Module:3

ROS Programming (part_2)

Credit hours:5 Hours

Dr. Abhishek Rudra Pal
Assistant Professor (Senior Grade-1)
SMEC
Vellore Institute of Technology - Chennai Campus
Email: abhishek.rudrapal@vit.ac.in
Mobile No: +91-9043534478;+91-9051728314(Whatsapp)

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.

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 ,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 defined yourself..

Example 4-1. WordCount.srv

```
string words
 ---
uint32 count
```

- 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).

Changes in files

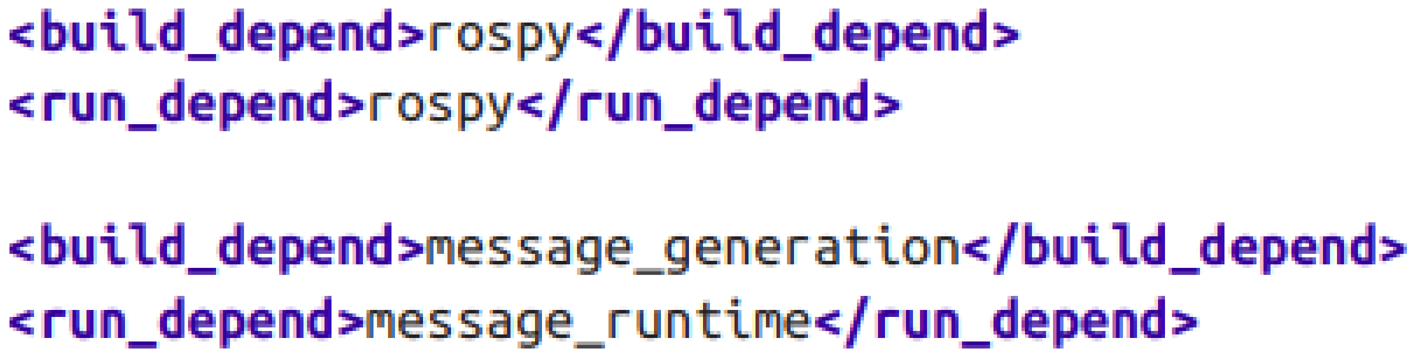
- Once we’ve got the definition file in the right place, we need to run catkin_make to create the code and class definitions that we will actually use when interacting with the service, just like we did for new messages.
- To get catkin_make to generate this code, we need to make sure that the find_package() call in CMakeLists.txt contains message_generation, just like we did for new messages.

Changes in files

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

Changes in files

- 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



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

<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
```

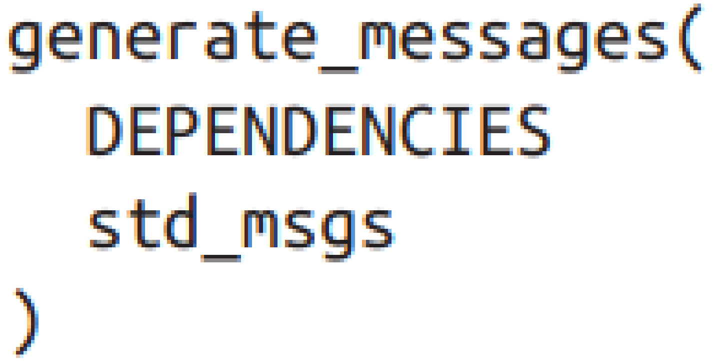
Changes in files

- Then, we need to tell catkin which service-definition files we want compiled, using the add_service_files() call in CMakeLists.txt:

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

Changes in files

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



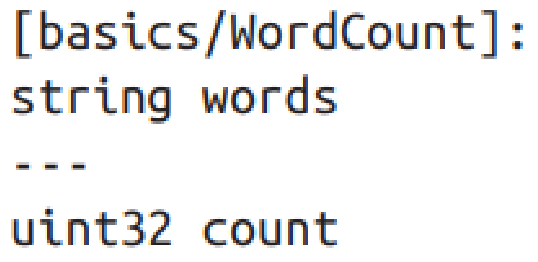
```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

Changes in files

- With all of this in place, running catkin_make will generate three classes:
- WordCount,
- WordCountRequest,
- WordCountResponse. These classes will be used to interact with the service, as we will see

verification

- rossrv show WordCount



```
[basics/WordCount]:
string words
---
uint32 count
```

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. 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()
```

explanation

- 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 are generated in a Python module with the same name as the package, with a .srv extension (basics.srv, in our case).
- The callback function takes a single argument of type WordCountRequest and returns a single argument of type WordCountResponse:
- def count_words(request): return WordCountResponse(len(request.words.split()))

explanation

- 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)

checking

- 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 /word_count
```

checking

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

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 val- ues 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())
```

Other Ways of Returning Values from a Service

- 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 types into a WordCountResponse object and return it to the calling node, just as in the initial example code.

Using a Service(client)

- 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. 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
```

explanation

- 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 a major difference between topics and services. We can subscribe to topics that are not yet advertised, but we can only use advertised services**. Once the service is advertised, we can set up a local proxy for it
- 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
  ```

Checking That Everything Works as Expected

- 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 Service

- 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:

Other Ways to Call Service

- string words
- int min_word_length
- ---
- uint32 count
- uint32 ignored
- Then the proxy function would take two arguments, and return two values:
- c,i = word_count(words, 3)

Other Ways to Call Service

- 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')