# A Gentle Introduction to ROS

Chapter: Services

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 8

# Services

*In which we call services and respond to service requests.*

In Chapters 2 and 3, we focused on how messages travel between nodes. Even though they are the primary method for communication in ROS, messages do have some limitations. This chapter introduces an alternative method of communication called **service calls**. Service calls differ from messages in two ways.

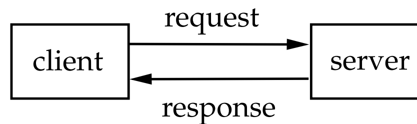
- ☞ Service calls are **bi-directional**. One node sends information to another node and waits for a response. Information flows in both directions. In contrast, when a message is published, there is no concept of a response, and not even any guarantee that anyone is subscribing to those messages.

- ☞ Service calls implement **one-to-one** communication. Each service call is initiated by one node, and the response goes back to that same node. On the other hand, each message is associated with a topic that might have many publishers and many subscribers.

Aside from these (very important!) differences, services are similar to messages. In this chapter, we'll see how to inspect and call services from the command line, and how to write nodes that act as either service clients or as servers.

## 8.1 Terminology for services

Here is the basic flow of information for service calls:




The idea is that a **client** node sends some data called a **request** to a **server** node and waits for a reply. The server, having received this request, takes some action (computing something, configuring hardware or software, changing its own behavior, *etc.*) and sends some data called a **response** back to the client.

The specific content of the request and response data is determined by the **service data type**, which is analogous to the message types that determine the content of messages (recall Section 2.7.2). Like a message type, a service data type is defined by a collection of named fields. The only difference is that a service data type is divided into two parts, representing the request (which is supplied by the client to the server) and the response (which is sent by the server back to the client).

## 8.2 Finding and calling services from the command line

Although services are most commonly used by code within nodes, there do exist a few command line tools for interacting with them. Experimenting with these tools can make it easier to understand how service calls work.

**Listing all services** You can get a list of services that are currently active using this command:[1]

```
rosservice list
```

On the author's computer, with just a turtlesim node running, the list of services looks like this:

```
/clear
/kill
/reset
/rosout/get_loggers
/rosout/set_logger_level
/spawn
/turtle1/set_pen
/turtle1/teleport_absolute
```

[1] http://wiki.ros.org/rosservice

```
/turtle1/teleport_relative
/turtlesim/get_loggers
/turtlesim/set_logger_level
```

Each line shows the name of one service that is currently available to call. Service names are graph resource names, and like other graph resource names can be specified as global, relative, or private names. The output of `rosservice list` shows the full global name of each service.

The services in this example, and many ROS services in general, can be divided into two basic types.

☞ Some services, such as the `get_loggers` and `set_logger_level` services in the list above, are used to get information from or pass information to specific nodes. These kinds of services usually use their node's name as a namespace to prevent name collisions, and to allow their nodes to offer them via private names like `~get_loggers` or `~set_logger_level`. (See Section 4.5 for details about loggers and logger levels.)

☞ Other services represent more general capabilities that are not conceptually tied to any particular node. For example, the service called `/spawn`, which creates a new simulated turtle, is offered by the `turtlesim` node. However, in a different system, this service could conceivably be offered by a different node; when we call `/spawn`, we only care that a new turtle is created, not about the details of which node does that work. All of the services in the list above, except the `get_loggers` and `set_logger_level` services, fit this description. These kinds of services typically have names that describe their function, but that do not mention any specific node.

**Listing services by node** To see the services offered by one particular node, use the `rosnode info` command:

```
rosnode info node-name
```

For example, here's the relevant portion of the output of this command for a `turtlesim` node:

```
Services:
* /turtle1/teleport_absolute
* /turtlesim/get_loggers
* /turtlesim/set_logger_level
* /reset
* /spawn
* /clear
```

```
* /turtle1/set_pen
* /turtle1/teleport_relative
* /kill
```

This shows, hopefully unsurprisingly, that most of the services currently available are offered by the turtlesim node. (The only exceptions are the two logging services offered by /rosout.)

**Finding the node offering a service** To perform the reverse query—that is, to see which node offers a given service—use this command:

```
rosservice node service-name
```

As expected, this command outputs /turtlesim when asked about any of the services listed by rosnode info /turtlesim, and /rosout when asked about the other two services.

**Finding the data type of a service** You can determine the service data type of a service using a command like this:

```
rosservice info service-name
```

For example, from the command

```
rosservice info /spawn
```

the output is:

```
Node: /turtlesim
URI: rosrpc://donatello:47441
Type: turtlesim/Spawn
Args: x y theta name
```

We can see that the data type of the /spawn service is turtlesim/Spawn. As with message types, a service data type has two parts, one naming the package that owns the type, and one naming the type itself:

$$\underbrace{\texttt{turtlesim}}_{\text{package name}} + \underbrace{\texttt{Spawn}}_{\text{type name}} \Rightarrow \underbrace{\texttt{turtlesim/Spawn}}_{\text{service data type}}$$

Service data types are always referenced by these kinds of complete names.

**Inspecting service data types** We can get some details about service data types using the rossrv command:

```
rossrv show service-data-type-name
```

For example,

```
rossrv show turtlesim/Spawn
```

produces this output:

```
float32 x
float32 y
float32 theta
string name
---
string name
```

In this case, the data before the dashes (---) are the elements of the **request**. This is the information that the client node sends to the server node. Everything after the dashes is the **response**, or information that the server sends back from the client when the server has finished acting on the request.

*Be careful about the difference between* rosservice *and* rossrv*. The former is for interacting with services that are currently offered by some node. The latter—whose name comes from the* .srv *extension used for files that declare service data types—is for asking about service data types, whether or not any currently available service has that type. The difference is similar to the difference between the* rostopic *and* rosmsg *commands:*

| | ***Topics*** | ***Services*** |
|---|---|---|
| ***active things*** | rostopic | rosservice |
| ***data types*** | rosmsg | rossrv |

Note that the request, the response, or both can be empty. For example, in the /reset service offered by turtlesim_node, which has type std_srvs/Empty, both the request and response parts are empty. This is roughly equivalent to a C++ function that accepts no arguments and returns void. No information goes in or out, but useful things (that is, side effects) still may happen.

**Calling services from the command line** To get a feel for how services work, you can call them from the command line using this command:

```
rosservice call service-name request-content
```

The request content part should list values for each field of the request, as shown by `rossrv show`. Here's an example:

```
rosservice call /spawn 3 3 0 Mikey
```

The effect of this service call is to create a new turtle named "Mikey," at position $(x, y) = (3, 3)$, facing angle $\theta = 0$, within the existing simulator.

*This new turtle comes with its own set of resources, including* `cmd_vel`, `pose`, *and* `color_sensor` *topics and* `set_pen`, `teleport_absolute`, `teleport_relative` *services. These new resources live in a namespace called—in this example—*`Mikey`. *These are in addition to the usual resources in the* `turtle1` *namespace, and are needed to allow other nodes to control the separate turtles individually. This nicely illustrates the way that namespaces can prevent name collisions.*

The output from `rosservice call` shows the server's response data. For the example above, the response should be:

```
name: Mikey
```

In this case, the server sends the new turtle's name back as part of the response.

In addition to sending the response data, the server also tells the client whether the call has succeed or failed. For example, in `turtlesim`, each turtle must have a unique name. If we run the `rosservice call` example above twice, the first call should succeed, but the second will generate an error that looks like this:

```
ERROR: service [/spawn] responded with an error:
```

The error occurs because we've attempted to create two turtles with the same name.

▶▶| *This error message ends with a colon because* `turtlesim` *has replied with an empty error message. The underlying infrastructure is able to return short error message strings when service calls fail, but the C++ client library, which* `turtlesim` *is using, does not provide an easy way to return a non-empty error message.*

## 8.3 A client program

Calling services from the command line is handy for exploring and for things that only need to be done occasionally, but of course it's much more useful to be able to call services from your code.[2] Listing 8.1 shows a short example of how to do that. That example illustrates all of the basic elements of a service client program.

**Declaring the request and response types** Just like message types (recall Section 3.3.1), every service data has an associated C++ header file that we must include:

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

In the example, we say

```
#include <turtlesim/Spawn.h>
```

to include the definition of a class called `turtlesim::Spawn`, which defines the data type—including both the request and response parts—of the service we want to call.

**Creating a client object** After initializing itself as a node (by calling `ros::init` and creating a `NodeHandle` object), our program must create an object of type `ros::ServiceClient`, whose job is to actually carry out the service call. The declaration of a `ros::ServiceClient` looks like this:

```
ros::ServiceClient client = node_handle.serviceClient<service_type>(
    service_name);
```

This line has three important parts.

- ☞ The *node_handle* is the usual `ros::NodeHandle` object. We're calling its `serviceClient` method.
- ☞ The *service_type* is the name of the service object defined in the header file we included above. In the example, it's `turtlesim::Spawn`.
- ☞ The *service_name* is a string naming the service that we want to call. This should be a relative name, but can also be a global name. The example uses the relative name `"spawn"`.

By default, creating this object is relatively inexpensive because it doesn't do much, except to store the details about the service we'll want to call later.

---

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

```
1  // This program spawns a new turtlesim turtle by calling
2  // the appropriate service.
3  #include <ros/ros.h>
4
5  // The srv class for the service.
6  #include <turtlesim/Spawn.h>
7
8  int main(int argc, char **argv) {
9    ros::init(argc, argv, "spawn_turtle");
10   ros::NodeHandle nh;
11
12   // Create a client object for the spawn service.  This
13   // needs to know the data type of the service and its
14   // name.
15   ros::ServiceClient spawnClient
16     = nh.serviceClient<turtlesim::Spawn>("spawn");
17
18   // Create the request and response objects.
19   turtlesim::Spawn::Request req;
20   turtlesim::Spawn::Response resp;
21
22   // Fill in the request data members.
23   req.x = 2;
24   req.y = 3;
25   req.theta = M_PI / 2;
26   req.name = "Leo";
27
28   // Actually call the service.  This won't return until
29   // the service is complete.
30   bool success = spawnClient.call(req, resp);
31
32   // Check for success and use the response.
33   if(success) {
34     ROS_INFO_STREAM("Spawned a turtle named "
35       << resp.name);
36   } else {
37     ROS_ERROR_STREAM("Failed to spawn.");
38   }
39
40 }
```

Listing 8.1: A program called spawn_turtle.cpp that calls a service.

*Notice that creating a* ros::ServiceClient *does not require a queue size, in contrast to the analogous* ros::Publisher*. This difference occurs because service calls do not return until the response arrives. Because the client waits for the service call to complete, there is no need to maintain a queue of outgoing service calls.*

**Creating request and response objects** Once the ros::ServiceClient has been constructed, the next step is to create a request object to contain the data to be sent to the server. The header we included above includes separate classes for the response and request parts of the service data type, named Request and Response, respectively. These classes must be referenced via the package name and service type, like this:

*package_name*::*service_type*::Request
*package_name*::*service_type*::Response

Each of these classes has data members matching the fields of the service type. (Recall that rossrv show can list those fields and their data types for us.) These fields are mapped to C++ data types in the same way that messages fields are. The Request constructor supplies meaningless default values for those fields, so we should assign a value to each field. In the example, we create a turtlesim::Spawn::Request object and assign values to its x, y, theta, and name fields.

We'll also need a Response object—in the example, a turtlesim::Spawn::Response—but, since that information should come from the server, we should not attempt to fill in its data members.

*Service type header files also define a single class (a* struct *really) named*

*package_name*::*service_type*

*that contains both a* Request *and a* Response *as data members. An object from this class is usually called a* srv*. If you prefer—as the authors of many online tutorials apparently do—you can pass an object of this class to the* call *method introduced below, instead of separate* Request *and* Response *objects.*

**Calling the service** Once we have a ServiceClient, a completed Request, and a Response, we can actually call the service:

```
bool success = service_client.call(request, response);
```

This method does the actual work of locating the server node, transmitting the request data, waiting for a response, and storing the response data the Response we provided.

The call method returns a boolean value that tells us if the service call completed successfully. Failures can occur because of problems with the ROS infrastructure—for example, attempting to call a service not offered by any node—or for reasons specific to an individual service. In the example, a failed call most commonly indicates that another turtle already exists with the requested name.

*A common mistake is to fail to check the return value of* call. *This can lead to unexpected problems if the service call fails. It takes only a minute or two to add code to check this value and call* ROS_ERROR_STREAM *when the service call fails. It's quite likely that this investment of time will be repaid with easier debugging in the future.*

⏭ *By default, the process of finding and connecting to the server node occurs inside the* call *method. This connection is used for that service call and then closed before* call *returns. ROS also supports a concept of* persistent *service clients, in which the* ros::ServiceClient *constructor establishes a connection to the server, which is then reused for every subsequent* call *for that client object. A persistent service client can be created by passing* true *for the second parameter of the constructor (which we've allowed to default to* false *in the previous examples):*

```
ros::ServiceClient client = node_handle.advertise<service_type>(
    service_name, true);
```

*The use of persistent clients is mildly discouraged by the documentation,*[3] *because the performance gains tend to be rather small—The author's informal experiments showed an improvement of only about 10%—and the resulting system can be less robust to restarts or changes of the server node.*

[3] http://www.ros.org/doc/api/roscpp/html/classros_1_1NodeHandle.html

After the service call successfully completes, you access the response data from the Request object that you passed to call. In the example, the response includes only an echo of the name field from the request.

**Declaring a dependency** That's all there is to the client code. However, to get catkin_make to correctly compile a client program, we must be sure that the program's package declares a dependency on the package that owns the service type. Such dependencies, which are the same as those we needed for message types (recall Section 3.3.3), require edits to CMakeLists.txt and to the manifest, package.xml. To compile the example program, we must ensure that the find_package line in CMakeLists.txt mentions the turtlesim package:

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

In package.xml, we should ensure that build_depend and run_depend elements exist that name the package:

```
<build_depend>turtlesim</build_depend>
<run_depend>turtlesim</run_depend>
```

After completing these changes, the usual catkin_make should compile the program.

## 8.4 A server program

Now let's take a look at the other side of service calls, by writing a program that acts as a server. Listing 8.2 shows an example that offers a service called toggle_forward and also drives a turtlesim robot, alternating between forward motions and rotations each time that service is called.

The code for acting as a server is remarkably similar to the code for subscribing to a topic. Aside from differences in names—we must create a ros::ServiceServer instead of a ros::Subscriber—the only difference is that a server can send data back to the client, via both a response object and a boolean indication of success or failure.

**Writing a service callback** Just like with subscriptions, each service that our nodes offer must be associated with a callback function. A service callback looks like this:

```
bool function_name(
  package_name::service_type::Request &req,
  package_name::service_type::Response &resp
) {
```

```
1  // This program toggles between rotation and translation
2  // commands, based on calls to a service.
3  #include <ros/ros.h>
4  #include <std_srvs/Empty.h>
5  #include <geometry_msgs/Twist.h>
6
7  bool forward = true;
8  bool toggleForward(
9    std_srvs::Empty::Request &req,
10   std_srvs::Empty::Response &resp
11 ) {
12   forward = !forward;
13   ROS_INFO_STREAM("Now␣sending␣" << (forward ?
14     "forward" : "rotate") << "␣commands.");
15   return true;
16 }
17
18 int main(int argc, char **argv) {
19   ros::init(argc, argv, "pubvel_toggle");
20   ros::NodeHandle nh;
21
22   // Register our service with the master.
23   ros::ServiceServer server = nh.advertiseService(
24     "toggle_forward", &toggleForward);
25
26   // Publish commands, using the latest value for forward,
27   // until the node shuts down.
28   ros::Publisher pub = nh.advertise<geometry_msgs::Twist>(
29     "turtle1/cmd_vel", 1000);
30   ros::Rate rate(2);
31   while(ros::ok()) {
32     geometry_msgs::Twist msg;
33     msg.linear.x = forward ? 1.0 : 0.0;
34     msg.angular.z = forward ? 0.0 : 1.0;
35     pub.publish(msg);
36     ros::spinOnce();
37     rate.sleep();
38   }
39 }
```

Listing 8.2: A program called pubvel_toggle.cpp that changes the velocity commands it publishes, based on a service that it offers.

```
    ...
}
```

ROS executes the callback function once for each service call that our node receives. The Request parameter contains the data sent from the client. The callback's job is to fill in the data members of the Response object. These are the same Request and Response types that we used on the client side above, and as such, they require the same header and the same package dependencies to compile properly. The callback function should return true to indicate success or false to indicate failure.

In the example, we use the std_srvs/Empty message type, in which both the Request and Response sides are empty, so there is no processing to perform for either of those objects. The callback's only work is to toggle a global boolean variable, called forward, that governs the velocity messages published in main.

**Creating a server object** To associate the callback function with a service name, and to offer the service to other nodes, we must advertise the service:

```
ros::ServiceServer server = node_handle.advertiseService(
    service_name,
    pointer_to_callback_function
);
```

All of these elements have appeared before.

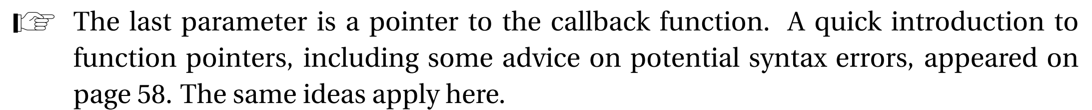
- ☞ The *node_handle* is the same old node handle that we know and love.
- ☞ The *service_name* is a the string name of the service we would like to offer. This should be a relative name, but could also be a global name.

> ▶▶| *Because of some perceived ambiguity in how private names should be resolved,* ros::NodeHandle::advertiseService *refuses to accept private names (that is, those that begin with ~).The solution to this constraint is to exploit the fact—one we have not used so far—that we can create* ros::NodeHandle *objects with their own specific default namespaces. For example, we could create a* ros::NodeHandle *like this:*
>
> ```
> ros::NodeHandle nhPrivate("~");
> ```
>
> *The default namespace for any relative names we send to this* NodeHandle *would then be the same as the node's name. In particular, this means that if*

*we use this handle and a relative name to advertise a service, it would have the same effect as using a private name. For example, in a node named* `/foo/bar`*, we can advertise a service called* `/foo/bar/baz` *like this:*

```
ros::ServiceServer server = nhPrivate.advertiseService(
    "baz",
    callback
);
```

*That is, this code has the same effect we might expect from attempting to advertise a service called* `~baz` *using our usual* `NodeHandle`*, if that handle were willing to accept private names.*



☞ The last parameter is a pointer to the callback function. A quick introduction to function pointers, including some advice on potential syntax errors, appeared on page 58. The same ideas apply here.

As with `ros::Subscriber` objects, it is rare to call any methods of `ros::ServiceServer` objects. Instead, we should keep careful track of the lifetime of that object, because the service will be available to other nodes only until the `ros::ServiceServer` is destroyed.

**Giving ROS control** Don't forget that ROS will not execute any callback functions until we specifically ask it to, using `ros::spin()` or `ros::spinOnce()`. (Details about the differences between these two functions appear, in the context of a subscriber program, near the end of Section 3.4.)

In the example, we use `ros::spinOnce()`, instead of `ros::spin()`, because we have other work to do—specifically, publishing velocity commands—when there are no incoming service calls to process.

### 8.4.1 Running and improving the server program

To test the `pubvel_toggle` example program, compile it and run both `turtlesim_node` and `pubvel_toggle`. With both running, you can switch the motion commands from translation to rotation and back by calling the `toggle_forward` service from the command line:

```
rosservice call /toggle_forward
```

Figure 8.1 shows an example of the results.

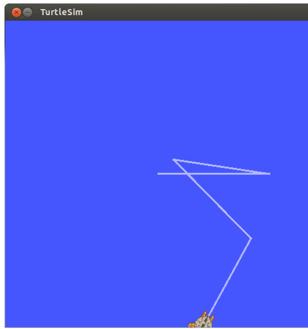



Figure 8.1: Results from running `pubvel_toggle` with some intermittent, manual calls to `/toggle_forward`.

One potentially unexpected "feature" of this program is that there can be a noticeable lag between starting the `rosservice call` command and observing an actual change in the turtle's motion. A very small part of this delay can be attributed to time needed for communication between `rosservice call`, `pubvel_toggle`, and `turtlesim_node`. However, most of the delay comes from the architecture of `pubvel_toggle`. Can you see where?

The answer is that, because we use the `sleep` method of a a `ros::Rate` object with a relatively slow frequency (only 2Hz), *this program spends most of its time asleep*. Most service calls will arrive when the `sleep` is executing, and these service calls cannot execute until the call to `ros::spinOnce()`, which happens only every 0.5 seconds. Therefore, there can be a delay of up to about half a second before each service call can be handled.

There are at least two ways to work around this kind of problem:

- ☞ We can use two separate threads: one to publish messages, and one to handle service callbacks. Although ROS doesn't require programs to use threads explicitly, it is quite cooperative if they do.
- ☞ We can replace the `sleep`/`ros::spinOnce` loop with a `ros::spin`, and use a **timer callback**[4] to the publish messages.

Issues like this can seem minor at this scale—A small delay in changing the turtle's movement pattern may not be a major problem—but for programs for which the timing is more important, the difference can be crucial.

## 8.5 Looking ahead

This chapter covered services, which have both strong similarities and vital differences from messages. In the next chapter we'll change gears, and learn about a tool called ros-

[4] `http://wiki.ros.org/roscpp/Overview/Timers`

`bag`, which enables rapid, repeatable experimentation by recording and playing back messages.