Chapter 4. Services

Services are another way to pass data between nodes in ROS. Services are just synchronous remote procedure calls; they allow one node to call a function that executes in another node. We define the inputs and outputs of this function similarly to the way we define new message types. The server (which provides the service) specifies a callback to deal with the service request, and advertises the service. The client (which calls the service) then accesses this service through a local proxy.

Service calls are well suited to things that you only need to do occasionally and that take a bounded amount of time to complete. Common computations, which you might want to distribute to other computers, are a good example. Discrete actions that the robot might do, such as turning on a sensor or taking a high-resolution picture with a camera, are also good candidates for a service-call implementation.

Although there are several services already defined by packages in ROS, we'll start by looking at how to define and implement our own service, since this gives some insight into the underlying mechanisms of service calls. As a concrete example in this chapter, we're going to show how to create a service that counts the number of words in a string.

Defining a Service

The first step in creating a new service is to define the service call inputs and outputs. This is done in a *service-definition file*, which has a similar structure to the message-definition files we've already seen. However, since a service call has both inputs and outputs, it's a bit more complicated than a message.

Our example service counts the number of words in a string. This means that the input to the service call should be a `string` and the output should be an integer. Although we're

using messages from `std_msgs` here, you can use *any* ROS message, even ones that you've


```
uint32 count
```

NOTE

Like message-definition files, service-definition files are just lists of message types. These can be built in, such as those defined in the `std_msgs` package, or they can be ones you have defined yourself.

The inputs to the service call come first. In this case, we're just going to use the ROS built-in `string` type. Three dashes (`---`) mark the end of the inputs and the start of the output definition. We're going to use a 32-bit unsigned integer (`uint32`) for our output. The file holding this definition is called *WordCount.srv* and is traditionally in a directory called *srv* in the main package directory (although this is not strictly required).

Once we've got the definition file in the right place, we need to run `catkin_make` to create the code and class definitions that we will actually use when interacting with the service, just like we did for new messages. To get `catkin_make` to generate this code, we need to

`message_generation`, just like we did for new messages:

```
find_package(catkin REQUIRED COMPONENTS
   roscpp
   rospy
   message_generation   # Add message_generation here, after the other packages
)
```

We also have to make an addition to the *package.xml* file to reflect the dependencies on both `rospy` and the message system. This means we need a build dependency on `message_generation` and a runtime dependency on `message_runtime`:

```
<build_depend>rospy</build_depend>
<run_depend>rospy</run_depend>
```

```
add_service_files(
  FILES
  WordCount.srv
)
```

Finally, we must make sure that the dependencies for the service-definition file are declared (again in *CMakeLists.txt*), using the `generate_messages()` call:

```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

With all of this in place, running `catkin_make` will generate three classes: `WordCount`, `WordCountRequest`, and `WordCountResponse`. These classes will be used to interact with the service, as we will see. Just like with messages, you will probably never have to look at the details of the generated classes. However, just in case you're interested, (part of) the classes generated by the `WordCount` example are shown in Example 4-2.

Example 4-2. The Python classes generated by catkin_make for the WordCount example (code in functions removed for clarity)

```
"""autogenerated by genpy from basics/WordCountRequest.msg. Do not edit."""
import sys
python3 = True if sys.hexversion > 0x03000000 else False
import genpy
import struct


class WordCountRequest(genpy.Message):
  _md5sum = "6f897d3845272d18053a750c1cfb862a"
  _type = "basics/WordCountRequest"
  _has_header = False #flag to mark the presence of a Header object
  _full_text = """string words

"""
  __slots__ = ['words']
```

```
  _slot_types = ['string']
```



```
    changes.  You cannot mix in-order arguments and keyword arguments.

    The available fields are:
       words

    :param args: complete set of field values, in .msg order
    :param kwds: use keyword arguments corresponding to message field names
    to set specific fields.
    """
    if args or kwds:
      super(WordCountRequest, self).__init__(*args, **kwds)
      #message fields cannot be None, assign default values for those that are
      if self.words is None:
        self.words = ''
    else:
      self.words = ''

  def _get_types(self):
    ...     """

  def serialize(self, buff):
    ...

  def deserialize(self, str):
    ...

  def serialize_numpy(self, buff, numpy):
    ...

  def deserialize_numpy(self, str, numpy):
    ...

class WordCountResponse(genpy.Message):
  ...

class WordCount(genpy.Message):
  ...
```

The details of the definitions for `WordCountResponse` and `WordCount` are similar to those for `WordCountRequest`. All of these are just ROS messages.

We can verify that the service call definition is what we expect by using the `rossrv`


```
---
uint32 count
```

You can see all available services using `rossrv list`, all packages offering services with `rossrv packages`, and all the services offered by a particular package with `rossrv package`.

Implementing a Service

Now that we have a definition of the inputs and outputs for the service call, we're ready to write the code that implements the service. Like topics, services are a callback-based mechanism. The service provider specifies a callback that will be run when the service call is made, and then waits for requests to come in. Example 4-3 shows a simple server that implements our word-counting service call.

Example 4-3. service_server.py

```
#!/usr/bin/env python

import rospy

from basics.srv import WordCount,WordCountResponse


def count_words(request):
    return WordCountResponse(len(request.words.split()))


rospy.init_node('service_server')

service = rospy.Service('word_count', WordCount, count_words)

rospy.spin()
```

We first need to import the code generated by `catkin`:

```
from basics.srv import WordCount,WordCountResponse
```

Notice that we need to import both `WordCount` and `WordCountResponse`. Both of these


```
def count_words(request):
    return WordCountResponse(len(request.words.split()))
```

The constructor for `WordCountResponse` takes parameters that match those in the service-definition file. For us, this means an unsigned integer. By convention, services that fail, for whatever reason, should return `None`.

After initializing the node, we advertise the service, giving it a name (`word_count`) and a type (`WordCount`), and specifying the callback that will implement it:

```
service = rospy.Service('word_count', WordCount, count_words)
```

Finally, we make a call to `rospy.spin()`, which gives control of the node over to ROS and exits when the node is ready to shut down. You don't actually have to hand control over by calling `rospy.spin()` (unlike in the C++ API), since callbacks run in their own threads. You could set up your own loop, remembering to check for node termination, if you have something else you need to do. However, using `rospy.spin()` is a convenient way to keep the node alive until it's ready to shut down.

Checking That Everything Works as Expected

Now that we have the service defined and implemented, we can verify that everything is working as expected with the `rosservice` command. Start up a `roscore` and run the service node:

```
user@hostname$ rosrun basics service_server.py
```

First, let's check that the service is there:

```
user@hostname$ rosservice list
/rosout/get_loggers
/rosout/set_logger_level
/service_server/get_loggers
```

```
/service_server/set_logger_level
```



```
user@hostname$ rosservice info word_count
Node: /service_server
URI: rosrpc://hostname:60085
Type: basics/WordCount
Args: words
```

This tells us the node that provides the service, where it's running, the type that it uses, and the names of the arguments to the service call. We can also get some of this information using `rosservice type word_count` and `roservice args word_count`.

Other Ways of Returning Values from a Service

In the previous example, we explicitly created a `WordCountResponse` object and returned it from the service callback. There are a number of other ways to return values from a service callback that you can use. In the case where there is a single return argument for the service, you can simply return that value:

```
def count_words(request):
    return len(request.words.split())
```

If there are multiple return arguments, you can return a tuple or a list. The values in the list will be assigned to the values in the service definition, in order. This works even if there's only one return value:

```
def count_words(request):
    return [len(request.words.split())]
```

You can also return a dictionary, where the keys are the argument names (given as strings):

```
def count_words(request):
    return {'count': len(request.words.split())}
```

In both of these cases, the underlying service call code in ROS will translate these return

The simplest way to use a service is to call it using the `rosservice` command. For our word-counting service, the call looks like this:

```
user@hostname$ rosservice call word_count 'one two three'
count: 3
```

The command takes the `call` subcommand, the service name, and the arguments. While this lets us call the service and make sure that it's working as expected, it's not as useful as calling it from another running node. Example 4-4 shows how to call our service programmatically.

Example 4-4. service_client.py

```
#!/usr/bin/env python

import rospy

from basics.srv import WordCount

import sys


rospy.init_node('service_client')

rospy.wait_for_service('word_count')

word_counter = rospy.ServiceProxy('word_count', WordCount)

words = ' '.join(sys.argv[1:])

word_count = word_counter(words)

print words, '->', word_count.count
```

First, we wait for the service to be advertised by the server:

```
rospy.wait_for_service('word_count')
```

If we try to use the service before it's advertised, the call will fail with an exception. This is


```
word_counter = rospy.ServiceProxy('word_count', WordCount)
```

We need to specify the name of the service (`word_count`) and the type (`WordCount`). This will allow us to use `word_counter` like a local function that, when called, will actually make the service call for us:

```
word_count = word_counter(words)
```

Checking That Everything Works as Expected

Now that we've defined the service, built the support code with `catkin`, and implemented both a server and a client, it's time to see if everything works. Check that your server is still running, and run the client node (make sure that you've sourced your workspace setup file in the shell in which you run the client node, or it will not work):

```
user@hostname$ rosrun basics service_client.py these are some words
these are some words -> 4
```

Now, stop the server and rerun the client node. It should stop, waiting for the service to be advertised. Starting the server node should result in the client completing normally, once the service is available. This highlights one of the limitations of ROS services: the service client can potentially wait forever if the service is not available for some reason. Perhaps the service server has died unexpectedly, or perhaps the service name is misspelled in the client call. In either case, the service client will get stuck.

Other Ways to Call Services

In our client node, we are calling the service through the proxy as if it were a local function. The arguments to this function are used to fill in the elements of the service request, in order. In our example, we only have one argument (`words`), so we are only allowed to give the proxy function one argument. Similarly, since there is only one output from the service call, the proxy function returns a single value. If, on the other hand, our service definition were to look like this:

```
string words
```


then the proxy function would take two arguments, and return two values:

```
c,i = word_count(words, 3)
```

The arguments are passed in the order they are defined in the service definition. It is also possible to explicitly construct a service request object and use that to call the service:

```
request = WordCountRequest('one two three', 3)
count,ignored = word_counter(request)
```

Note that, if you choose this mechanism, you will have to also import the definition for `WordCountRequest` in the client code, as follows:

```
from basics.srv import WordCountRequest
```

Finally, if you only want to set some of the arguments, you can use keyword arguments to make the service call:

```
count,ignored = word_counter(words='one two three')
```

While this mechanism can be useful, you should use it with care, since any arguments that you do not explicitly set will remain undefined. If you omit arguments that the service needs to run, you might get strange return values. You should probably steer clear of this calling style, unless you actually *need* to use it.

Summary

Now you know all about services, the second main communication mechanism in ROS. Services are really just synchronous remote procedure calls and allow explicit two-way communication between nodes. You should now be able to use services provided by other packages in ROS, and also to implement your own services.

Once again, we didn't cover all of the details of services. To get more information on more

highly variable, you should think about using an *action*, which we describe in the next chapter.