Lecture 08.02 Serving and calling a ROS service

server node Creating a *server node* is our first consideration.

08.02.1 Creating a server node

Here are some key aspects of a `rospy` server, listed below as instructions for creating such a node.

1. Import the service type and its `Response` function:
   `from <pkg>.srv import <srv_type> <srv_type>Response`.
2. Define a function to serve:
   `def fun(request)`.
3. Register a service:
   `rospy.Service(<srv_name>,<srv_type>,<fun>)`.
4. Wait for service requests: `rospy.spin()`.

The service function can return:

1. a `<srv_type>Response` object:
   `return <srv_type>Response(<value1>,<value2>,...)` or
2. a single value (matching a single service output type):
   `return <value>` or
3. a `list` of values (matching the output types):
   `return [<value1>,<value2>,...]` or
4. a `dict`ionary of values (matching the output names and types):
   `return {'name1':<value1>,'name2':<value2>}`.

08.02.2 An example server node

Let's implement our new service `word_count`, created in Lecture 08.01. We need a server node to do so. Create (`touch`) a Python node file `my_services/src/service_server.py`, change its permissions to user-executable (`chmod u+x`), and edit it to have the same contents as the `rico_services/src/service_server.py` file shown in Figure 08.1.

08.02.3 Creating a client node

client node The key elements to creating a *client node* are:

1. Import the service:
   `from <pkg>.srv import <srv_type>`.

```python
#!/usr/bin/env python
import rospy
from rico_services.srv import WordCount, WordCountResponse

def count_words(request):
  return len(request.words.split()) # num of words

rospy.init_node('service_server')

service = rospy.Service( # register service
  'word_count', # service name
  WordCount,    # service type
  count_words   # function service provides
)

rospy.spin()
```

Figure 08.1: `rico_services/src/service_server.py` listing.

2. Wait for a service:
   `rospy.wait_for_service('service_name')`.
3. Set up a proxy server for communication:
   `rospy.ServiceProxy(<srv_name>,<srv_type>)`.
4. Use the service: `fun(...)`.

08.02.4 An example client node

Let's create a client for our new service `word_count`. We need a client node to do so. Create (`touch`) a Python node file `my_services/src/service_client.py`, change its permissions to user-executable (`chmod u+x`), and edit it to have the same contents as the `rico_services/src/service_client.py` file shown in Figure 08.2.

The only thing that may surprise us here is the line `words = ' '.join(sys.argv[1:])`. The inner statement `sys.argv[1:]` returns a `list` of command-line arguments supplied to the node. Then `' '.join(...)` concatenates the (`string`) elements of the list with a space character between each pair. This is one of many ways we could *parse* command-line arguments.

argument parsing

```
#!/usr/bin/env python
import rospy
from rico_services.srv import WordCount
import sys

rospy.init_node('service_client')

rospy.wait_for_service('word_count') # wait for registration
word_counter = rospy.ServiceProxy( # set up proxy
  'word_count', # service name
  WordCount     # service type
)
words = ' '.join(sys.argv[1:]) # parse args
word_count = word_counter(words) # use service

print(words+'--> has '+str(word_count.count)+' words')
```

Figure 08.2: `rico_services/src/service_client.py` listing.

08.02.5 Running and verifying the server and client nodes

Navigate to your workspace root and build the workspace.

```
catkin_make
```

Run a `roscore`. In a new Terminal, in your workspace root, `source devel/setup.bash`, then run the server node.

```
rosrun my_services service_server.py
```

In a new Terminal, in your workspace root, `source devel/setup.bash`, then run the client node with command line arguments passed.

```
rosrun my_services service_client.py hello world sweet world
```

```
hello world sweet world--> has 4 words
```

It works!