ros::NodeHandle node;

  ros::ServiceServer service = node.advertiseService("count", count);  ❷

  ros::spin();  ❸


  return 0;
}
```

❶ Define the callback function.

❷ Create the server.

❸ Give control over to ROS.

The main differences here are that the callback function takes two arguments: the request, of type `WordCount::Request`, and a response, of type `WordCount::Response`. Again, these are provided automatically when you build the service definition. The return value is placed in the response argument, and the callback returns true or false, indicating success or failure. Once again, we advertise it through the node handle.

Example 23-6 shows how to use the service.

Example 23-6. service_client.cpp

```
#include <ros/ros.h>
#include <cpp/WordCount.h>
```

```
#include <iostream>


int main(int argc, char **argv) {
  ros::init(argc, char **argv, "count_client");
  ros::NodeHandle node;

  ros::ServiceClient client = node.serviceClient<cpp::WordCount>("count");  ❶

  cpp::WordCount srv;  ❷
  srv.request.words = "one two three four";

  if (client.call(srv))  ❸
    std::cerr << "success: " << srv.response.count << std::endl;  ❹
  else
    std::cerr << "failure" << std::endl;

  return 0;
}
```

❶ Create the service client.

❷ Create a data structure for the request and response.

❸ Call the service, testing for success.

❹ Access the response through the data structure.

Again, we make a call on the node handle, templated on the service data type, to set up the client. We then create an instance of the service data type, and fill in the request information. The actual service call is made using the `client.call(srv)` call, which will return true if successful, and false otherwise. Note that it is the responsibility of the service server to return this value. Finally, we can access the results of the call through the data structure's `response` field.

Summary

In this final chapter, we've seen how to translate some of the Python code from the rest of the book into C++. All of the idioms and design patterns that we've talked about previously are the same, regardless of the language that you write your code in; only the syntax and details change. Once you learn how to make these cosmetic changes, you should be able to switch from Python to C++ and back again with ease.

Of course, we've only scratched the surface of the C++ API in this chapter. Dealing with it completely would take a whole other book. However, if you're already familiar with the language, then you should be able to take this chapter in one hand and the

ROS wiki documentation in the other, and start crafting your own ROS nodes in C++. Or, you can choose a simpler life, and stick with Python. Your choice.