Robotic Systems Engineering Lab 03

This document contains examples and exercises on writing code for customized ROS messages and ROS service-client communication.

A 'msg' file is a simple text file that describes the fields of a ROS message. Similarly, an 'srv' file describes a service. A service is composed of two parts, a request and a response.

Visit http://wiki.ros.org/ROS/Tutorials/CreatingMsgAndSrv#Introduction_to_msg_and_srv to take an extensive look on how 'msg' and 'srv' files are composed and what field types are used.

Message Example

In the previous lab we created our own package inside our Catkin Workspace called 'beginner_tutorials'. We will navigate into that package folder and we will create a ROS message.

```
cd ~/catkin_ws/src/beginner_tutorials
mkdir msg
echo int64 num > msg/Num.msg
```

Now navigate into the 'msg' folder of the 'beginner_tutorials' package and take a look at the message we just created. You can see that it is the simplest of messages, containing just a line with a file type. You can add more file types, always one per line.

Now that we have created our message, we need to turn it into source code for C++, Python etc.

In the 'package.xml' file of the package, add the following lines.

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

And in the 'CMakeLists.txt' file of the package, add the 'message_generation' line into your **existing** 'find_package' text field:

```
find_package(catkin REQUIRED COMPONENTS  roscpp  rospy  std_msgs  message_generation)
```

Also, in the same file add 'CATKIN_DEPENS message_runtime' in the **existing** 'catkin_package' text field:

```
catkin_package(CATKIN_DEPENDS message_runtime)
```

Next, add the following block of code into the file:

```
add_message_files(FILES  Num.msg)
```

This makes sure that CMake knows when it has to reconfigure the project after you add other '.msg' files and it has to be placed before the 'generate_messages' command. And lastly make sure the lines below are uncommented.

```
generate_messages(DEPENDENCIES  std_msgs)
```

To check if your message was created successfully, simply run the following command, after you have ran 'catkin_make'. You should see 'int64 num'.

```
rosmsg show beginner_tutorials/Num
```

Service Example

Navigate into that package folder and we will create a ROS service.

```
cd ~/catkin_ws/src/beginner_tutorials
mkdir srv
```

Instead of creating a new 'srv' definition by hand, we will copy an existing one from another package. For that, roscp is a useful commandline tool for copying files from one package to another. Now we can copy a service from the 'rospy_tutorials' package:

```
roscp rospy_tutorials AddTwoInts.srv srv/AddTwoInts.srv
```

Similarly with before, we need to turn the 'srv' file into source code. Both the 'package.xml' and 'CMakeLists.txt' files need to contain everything we added before in the 'Message Example'. Additionally, in the 'CMakeLists.txt' file we need to add before the 'generate_messages' command, this command:

```
add_service_files(FILES  AddTwoInts.srv)
```

To check if your message was created successfully simply run

```
rossrv show beginner_tutorials/AddTwoInts
```

And you should see:
'int64 a
int64 b
- - -
int64 sum'

Now that we made all these changes we should 'catkin_make' our environment once again.

To get a better understanding of how services are used in actual applications, you can either take a look at the examples we prepared, or for a far more extensive understanding, you can visit:

http://wiki.ros.org/roscpp/Overview/Services for C++,

http://wiki.ros.org/rospy/Overview/Services for Python.

Examples

Update your repository with the current lab folder from 'github', just like you did in previous labs.

In the 'example_msg_srv' folder we created a customized message of three floats, the coordinates (x,y,z) of a point. Inside the same folder we created two services. The first service requests three 'std_msgs' file types and responds with a 'geometry_msgs' file type while the second one both requests and responds with 'geometry_msgs' file types.

Example 01

Go through the code of the first example and try to understand what the script does and what kind of information the two nodes will exchange. What's the difference between this example and example01 from Lab02, in terms of information type?

To run this example, you need to have the master running and then in two separate terminals:

```
rosrun lab03example01 lab03example01_pub_node
```
, for C++

```
rosrun lab03example01 lab03_example01_pub.py
```
, for Python

and

```
rosrun lab03example01 lab03example01_sub_node
```
, for C++

```
rosrun lab03example01 lab03_example01_sub.py
```
, for Python

Alternatively, try and create a '.launch' file that simultaneously invokes two nodes without the need to open two separate terminals every time.

In this example we used ROS customised message in publisher/subscriber communication to convert the rotation representations from angle-axis to rotation matrix.

Example 02

Use the 'CMakeLists.txt' file and the previous example to run this example. Go through the code and try to understand how the service-request communication takes place. What's the difference between this example and example01 from Lab02, in terms of communication?

In this example we created ROS request/service communication to convert the incoming request angle/axis data to a response quaternion.

Tasks

Task 01

In this Task, you will create a point rotation-by-quaternion service.

1. Create a service node that rotates a point by the service response. When this service is requested, the requested point should be rotated in 3D space by the requested quaternion.
2. Create a request node that generates a random point and a random quaternion and uses the service to request the rotation of the 3D point. Calculate and print the value of the computational time between the request and the response and print the appropriate point response.

Task 02

In this task you're going to create three services that convert between three different rotation representations. The three rotation representations conversions are 'quaternion to *zyx'*, *'quaternion to angle/axis'* and *'rotation matrix to quaternion'.*