ROS Basics in 5 days (C++)

Unit 6 Services in ROS: Servers & Messages

- Summary -

Estimated time to completion: 3 hours

What will you learn with this unit?

- How to give a service
- How to create your own service server message

- End of Summary -

6.1 Service Server

Until now, you have called services that others provided. Now, you are going to create your own.

- Exercise 6.3 -

- Create a new package named **service_server_pkg**. When creating the package, add **roscpp** as a dependency.
- Inside the src folder of the package, create a new file named **simple_service_server.cpp**. Inside this file, copy the contents of simple_service_server.cpp
- Create a launch file for launching this code.
- Do the necessary modifications to your **CMakeLists.txt** file, and compile the package.
- Execute the launch file to run your executable.

C++ Program {6.2}: simple_service_server.cpp

In [ ]:
```
#include "ros/ros.h"
#include "std_srvs/Empty.h"
// Import the service message header file generated from the Empty.srv message

// We define the callback function of the service
bool my_callback(std_srvs::Empty::Request  &req,
                 std_srvs::Empty::Response &res)
{
  // res.some_variable = req.some_variable + req.other_variable;
  ROS_INFO("My_callback has been called"); // We print an string whenever the
  return true;
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "service_server");
  ros::NodeHandle nh;

  ros::ServiceServer my_service = nh.advertiseService("/my_service", my_callba
  ros::spin(); // mantain the service open.

  return 0;
}
```

END C++ Program {6.2}: simple_service_server.cpp

- End of Exercise 6.3 -

Did something happen?

Of course not! At the moment, you have just created and started the Service Server. So basically, you have made this service available for anyone to call it.

This means that if you do a ***rosservice list***, you will be able to visualize this service on the list of available services.

▶ Execute in Shell #2

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

On the list of all available services, you should see the **/my_service** service.

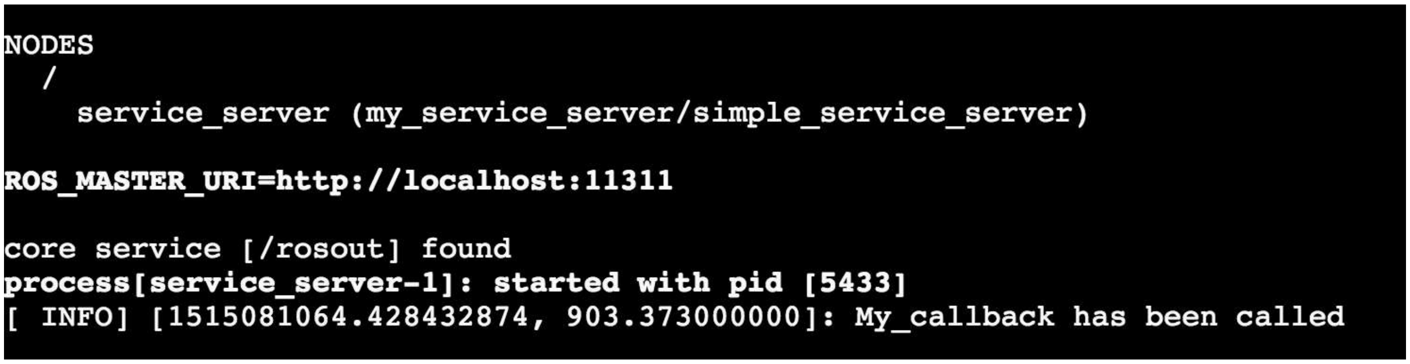
In [ ]:
```
/base_controller/command_select
/bb8/camera1/image_raw/compressed/set_parameters
/bb8/camera1/image_raw/compressedDepth/set_parameters
/bb8/camera1/image_raw/theora/set_parameters
...
/my_service
...
```

Now, you have to actually **CALL** it. So, call the **/my_service** service manually. Remember the calling structure discussed in the previous chapter and don't forget to TAB-TAB to autocomplete the structure of the Service message.

▶ Execute in Shell #2

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

Did it work? You should've seen the message, **'My callback function has been called'**, printed at the output of the shell where you executed the service server code. Great!


```
NODES
  /
    service_server (my_service_server/simple_service_server)

ROS_MASTER_URI=http://localhost:11311

core service [/rosout] found
process[service_server-1]: started with pid [5433]
[ INFO] [1515081064.428432874, 903.373000000]: My_callback has been called
```

INFO: Note that, in the example, there is a commented line in **my_callback function**. It gives you an example of how you would access the **REQUEST** given by the caller of your service. It's always **req.*variables_in_the_request_part_of_srv_message***. In this case, this is not necessary because we are working with the Empty message, which is a special message that doesn't contain anything.

So, for instance, let's do a flashback to the previous chapter. Do you remember Example 6.5, where you had to perform calls to a service in order to execute a certain trajectory with the robotic arm? Well, for that case, you were passing the name of the trajectory to execute to the Service Server in a variable called **traj_name**. So, if you wanted to access the value of that **traj_name** variable in the Service Server, you would have to do it like this:

In [ ]:
```
request.traj_name
```

Quite simple, right?

That commented line also shows you how you would return the **RESPONSE** of the service. For that, you have to access the variables in the **RESPONSE** part of the message. It would be like this: **res.*variables_in_the_response_part_of_srv_message***.

And why do we use **req** and **res** for accessing the **REQUEST** and **RESPONSE** parts of the service message? Well, it's just because we are defining this variables here:

In [ ]:
```
bool my_callback(std_srvs::Empty::Request  &req,
                 std_srvs::Empty::Response &res)
```

- Exercise 6.4 -

- The objective of Exercise 6.4 is to create a service that, when called, will make the BB-8 robot move in a circle-like trajectory.

- You can work on a new package or use one of the ones you have already created.

- Create a Service Server that accepts an **Empty** service message and activates the circle movement. This service could be called **/move_bb8_in_circle**.

  You will place the necessary code into a new C++ file named **bb8_move_in_circle_service_server.cpp**. You can use the C++ file simple_service_server.cpp as an example.

  To move the BB-8 robot, you just have to write into the /cmd_vel topic, as you did in the Topics Units.

- Create a launch file called **start_bb8_move_in_circle_service_server.launch**. Inside it, you have to start a node that launches the **bb8_move_in_circle_service_server.cpp**.

- Launch **start_bb8_move_in_circle_service_server.launch** and check that when called through the WebShell, BB-8 moves in a circle.

- Create a new C++ file, called **bb8_move_in_circle_service_client.cpp**, that calls the service **/move_bb8_in_circle**. Remember how it was done in the previous Chapter: **Services Part1**.
  Then, generate a new launch file, called **call_bb8_move_in_circle_service_server.launch**, that executes the code in the **bb8_move_in_circle_service_client.cpp** file.

- Finally, when you launch this **call_bb8_move_in_circle_service_server.launch** file, BB-8 should move in a circle.

- End of Exercise 6.4 -

6.2 How to create your own service message

So, what if none of the service messages that are available in ROS fit your needs? Then, you create your own message, as you did with the Topic messages.

In order to create a service message, you will have to follow the next steps:

- Example 6.6 -

1) Create a package like this:

▶ Execute in Shell #1

In [ ]:
```
roscd;cd ..;cd src
catkin_create_pkg my_custom_srv_msg_pkg roscpp
```

2) Create your own Service message with the following structure. You can put as many variables as you need, of any type supported by ROS: [ROS Message Types (http://wiki.ros.org/msg)](http://wiki.ros.org/msg). Create a ***srv*** folder inside your package , as you did with the topics ***msg*** folder. Then, inside this ***srv*** folder, create a file called ***MyCustomServiceMessage.srv***. You can create it with the IDE or the WebShell, as you wish.

▶ Execute in Shell #1

In [ ]:
```
roscd my_custom_srv_msg_pkg/
mkdir srv
vim srv/MyCustomServiceMessage.srv
```

You can also create the ***MyCustomServiceMessage.srv*** through the IDE, if you don't feel confortable with vim.

The **MyCustomServiceMessage.srv** could be something like this:

In [ ]:
```
int32 duration    # The time (in seconds) during which BB-8 will keep moving
---
bool success          # Did it achieve it?
```

6.2.1 Prepare CMakeLists.txt and package.xml for custom service compilation

You have to edit two files in the package, in a way similar to how we explained it for Topics:

- CMakeLists.txt
- package.xml

Modification of CMakeLists.txt

You will have to edit four functions inside CMakeLists.txt:

- find_package()
- add_service_files()
- generate_messages()
- catkin_package()

I. find_package()

All the packages needed to COMPILE the messages of topic, services, and actions go here. It's only getting its paths, and not really importing them to be used in the compilation.
The same packages you write here will go in **package.xml,** stating them as **build_depend**.

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

II. add_service_files()

This function contains a list with all of the service messages defined in this package (defined in the srv folder).
For our example:

In [ ]:
```
add_service_files(
  FILES
  MyCustomServiceMessage.srv
)
```

III. generate_messages()

Here is where the packages needed for the service messages compilation are imported.

In [ ]:
```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

IV. catkin_package()

State here all of the packages that will be needed by someone that executes something from your package. All of the packages stated here must be in the **package.xml** file as **<exec_depend>**.

In [ ]:
```
catkin_package(
      CATKIN_DEPENDS
      roscpp
)
```

Once you're done, you should have something similar to this:

In [ ]:

```
cmake_minimum_required(VERSION 2.8.3)
project(my_custom_srv_msg_pkg)


## Here is where all the packages needed to COMPILE the messages of topic, ser
## It's only getting its paths, and not really importing them to be used in th
## It's only for further functions in CMakeLists.txt to be able to find those
## In package.xml you have to state them as build
find_package(catkin REQUIRED COMPONENTS
  roscpp
  std_msgs
  message_generation
)

## Generate services in the 'srv' folder
## In this function will be all the action messages of this package ( in the a
## You can state that it gets all the actions inside the action directory: DIR
## Or just the action messages stated explicitly: FILES my_custom_action.actio
## In your case, you only need to do one of two things, as you wish.
add_service_files(
  FILES
  MyCustomServiceMessage.srv
)

## Here is where the packages needed for the action messages compilation are i
generate_messages(
  DEPENDENCIES
  std_msgs
)

## State here all the packages that will be needed by someone that executes so
## All the packages stated here must be in the package.xml as exec_depend
catkin_package(
  CATKIN_DEPENDS roscpp
)


include_directories(
  ${catkin_INCLUDE_DIRS}
)
```

Modification of package.xml:

Just add these 3 lines to the package.xml file.

In [ ]:
```
<build_depend>message_generation</build_depend>

<build_export_depend>message_runtime</build_export_depend>
<exec_depend>message_runtime</exec_depend>
```

You should have something similar to:

In [ ]:
```
<?xml version="1.0"?>
<package format="2">
  <name>my_custom_srv_msg_pkg</name>
  <version>0.0.0</version>
  <description>The my_custom_srv_msg_pkg package</description>

  <maintainer email="user@todo.todo">user</maintainer>

  <license>TODO</license>

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>message_generation</build_depend>
  <build_export_depend>roscpp</build_export_depend>
  <exec_depend>roscpp</exec_depend>
  <build_export_depend>message_runtime</build_export_depend>
  <exec_depend>message_runtime</exec_depend>

  <export>
  </export>
</package>
```

Once done, compile your package and source the newly generated messages:

In [ ]:
```
roscd;cd ..
catkin_make
source devel/setup.bash
```

Important!! When you compile new messages through catkin_make, there is an extra step that needs to be done. You have to type in the WebShell, in the **catkin_ws** directory, the following command: ***source devel/setup.bash***.

This command executes the bash file that sets, among other things, the newly generated messages created with ***catkin_make***.

To check that you have the new service msg in your system prepared for use, type the following:

▶ **Execute in Shell #1**

In [ ]:
```
rossrv list | grep MyCustomServiceMessage
```

It should output something like:

Shell #1 Output

In [ ]:
```
user ~ $ rossrv list | grep MyCustomServiceMessage
my_custom_srv_msg_pkg/MyCustomServiceMessage
```

That's it! You have created your own service msg.

- End of Example 6.6 -

6.3 Using a Custom Service Message

If you want to use the custom message you have just generated in a Service Server, you will need to create a program similar to the one below:

C++ Program {6.3}: custom_service_server.cpp

In [ ]:
```
#include "ros/ros.h"
#include "my_custom_srv_msg_pkg/MyCustomServiceMessage.h"

bool my_callback(my_custom_srv_msg_pkg::MyCustomServiceMessage::Request  &req,
                 my_custom_srv_msg_pkg::MyCustomServiceMessage::Response &res)
{
  ROS_INFO("Request Data==> radius=%f, repetitions=%d", req.radius, req.repeti
  if (req.radius > 5.0)
  {
    res.success = true;
    ROS_INFO("sending back response:true");
  }
  else
  {
    res.success = false;
    ROS_INFO("sending back response:false");
  }

  return true;
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "service_server");
  ros::NodeHandle nh;

  ros::ServiceServer my_service = nh.advertiseService("/my_service", my_callba
  ros::spin();

  return 0;
}
```

END C++ Program {6.3}: custom_service_server.cpp

And the **CMakeLists.txt** needed to make this work:

In [ ]:
```
cmake_minimum_required(VERSION 2.8.3)
project(my_custom_srv_msg_pkg)

find_package(catkin REQUIRED COMPONENTS
  roscpp
  std_msgs
  message_generation
)


add_service_files(
   FILES
   MyCustomServiceMessage.srv
 )

generate_messages(
   DEPENDENCIES
   std_msgs
 )


catkin_package(
  CATKIN_DEPENDS roscpp
)

include_directories(include ${catkin_INCLUDE_DIRS})


add_executable(custom_service_server src/custom_service_server.cpp)
target_link_libraries(custom_service_server ${catkin_LIBRARIES})
add_dependencies(custom_service_server ${my_custom_srv_msg_pkg_EXPORTED_TARGET
```

You also need to know that, if you wanted to use this Custom Service Message in another ROS package, you will need to add the following line to the **find_package()** function of the **CMakeLists.txt** file of the package that wants to use this message:

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

This is because at compilation time, the package that wants to use a custom message needs to know in which ROS package this message is. Otherwise, its compilation will fail. Adding the package that contains the message to the **find_package()** function ensures that the package (and therefore, the messages that it contains) will be found.

- Exercise 6.5 -

- Create a new C++ file called **bb8_move_custom_service_server.cpp**. Inside this file, modify the code you used in **Exercise 6.4**, which contained a Service Server that accepted an Empty Service message to activate the circle movement. This new service will be called **/move_bb8_in_circle_custom**. This new service will have to be called through a custom service message. The structure of this custom message is presented below:

In [ ]:
```
int32 duration    # The time (in seconds) during which BB-8 will keep moving
---
bool success      # Did it achieve it?
```

- Use the data passed to this new **/move_bb8_in_circle_custom** to change the BB-8 behavior. During the specified duration time, BB-8 will keep moving in circles. Once this time has ended, BB-8 will then stop its movement and the Service Server will return a **True** value (in the **success** variable). Keep in mind that even after BB-8 stops moving, there might still be some rotation on the robot, due to inertia.

- Create a new launch file called **start_bb8_move_custom_service_server.launch** that launches the new **bb8_move_custom_service_server.cpp** file.

- Test that when calling this new **/move_bb8_in_circle_custom** service, BB-8 moves accordingly.

- Create a new C++ file, called **call_bb8_move_custom_service_server.cpp** that calls the service **/move_bb8_in_circle_custom**. Remember how it was done in **Unit 3 Services Part 1**.

  Then, generate a new launch file, called **call_bb8_move_custom_service_server.launch**, that executes the **call_bb8_move_custom_service_server.cpp** through a node.

- End of Exercise 6.5 -

6.4 Summary

Let's do a quick summary of the most important parts of **ROS Services**, just to try to put everything in place.

A **ROS Service** provides a certain functionality to your robot. A ROS Service is composed of two parts:

- **Service Server**: It is the one that **PROVIDES** the functionality. Whatever you want your Service to do, you have to place it in the Service Server.
- **Service Client**: It is the one that **CALLS** the functionality provided by the Service Server. That is, it CALLS the Service Server.

ROS Services use a special service message, which is composed of two parts:

- **Request**: The request is the part of the message that is used to CALL the Service. Therefore, it is sent by the Service Client to the Service Server.
- **Response**: The response is the part of the message that is returned by the Service Server to the Service Client, once the Service has finished.

ROS Services are synchronous. This means that whenever you CALL a Service Server, you have to wait until the Service has finished (and returns a response) before you can do other stuff with your robot.

6.5 Services Quiz

For evaluating this Quiz, we will ask you to perform different tasks. For each task, very **specific instructions** will be provided: name of the package, names of the launch files and Cpp scripts, topic names to use, etc.

It is **VERY IMPORTANT** that you strictly follow these instructions, since they will allow our automated correction system to properly grade your Quiz, and assign a score to it. If the names you use are different from the ones specified in the exam instructions, your exercise will be marked as **FAILED**, even though it works correctly.

- Upgrade the C++ file called **bb8_move_custom_service_server.cpp**. Modify the code you used in **Exercise 6.5**, which contained a Service Server that accepted a custom Service message to activate the circle movement (with a defined duration). This new service will be called **/move_bb8_in_square_custom**. This new service will have to use service messages of the **BB8CustomServiceMessage** type, which are defined here:

The **BB8CustomServiceMessage.srv** could be something like this:

In [ ]:
```
float64 side          # The distance of each side of the square
int32 repetitions     # The number of times BB-8 has to execute the square move
---
bool success          # Did it achieve it?
```

NOTE: The **side** variable doesn't represent the real distance of each size of the square. It's just a variable that will be used to change the size of the square. The bigger the **side** variable is, the bigger the square performed by the BB-8 robot will be.

- In the previous exercises, you were triggering a circle movement when calling to your service. In this new service, the movement triggered will have to be a square, like in the image below:

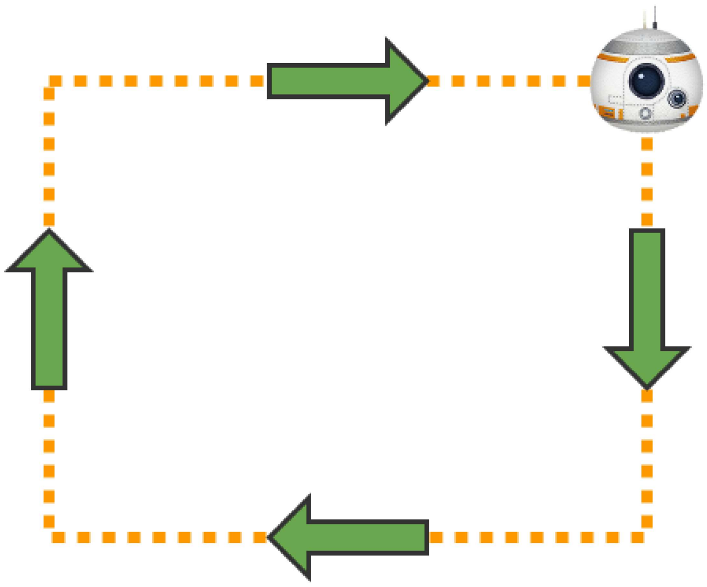


Fig.6.1 - BB-8 Square Movement Diagram

- Use the data passed to this new **/move_bb8_in_square_custom** to change the way BB-8 moves. Depending on the **side** value, the service must move the BB-8 robot in a square movement based on the **side** given.
  Also, the BB-8 must repeat the shape as many times as indicated in the **repetitions** variable of the message.
  Finally, it must return **True** if everything went OK in the **success** variable.

NOTE: The **side** variable doesn't represent the real distance of each size of the square. It's just a variable that will be used to change the size of the square. The bigger the **side** variable is, the bigger the square performed by the BB-8 robot will be.

- Create a new launch file, called **start_bb8_move_custom_service_server.launch**, that launches the new C++ file **bb8_move_custom_service_server.cpp**.

- Test that when calling this new **/move_bb8_in_square_custom** service, BB-8 moves accordingly. This means, the square is performed, taking into account the **side** and **repetitions** variables.

- Create a new service client that calls the service **/move_bb8_in_square_custom**, and makes BB-8 move in a small square **twice**, and in a bigger square **once**. It could be called **bb8_move_custom_service_client.cpp**. The launch file that starts it could be called **call_bb8_move_in_square_custom_service_server.launch**.

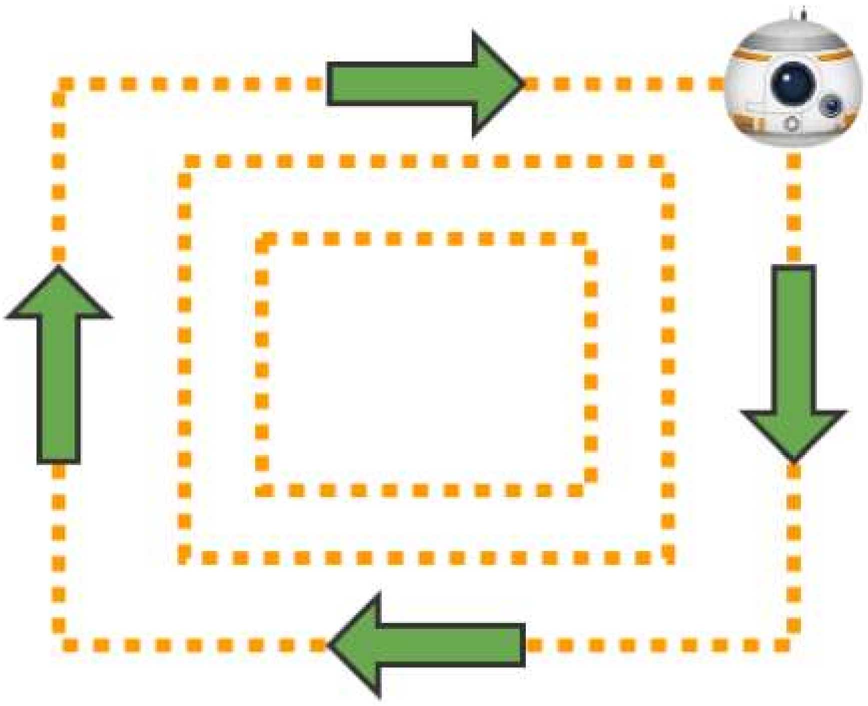
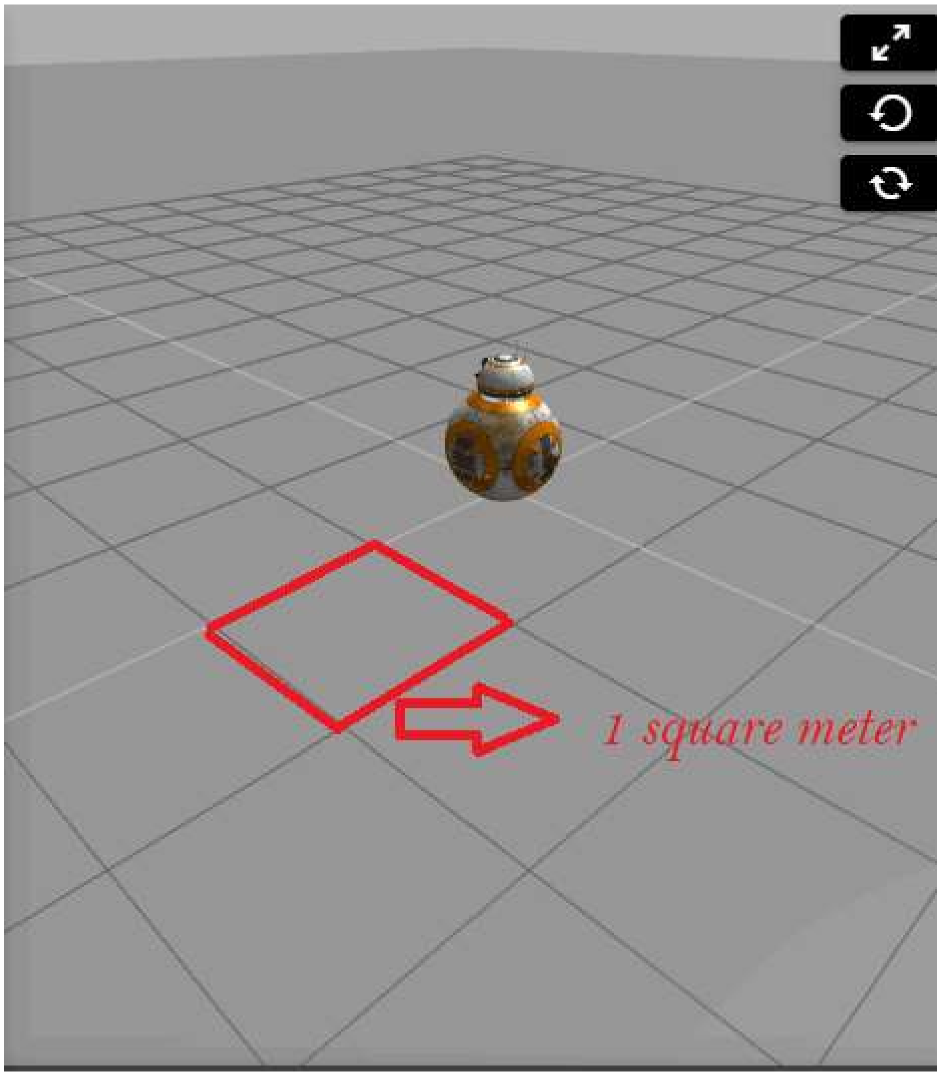


Fig.6.2 - BB-8 Dynamic Square Diagram

- The **small square** has to be of, at least, **1 sqm**. The **big square** has to be of, at least, **2 sqm**.
- Please take a square on the floor "tiles" in the image below (as seen on the simulation) as **1 sqm**.



Square Meter Specification

Specifications

- The name of the package where you'll place all the code related to the Quiz will be **services_quiz**.
- The name of the launch file that will start your Service Server will be `start_bb8_move_custom_service_server.launch` . **This should only start the service server, and not the service client.**

- The name of the service will be **/move_bb8_in_square_custom**.

- The name of your Service message file will be **BB8CustomServiceMessage.srv**.

- The name of the launch file that will start your Service Client will be `call_bb8_move_in_square_custom_service_server.launch` . **This launch file should not start the Service Server, only the Service Client**.

- Your service client script, `bb8_move_custom_service_client.cpp` , **must exit cleanly** after it completes the movements. Please also **ensure that it completes the movements within THREE minutes**.
  - If it does not exit after three minutes, it will be killed automatically and you may not get credit for work done even if it works as expected.
  - Do not use `ros::spin()` as it will block the script from exiting.

Grading Guide

The following will be checked, in order. *If a step fails, the steps following are skipped.*

1. Does the package exist?
2. Did the package compile successfully?
3. Is the custom service message created successfully?
4. Is the service server started by the servive server launch file?
5. Is the service client started by the servive client launch file?
6. Did the robot perform the expected movements?

The grader will provide as much feedback on any failed step so you can make corrections where necessary.

Quiz Correction

When you have finished the quiz, you can correct it in order to get a mark. For that, just click on the following button at the top of this notebook.

Final Mark

In case you fail the Quiz, or you don't get the desired mark, do not get frustrated! You will have the chance to resend the Quiz in order to improve your score.

Congratulations! You are now ready to add all of the services that you want to your own personal astromech droid!

English proofread