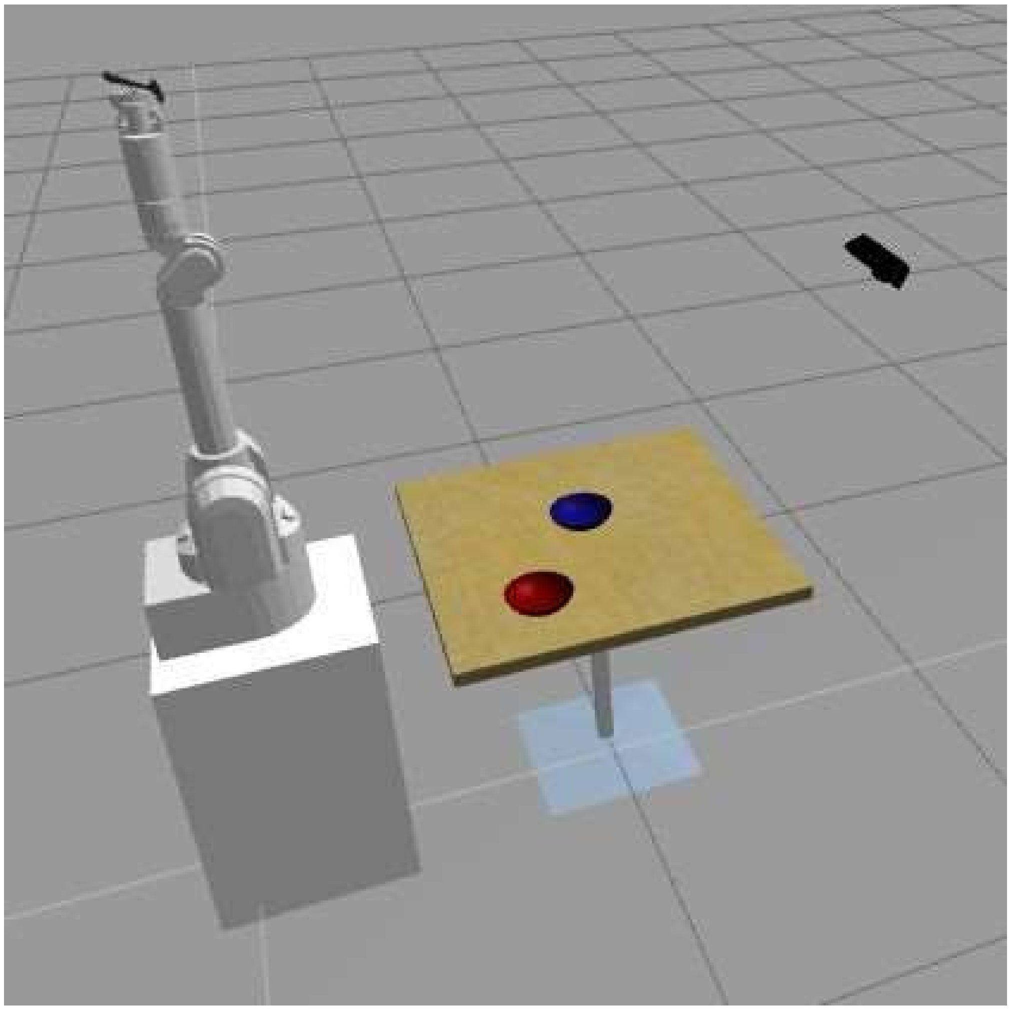
ROS Basics in 5 days (C++)

Unit 5 Services in ROS: Clients



- Summary -

Estimated time to completion: 2.5 hours

What will you learn with this unit?

- What a service is
- How to manage services of a robot
- How to call a service

- End of Summary -

Congratulations! You now know **75%** of ROS-basics!
The reason is that, with topics, you can do more or less whatever you want and need for your astromech droid. Many ROS packages only use topics and have the work perfectly done.

Then why do you need to learn about **services**?
Well, that's because for some cases, topics are insufficient or just too cumbersome to use. Of course, you can destroy the *Death Star* with a stick, but you will just spend ages doing it. Better tell Luke SkyWalker to do it for you, right? Well, it's the same with services. They just make life easier.

5.1 Topics - Services - Actions

To understand what services are and when to use them, you have to compare them with topics and actions. Imagine you have your own personal BB-8 robot. It has a laser sensor, a face-recognition system, and a navigation system. The laser will use a **Topic** to publish all of the laser readings at 20hz. We use a topic because we need to have that information available all the time for other ROS systems, such as the navigation system.

The Face-recognition system will provide a **Service**. Your ROS program will call that service and **WAIT** until it gives you the name of the person BB-8 has in front of it.
The navigation system will provide an **Action**. Your ROS program will call the action to move the robot somewhere, and **WHILE** it's performing that task, your program will perform other tasks, such as complain about how tiring C-3PO is. And that action will give you **Feedback** (for example: distance left to the desired coordinates) along the process of moving to the coordinates.

So... What's the difference between a **Service** and an **Action**?
Services are **Synchronous**. When your ROS program calls a service, your program can't continue until it receives a result from the service.
Actions are **Asynchronous**. It's like launching a new thread. When your ROS program calls an action, your program can perform other tasks while the action is being performed in another thread.

Conclusion: Use services when your program can't continue until it receives the result from the service.

5.2 Services Introduction

Enough talk for now, let's go play with a robot and launch a prepared demo!

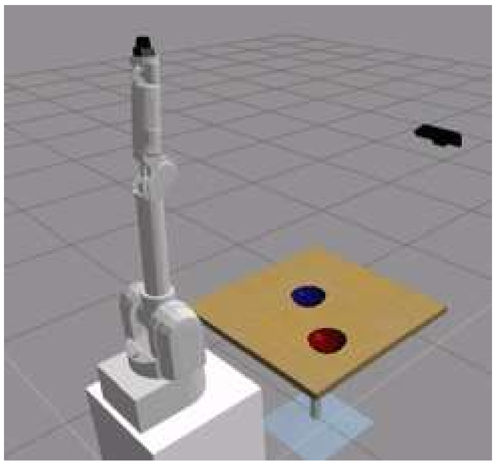
- Example 5.1 -

Go to the WebShell and do the following:

▶ Execute in Shell #1

In [ ]:
```
roslaunch iri_wam_aff_demo start_demo.launch
```

This will make the Wam robot-arm of the simulation move.
You should get something similar to this:



- End of Example 5.1 -

5.2.1 What did you do just now?

The launch file has launched two nodes (Yes! You can launch more than one node with a single launch file):

- /iri_wam_reproduce_trajectory
- /iri_wam_aff_demo

The first node provides the ***/execute_trajectory*** service. This is the node that contains the **service**. The second node, performs calls to that service. When the service is called, the robot will execute a given trajectory.

5.2.2 Let's learn more about services

- Example 5.2 -

Let's see a list of the available services in the Wam robot. For that, open another shell.
You have to leave the *start_demo.launch* running, otherwise the services won't be there to see them.

Execute the following command in a different shell from the one that has the roslaunch **start_demo.launch** running:

▶ Execute in Shell #2

In [ ]:
```
rosservice list
```

You should see something like the following image, listing all the services available:

Shell #2 Output

In [ ]:
```
user ~ $ rosservice list
/camera/rgb/image_raw/compressed/set_parameters
/camera/rgb/image_raw/compressedDepth/set_parameters
/camera/rgb/image_raw/theora/set_parameters
/camera/set_camera_info
/camera/set_parameters
/execute_trajectory
/gazebo/apply_body_wrench
...
```

WARNING: If the */execute_trajectory* server is not listed, maybe that's because you stopped the *start_demo.launch*. If that is the case, launch it again and check for the service.

There are a few services, aren't there? Some refer to the simulator system (*/gazebo/...*), and others refer to the Kinect Camera (*/camera/...*) or are given by the robot himself (*/iri_wam/...*). You can see how the service */execute_trajectory* is listed there.

You can get more information about any service by issuing the following command:

In [ ]:
```
rosservice info /name_of_your_service
```

Execute the following command to know more about the service */execute_trajectory*

▶ *Execute in Shell #2*

In [ ]:
```
rosservice info /execute_trajectory
```

Shell #2 Output

In [ ]:
```
user ~ $ rosservice info /execute_trajectory
Node: /iri_wam_reproduce_trajectory
URI: rosrpc://ip-172-31-17-169:35175
Type: iri_wam_reproduce_trajectory/ExecTraj
Args: file
```

Here you have two relevant parts of data.

- ***Node**: It states the node that provides (has created) that service.*
- ***Type**: It refers to the kind of message used by this service. It has the same structure as topics do. It's always made of **package_where_the_service_message_is_defined** / **Name_of_the_File_where_Service_message_is_defined**. In this case, the package is **iri_wam_reproduce_trajectory**, and the file where the Service Message is defined is called **ExecTraj**.*
- ***Args**: Here you can find the arguments that this service takes when called. In this case, it only takes a **trajectory file path** stored in the variable called **file**.*

- End of Example 5.2 -

5.2.3 Want to know how this /execute_trajectory service is started?

*Here you have an example on how to check the **start_demo.launch** file through WebShell.*

- Example 5.3 -

Do you remember how to go directly to a package and where to find the launch files?

▶ **Execute in Shell #2**

In [ ]:
```
roscd iri_wam_aff_demo
cd launch/
cat start_demo.launch
```

You should get something like this:

In [ ]:
```
<launch>

  <include file="$(find iri_wam_reproduce_trajectory)/launch/start_service.lau

  <node pkg ="iri_wam_aff_demo"
        type="iri_wam_aff_demo_node"
        name="iri_wam_aff_demo"
        output="screen">
  </node>

</launch>
```

Some interesting things here:

1) The first part of the launch file calls another launch file called start_service.launch.

That launch file starts the node that provides the /execute_trajectory service. Note that it's using a special ROS launch file function to find the path of the package given.

In [ ]:
```
<include file="$(find package_where_launch_is)/launch/my_launch_file.launch"/>
```

*2) The second part launches a node just as you learned in the **ROS Basics Unit**. That node is the one that will call the /execute_trajectory service in order to make the robot move.*

- End of Example 5.3 -

5.3 How to call a service

You can call a service manually from the console. This is very useful for testing and having a basic idea of how the service works.

In [ ]:
```
rosservice call /the_service_name TAB-TAB
```

Info: TAB-TAB means that you have to quickly press the TAB key twice. This will autocomplete the structure of the Service message to be sent for you. Then, you only have to fill in the values.

- Example 5.4 -

*Let's call the service with the name **/trajectory_by_name** by issuing the following command. But before being able to call this Service, you will have to launch it. For doing so you can execute the following command:*

▶ *Execute in Shell #1*

In [ ]:
```
roslaunch trajectory_by_name start_service.launch
```

Now let's call the Service.

▶ *Execute in Shell #2*

In [ ]:
```
rosservice call /trajectory_by_name [TAB]+[TAB]
```

When you [TAB]+[TAB], an extra element appears. ROS autocompletes with the structure needed to input/request the service.
In this case, it gives the following structure:

"traj_name: 'the_name_of_the_trajectory_you_want_to_execute'"

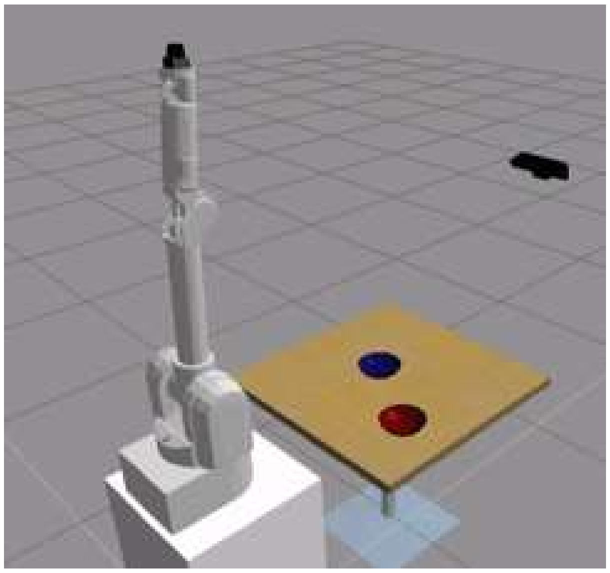
*The **/trajectory_by_name** Service is a service provided by the Robot Ignite Academy as an example, that allows you **to execute a specified trajectory** with the robotic arm.*

*Use that service to execute one trajectory with the WAM Arm. The trajectories that are available are the following ones: **init_pos**, **get_food** and **release_food**.*

▶ *Execute in Shell #2*

In [ ]:
```
rosservice call /trajectory_by_name "traj_name: 'get_food'"
```

Did it work? You should have seen the robotic arm executing a trajectory, like the one at the beginning of this Chapter:



You can now try to call the service providing different trajectory names (from the ones indicated above), and see how the robot executes each one of them.

- End of Example 5.4 -

5.4 How to interact with a service programatically

- Exercise 5.1 -

- *Create a new package named **service_client_pkg**. When creating the package, add as dependencies the packages **roscpp** and also **trajectory_by_name_srv**.*
- *Inside the src folder of the package, create a new file named **simple_service_client.cpp**. Inside this file, copy the contents of simple_service_client.cpp*
- *Create a launch file for launching this code. Keep in mind that, in order to be able to call the /trajectory_by_name service, you need to have it running.*
- *Do the necessary modifications to your **CMakeLists.txt** file, and compile the package.*
- *Execute the launch file to run your executable.*

What do you think it will do?

C++ Program {5.1}: simple_service_client.cpp

In [ ]:
```
#include "ros/ros.h"
#include "trajectory_by_name_srv/TrajByName.h"
// Import the service message used by the service /trajectory_by_name

int main(int argc, char **argv)
{
  ros::init(argc, argv, "service_client"); // Initialise a ROS node with the r
  ros::NodeHandle nh;

  // Create the connection to the service /trajectory_by_name
  ros::service::waitForService("/trajectory_by_name");  // wait for service tc
  ros::ServiceClient traj_by_name_service = nh.serviceClient<trajectory_by_nam
  trajectory_by_name_srv::TrajByName srv; // Create an object of type TrajByNc
  srv.request.traj_name = "release_food"; // Fill the variable traj_name with

  if (traj_by_name_service.call(srv)) // Send through the connection the name
  {
    ROS_INFO("%s", srv.response.status_message.c_str()); // Print the result g
  }
  else
  {
    ROS_ERROR("Failed to call service /trajectory_by_name");
    return 1;
  }

  return 0;
}
```

END C++ Program {5.1}: simple_service_client.cpp

You should have seen the robotic arm execute the trajectory after executing this snippet of code.

- End of Exercise 5.1 -

5.4.1 How to know the structure of the service message used by the service?

- Example 5.5 -

*You can do a **rosservice info** to know the type of service message that it uses.*

In [ ]:
```
rosservice info /name_of_the_service
```

This will return you the ***name_of_the_package/Name_of_Service_message***

Then, you can explore the structure of that service message with the following command:

In [ ]:
```
rossrv show name_of_the_package/Name_of_Service_message
```

Execute the following command to see what is the service message used by the ***/trajectory_by_name*** *service:*

▶ Execute in Shell #2

In [ ]:
```
rosservice info /trajectory_by_name
```

You should see something like this:

Shell #2 Output

In [ ]:
```
Node: /traj_by_name_node
URI: rosrpc://rosdscomputer:38301
Type: trajectory_by_name_srv/TrajByName
Args: traj_name
```

Now, execute the following command to see the structure of the message ***TrajByName****:*

▶ Execute in Shell #2

In [ ]:
```
rossrv show trajectory_by_name_srv/TrajByName
```

You should see something like this:

Shell #2 Output

In [ ]:
```
user catkin_ws $ rossrv show trajectory_by_name_srv/TrajByName
string traj_name
---
bool success
string status_message
```

Does it seem familiar? It should, because it's the same structure as the Topics messages, with some addons.

1- Service messages have the extension .srv. Remember that Topic messages have the extension .msg

2- Service messages are defined inside a srv directory instead of a msg directory.

You can type the following command to check it.

▶ ***Execute in Shell #2***

In [ ]:
```
roscd trajectory_by_name_srv; ls srv
```

3- Service messages have TWO parts:

REQUEST

RESPONSE

*In the case of the TrajByName service, **REQUEST** contains a string called **traj_name** and **RESPONSE** is composed of a boolean named **success**, and a string named **status_message**.*

*The Number of elements on each part of the service message can vary depending on the service needs. You can even put none if you find that it is irrelevant for your service. The important part of the message is the three dashes **---**, because they define the file as a Service Message.*

Summarizing:

*The **REQUEST** is the part of the service message that defines **HOW you will do a call** to your service. This means, what variables you will have to pass to the Service Server so that it is able to complete its task.*

*The **RESPONSE** is the part of the service message that defines **HOW your service will respond** after completing its functionality. If, for instance, it will return an string with a certaing message saying that everything went well, or if it will return nothing, etc...*

- End of Example 5.5 -

- Exercise 5.2 -

- *Create a launch that starts the **/execute_trajectory** service , called **my_robot_arm_demo.launch**. As explained in the Example 5.3, this service is launched by the launch file **start_service.launch**, which is in the package **iri_wam_reproduce_trajectory**.*
- *Get information of what type of service message does this execute_trajectory service use, as explained in Example 5.5.*
- *Make the arm robot move following a trajectory, which is specified in a file.
Modify the previous code of Exercise 5.1, which called the **/trajectory_by_name** service, to call now the **/execute_trajectory** service instead.*
- *Here you have the code necessary to get the path to the trajectory files based on the package where they are. Here, the trajectory file **get_food.txt** is selected, but you can use any of the available in the **config** folder of the **iri_wam_reproduce_trajectory** package.*

In [ ]:
```
#include <ros/package.h>

// This ros::package::getPath works in the same way as $(find name_of_package)
trajectory.request.file = ros::package::getPath("iri_wam_reproduce_trajectory'
```

- *In order to be able to properly compile the the **ros::package::getPath()** function, you will need to add dependencies to the **roslib** library. You can do that by modifying the **find_package()** function in the **CmakeLists.txt** file. Like this:*

In [ ]:
```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  roslib
)
```

- *Modify the main launch file **my_robot_arm_demo.launch**, so that now it also launches the new C++ code you have just created.*
- *Compile again your package.*
- *Finally, execute the **my_robot_arm_demo.launch** file and see how the robot performs the trajectory.*

- End of Exercise 5.2 -

5.5 Summary

Services provide functionality to other nodes. If, for instance, a node knows how to delete an object on the simulation, it can provide that functionality to other nodes through a service call, so they can call the service when they need to delete something.

Services allow the specialization of nodes (each node specializes in one thing).