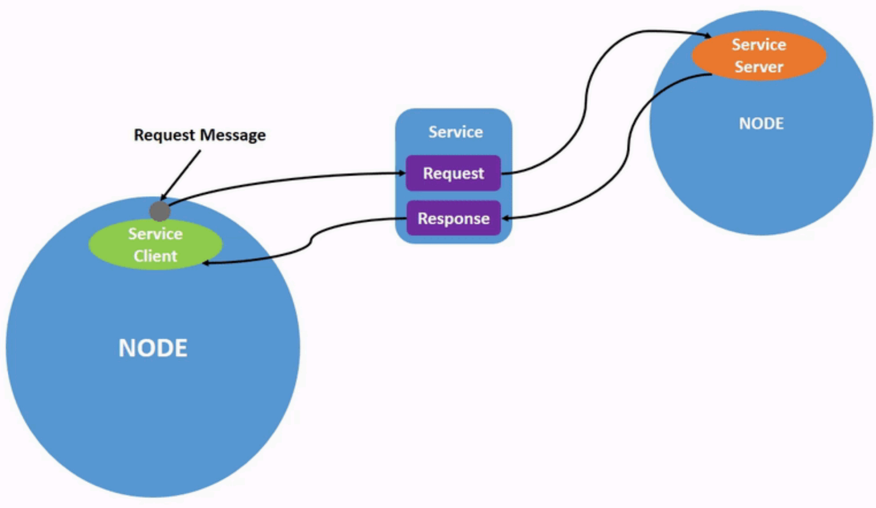
Lab Session 5
ROS Services
(Server node + Client Node + Creating custom services)

The publish / subscribe model is a very flexible communication paradigm, but its many-to-many one-way transport is not appropriate for RPC request / reply interactions, which are often required in a distributed system. Request / reply is done via a *Service,* which is defined by a pair of messages(one for the request and one for the reply). A providing ROS node offers a service under a string name, and a client calls the service by sending the request message and awaiting the reply.



But what is the difference between publisher/subscriber and server/client?
Think about the following cases:

1. Think of a robot and a simulator with the robot's model. The simulator needs to update the robot's model in real-time (as the robot in the real world changes configuration, the simulator needs to update the model to reflect that change such that the robot model in the simulator is always up-to-date with the current configuration of the real robot).

2. Think of a node that controls a robot cashier. This node requires to detect a customer using the robot's camera to start the interaction.

In the first case, you will need a publishing/subscribing model because the simulator will need data to flow in real-time while doing something else. So the robot will publish its joint values to a topic and the simulator will subscribe to that topic with a callback function that will update the robot model in the simulator in real-time **asynchronously**.
In the second case, however, you don't want a node to constantly check for a customer and is not something you want to do constantly. You know in your program logic when you need to detect a customer. When you first start your node, you know that you need to block and wait for a customer to come in. It's more appropriate here to use a service. When you want to detect a customer you send a request to the server (and as a result your program blocks waiting for a response). The server will use the camera to detect a customer (using some detection algorithm) and will respond accordingly back to you.
Generally speaking, you will use publishing/subscribing when you need data to flow constantly and you want to act on these data asynchronously, and services when you need a specific calculation to happen **synchronously.**

1. Creating our own Service

Defining our service is very similar to how we defined our own message previously. We are going to create a new /srv directory inside our package and we are going to add a new file called geometry.srv. like so:

```
$ roscd <your_package>
$ mkdir srv
$ cd srv
$ gedit geometry.srv
```

Add the following lines and save the file:

```
float32 length
float32 width
float32 height
- - -
float32 area
float32 volume
```

Next we are going to edit our package.xml file as well as the CMAKELISTS.txt file.

Please note: some of these steps will have already been done if you have previously created custom messages for your package

open the package.xml file and add these two line

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

1. Add message_generation to your list of dependencies (lines 10 -> 14) so it looks like this:

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

2.Uncomment lines 58 -> 62 and add the service file

```
add_service_files(
  FILES
  geometry.srv
)
```

3. Uncomment lines 73 -> 76 so it looks like this:

```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

After modifying both files, we can now execute $catkin_make inside our workspace.
Now use the $rossrv show command to ensure our message is visible within our environment (make sure your workspace is sourced)

```
$ rossrv show geometry
```

2. Creating the Server Node

In the /src directory of your package create a new python script with the name server.py and add the following code inside:

```
import rospy
from manara_test1.srv import *
def geometry_server(request):
    print("I Was Called Upon")
    response = geometryResponse()
    response.area = request.width * request.length
    response.volume = response.area * request.height
    return response

def serv():
    rospy.init_node('SERVER_NODE')
    rospy.Service('getGeometry', geometry, geometry_server)
    print('Server is running')
    rospy.spin()

if __name__ == '__main__':
    serv()
```

Then add the node inside the CMAKELISTS.txt file as usual and run the node using $rosrun:

```
$ rosrun <your_package> server
```

After running the node, we can notice a new service called (getGeometry) is shown:

```
$ rosservice show getGeometry
```

We can directly call the service from the terminal window. Say we want to get the area and volume of a rectangle with the following dimensions (Length, Width, Height) = (1.2, 5.2, 8.9)

We can do the following inside our terminal:

```
$ rosservice call /getGeometry 1.2 5.2 8.9
```

3. Creating the Client Node

Say we want to create a node where the dimensions of the rectangle is given is arguments to a node, and we want the node to print out the area and the volume. In the /src directory of your package create a new python script with the name client.py and add the following code inside:

```
import rospy
from manara_test1.srv import *
from sys import argv

def get_geometry_client(l, w, h):
    rospy.wait_for_service('getGeometry')
    server = rospy.ServiceProxy('getGeometry', geometry)
    request = server(l, w, h)

    return request.area, request.volume

if __name__ == '__main__':

    area ,volume = get_geometry_client(float(argv[1]),
float(argv[2]), float(argv[3]))

    print(f'The Area is : {area}\nThe Volume is: {volume}')
```

After adding the node to our CMAKELISTS.txt file and running $catkin_make, we can run:

```
$ rosrun <your_package> client 1.2 5.2 8.9
```
