**Department of Electrical Engineering and   
Computer Science**

**Faculty Member:** Dr. Hafsa Iqbal **Date:** 19/03/2024

**Semester:** 8th **Group:** GP-1

**Lab Engineer:** Munadi Ahmad Sial

**EE-381 Robotics**

Lab 7: Parameter Server and Services in ROS

**Group Members**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | **PLO5 – CLO4** | | **PLO9 –**  **CLO4** |  |
| **Name** | **Reg. No** | **Analysis of Data in Lab Report** | **Modern Tool Usage** | **Individual and Team Work** | **Total Marks** |
|  |  | **10 Marks** | **5 Marks** | **5 Marks** | **20 Marks** |
| Muhammad Abdullah Sohail | 343642 |  |  |  |  |
| Muhammad Ahmed Mohsin | 333060 |  |  |  |  |
| Muhammad Umer | 345834 |  |  |  |  |
| Hassan Rizwan | 335753 |  |  |  |  |
| Muhammad Saad | 333414 |  |  |  |  |

**Table of Contents**

[2 Parameter Server and Services in ROS 3](#_Toc162887662)

[2.1 Introduction 3](#_Toc162887663)

[2.2 Objectives 3](#_Toc162887664)

[2.3 Theory 3](#_Toc162887665)

[3 Lab Tasks 4](#_Toc162887666)

[3.1 Lab Task 1 – Parameters in ROS 4](#_Toc162887667)

[3.2 Lab Task 2 – Services 5](#_Toc162887668)

[3.3 Lab Task 3 – Modified Service 8](#_Toc162887669)

# Parameter Server and Services in ROS

## Introduction

This laboratory exercise this lab will also introduce ROS parameters to add more flexibility to the robot programming. Additionally, this lab will introduce another method of communication (other than topics) in ROS known as services. This lab will involve creating the service definition file as well as the client and server nodes for the service.

## Objectives

The following are the main objectives of this lab:

* Implement Wall Following around a convex obstacle
* Declare and input parameters in the nodes
* Create a service definition (.srv)
* Implement a server node
* Implement a client node

## Theory

Besides topics, another mode of communication in ROS is through services. A service uses a client/server mechanism which is useful for tasks that are to be done quickly (such as turning a sensor off). Services need to be defined in service definition file (.srv) before they can be used. A server node provides a service while a client node requests for that service.

The use of ROS parameters allows setting up of variables of a script via the terminal. This is an important feature as changing variables from the script is usually not a good practice when deploying robotic platforms.

The terminal commands are given as:

**cd <directory>**  change directory

**cd..**  go back to previous directory

**pwd**  print the current directory

**ls** list the contents of the current directory

**python <script.py>** execute python script

A brief summary of the terminal commands needed for working in ROS are provided below:

**colcon build --packages-select <package\_name>**

build a specific package whenever a node is created or modified

**. install/setup.bash**

make the terminal “aware” of the workspace (notice the dot and space)

**ros2 pkg create --build-type ament\_python <package\_name>**

create a new package (must be done in src directory)

**ros2 pkg create --build-type ament\_python <package\_name> --dependencies rclpy std\_msgs geometry\_msgs sensor\_msgs**

create a new package with dependencies

**ros2 run <package\_name> <node\_name>**

execute a node in the terminal

# Lab Tasks

## Lab Task 1 – Parameters in ROS

In this task, you will revisit the custom teleoperation program you built previously and improve it by employing parameters. Create a package in the robot simulator workspace. Next, create the custom teleoperation program (you can refer to the previous lab if needed). Copy the keys\_to\_twist.py node, rename it to keytwist\_param.py and add its entries in the setup file. The keytwist\_param node subscribes to the detected keystrokes and publishes the twist messages. It will also parameterize the linear and angular velocities which are described below.

To add a parameter with a default value of 5, you can declare it in the node constructor with the following statement:

self.declare\_parameter('parameter\_name', 5.0)

The parameter will either have the default value or the value provided by the user (from the terminal). In either case, the parameter can be used in the subscriber callback after the following given statement:

paramVal = self.get\_parameter('parameter\_name').value

The paramVal will now be usable in the code after the above statement is executed. When running the node, the parameter value can be provided by the user by executing the following command in a terminal:

ros2 param set /keytwist\_param parameter\_name 3.5

If the above command is not given, the parameter will have the default value when the node is executing.

For this task, you need to add 2 parameters in the keystwist\_param node: linear\_speed (default value = 0.1) and angular\_speed (default value = 0.1). Then, run the custom teleoperation program (key\_pub and keytwist\_param nodes). For the submission, you will need to provide the class code for the keytwist\_param node and also to make a video called teleop\_param. The video must show the key\_pub terminal, a second terminal for entering parameters and the Gazebo simulation. You must use your program to first move the robot with the default values then use the second terminal to change the linear speed to 0.3 and angular speed to 5.5. Then move the robot again to show the changed speeds. All of this must be shown in a single video.

### KEYTWIST\_PARAM CLASS CODE STARTS HERE ###

*class* KeysToTwist(*Node*):

*def* \_\_init\_\_(*self*):

*super*().\_\_init\_\_("keys\_to\_twist")

*self*.subscription = *self*.create\_subscription(

            String, "keys", *self*.listener\_callback, 10

        )

*self*.publisher = *self*.create\_publisher(Twist, "cmd\_vel", 10)

*self*.declare\_parameter("linear\_speed", 0.1)

*self*.declare\_parameter("angular\_speed", 0.1)

*def* listener\_callback(*self*, *msg*):

        key = msg.data

*self*.get\_logger().info(*f*"Received: {key}")

        twist = Twist()

        if key == "w":

            twist.linear.x = *self*.get\_parameter("linear\_speed").value

        elif key == "s":

            twist.linear.x = -*self*.get\_parameter("linear\_speed").value

        elif key == "a":

            twist.angular.z = *self*.get\_parameter("angular\_speed").value

        elif key == "d":

            twist.angular.z = -*self*.get\_parameter("angular\_speed").value

        elif key == "q":

            twist.linear.x = 0.0

            twist.angular.z = 0.0

        else:

*self*.get\_logger().info(*f*"Unknown key: {key}")

*self*.publisher.publish(twist)

### KEYTWIST\_PARAM CLASS CODE ENDS HERE ###

## Lab Task 2 – Services

In this task, you will create a simple service. Start by creating another package in the robot workspace: **cpp\_package**

ros2 pkg create –-build-type ament\_cmake cpp\_package

Notice the ament\_cmake build type in the above command. Currently, ROS2 does not allow service definition in ament\_python packages so we use the ament\_cmake build type for the cpp\_package. The lab8 package is still the default package from where you will execute the service. This task can be divided into 3 parts:

1. Service definition file (.srv) in cpp\_package
2. Server node in lab8 package
3. Client node in lab8 package

**Part 1. Service Definition**

Create a folder named “srv” in the cpp\_package. Then, create a text file “AddTwo.srv” in the srv folder. This is the service definition file. In the AddTwo.srv file, place the following four lines:

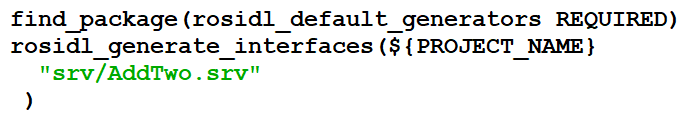
int64 a

int64 b

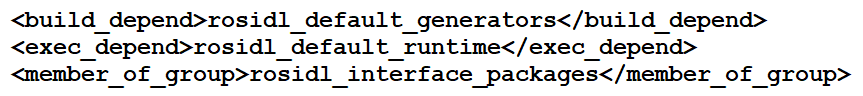
---

int64 sum

The a and b are the requests while the sum is the response. In the cpp\_package, you also need to make a few changes to the CMakeLists.txt and package.xml files. In the CMakeLists.txt, add the following lines after find\_package (<dependency> REQUIRED):



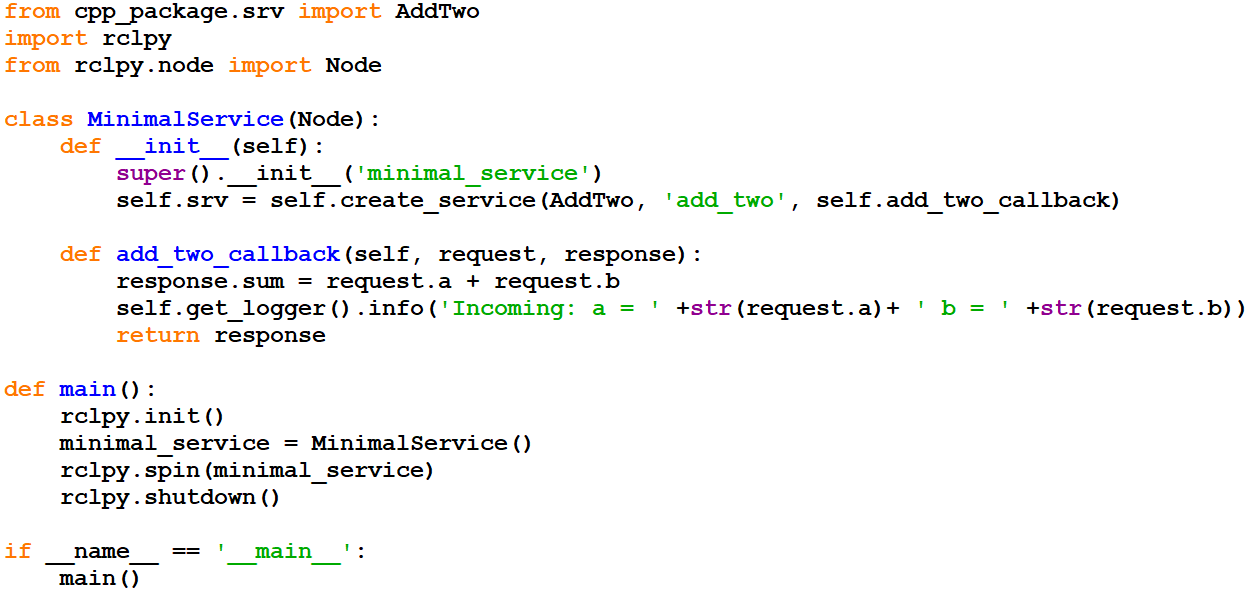
In the package.xml, add the following dependencies:

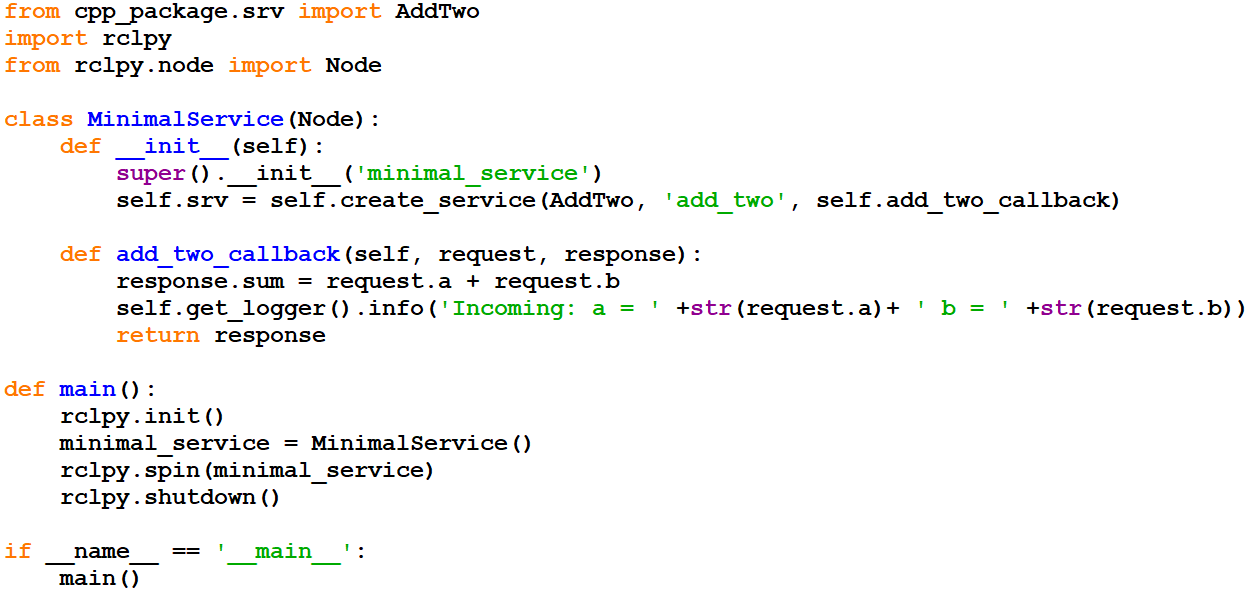


The service will now be defined (in the cpp\_package) and will be usable. We will implement the service nodes in the python package (you can use the custom teleop package or create a new one). In the python package, create 2 nodes (myServer.py and myClient.py) and add the node entries in the setup file.

**Part 2. Server Node**

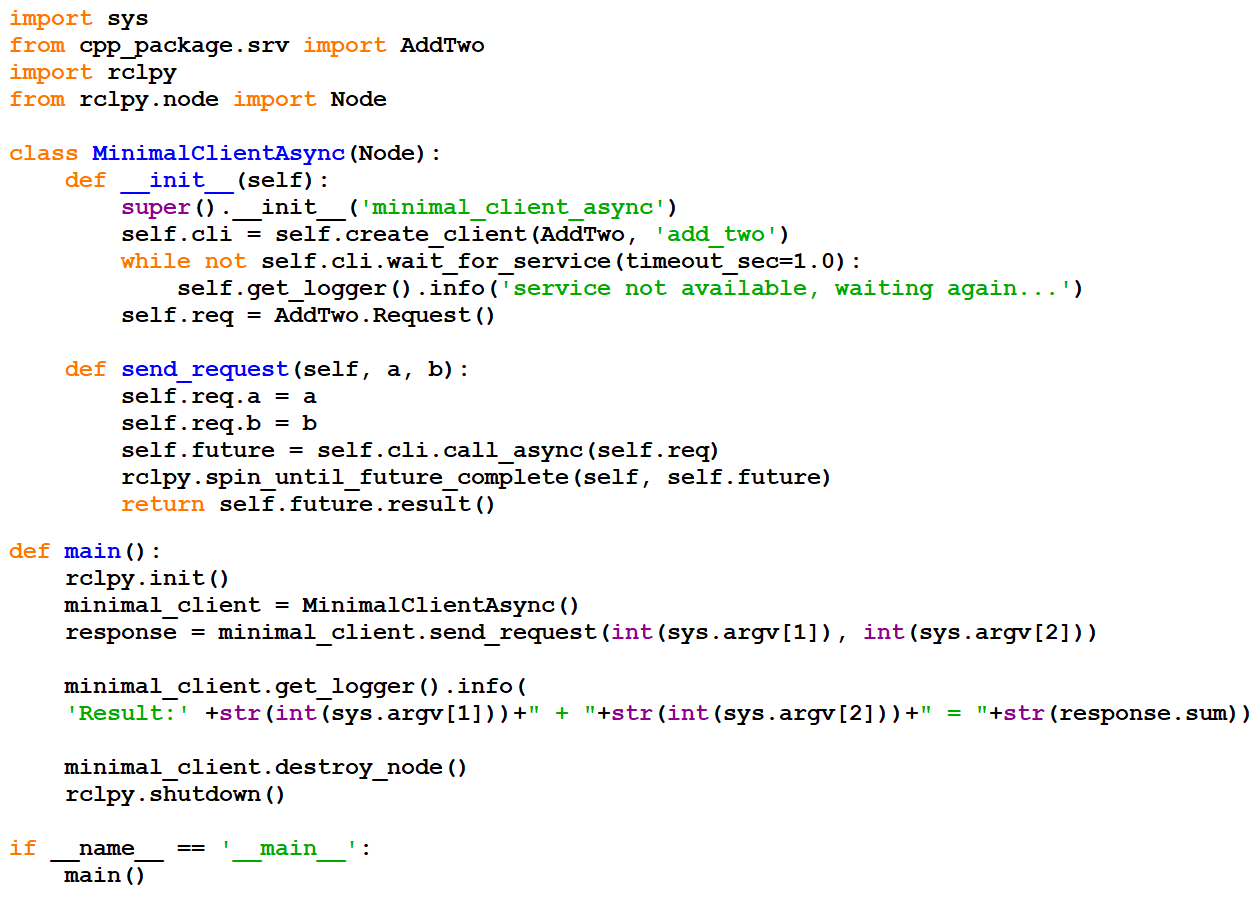
Place the following code in the server node:





**Part 3. Client Node**

Place the following code in the client node:

****

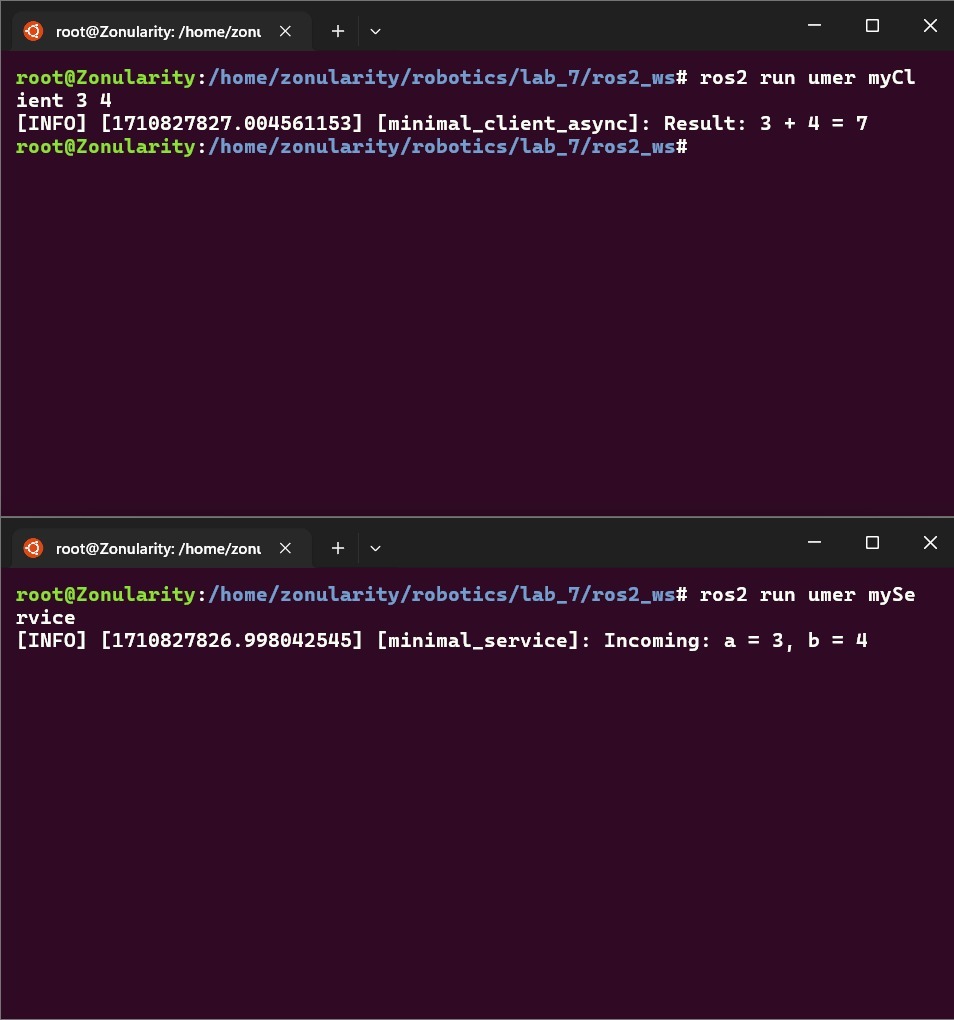
Next, execute the server and client nodes in separate terminals:

ros2 run lab8 myServer

ros2 run lab8 myClient 3 4

Experiment with the above the statements to understand how the service is working. For the task submission, provide a single screenshot showing both the client and server nodes. You must show at least 10 requests (and their consequent responses). Ensure your names are appearing on the terminal in the screenshot.

### SERVICE SCREENSHOT STARTS HERE ###



### SERVICE SCREENSHOT ENDS HERE ###

## Lab Task 3 – Modified Service

In this task, you will use your understanding from the previous task to create another service which will multiply 3 integers. The client will request for 3 integers and the response will be the product of those integers.

Create a service definition called “MultiplyThree.srv” in the cpp\_package. The definition will require 3 integers for the request and 1 integer for the response. Add this definition in the CMakeLists.txt file. (You do not need to use commas to separate the entries). Make copies of the server and client nodes (python package) and rename them to modifyServer and modifyClient. Add the node entries in the setup file.

Carefully observe the client and server codes from the previous task to understand how the 2 numbers were added. Then, apply your understanding in the new nodes to implement a service that will multiply 3 numbers. For the submission, provide the codes for both the client and server nodes. Also, provide a screenshot showing both terminals and at least 10 requests (and their consequent responses). Ensure your names are appearing on the terminal in the screenshot.

### MODIFIED SERVER CODE STARTS HERE ###

*class* ModMinimalService(*Node*):

*def* \_\_init\_\_(*self*):

*super*().\_\_init\_\_("mod\_minimal\_service")

*self*.srv = *self*.create\_service(

            MultiplyThree, "multiply\_three", *self*.multiply\_three\_callback

        )

*def* multiply\_three\_callback(*self*, *request*, *response*):

        response.product = request.a \* request.b \* request.c

*self*.get\_logger().info(

            "Incoming: a = %d, b = %d, c = %d" % (request.a, request.b, request.c)

        )

        return response

### MODIFIED SERVER CODE ENDS HERE ###

### MODIFIED CLIENT CODE STARTS HERE ###

*class* ModMinimalClientAsync(*Node*):

*def* \_\_init\_\_(*self*):

*super*().\_\_init\_\_("mod\_minimal\_client\_async")

*self*.cli = *self*.create\_client(MultiplyThree, "multiply\_three")

        while not *self*.cli.wait\_for\_service(*timeout\_sec*=1.0):

*self*.get\_logger().info("service not available, waiting again...")

*self*.req = MultiplyThree.Request()

*def* send\_request(*self*, *a*, *b*, *c*):

*self*.req.a = a

*self*.req.b = b

*self*.req.c = c

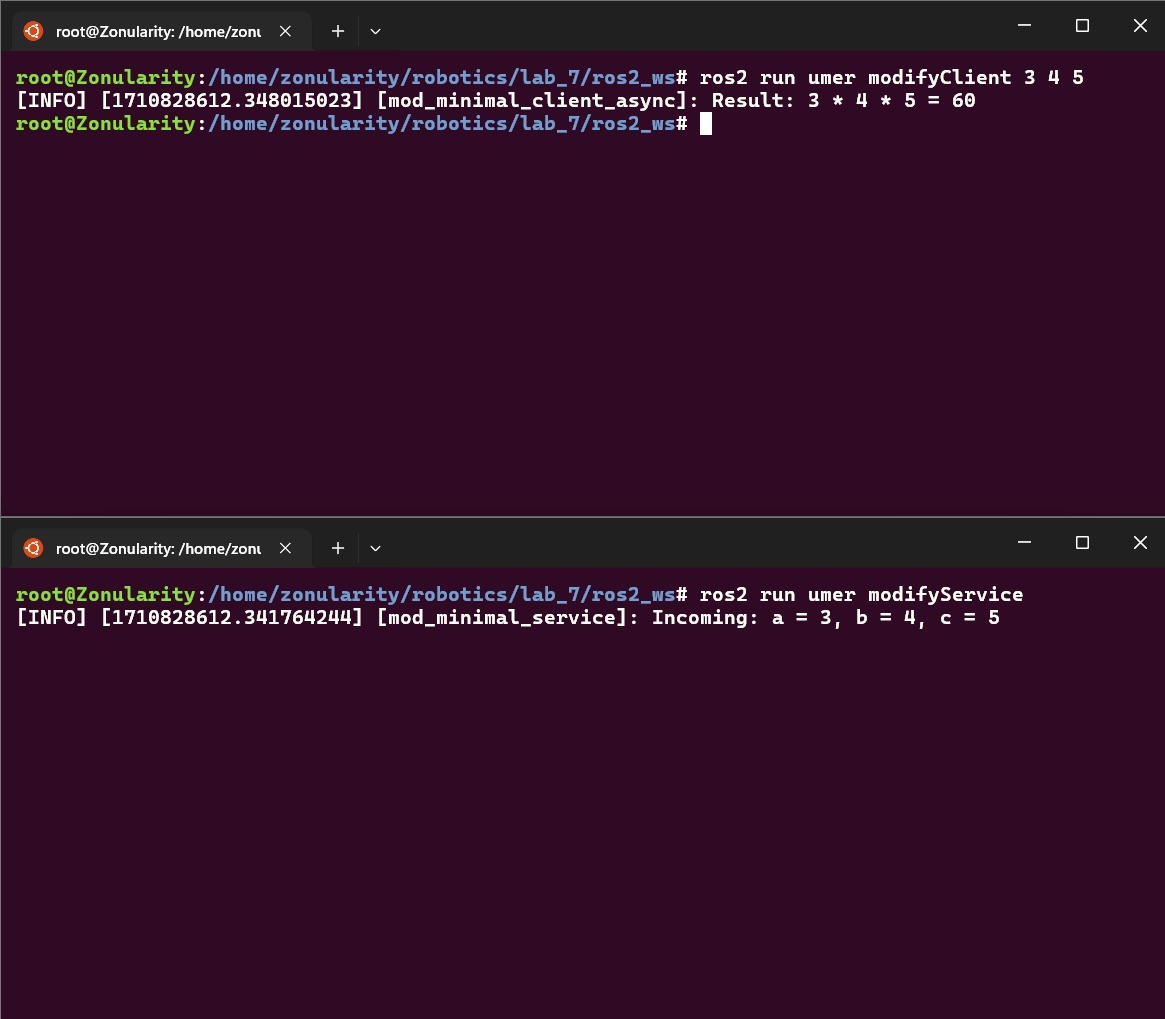
*self*.future = *self*.cli.call\_async(*self*.req)

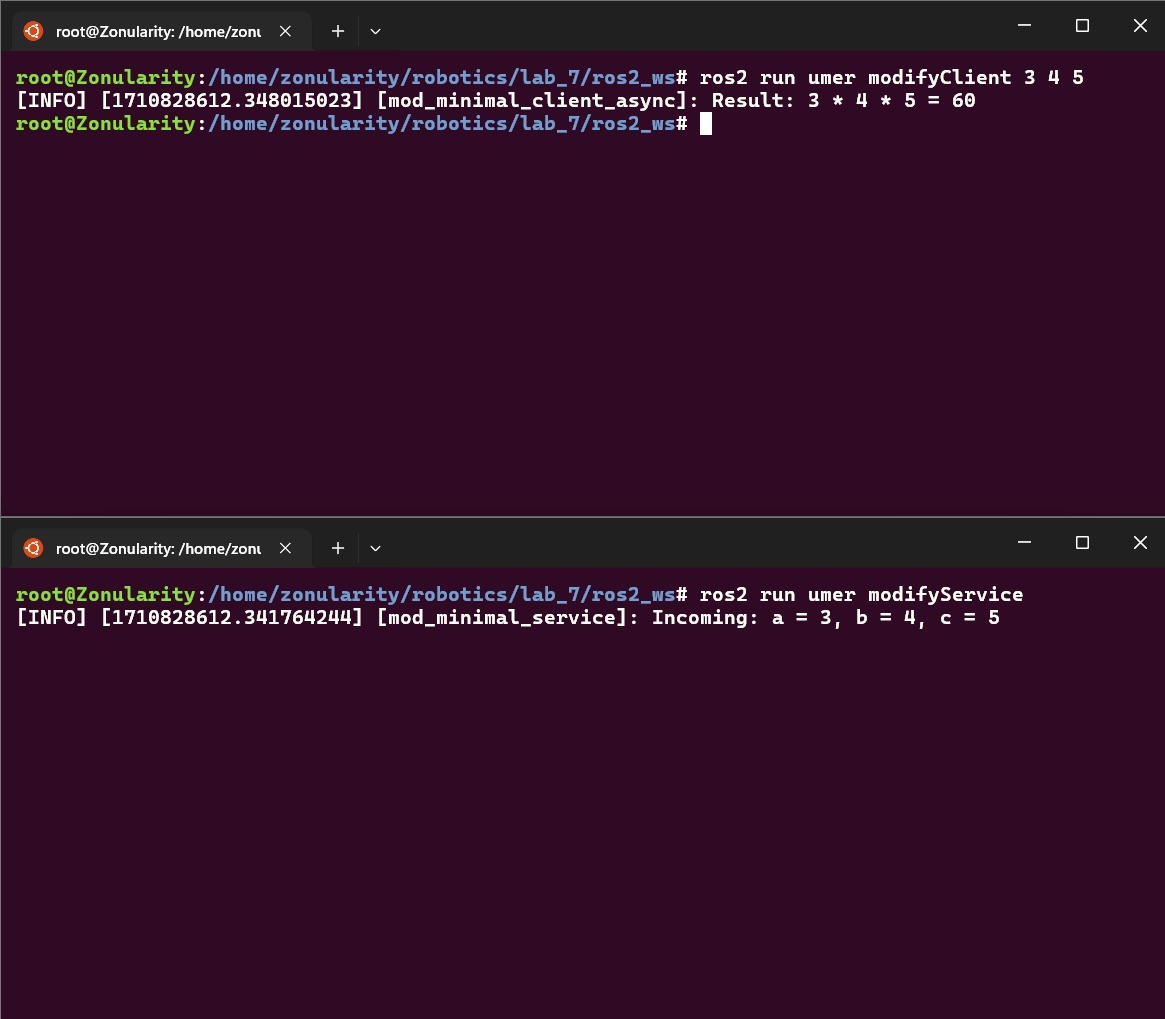
        rclpy.spin\_until\_future\_complete(*self*, *self*.future)

        return *self*.future.result()

### MODIFIED CLIENT CODE ENDS HERE ###

### MODIFIED SERVICE SCREENSHOT STARTS HERE ###





### MODIFIED SERVICE SCREENSHOT ENDS HERE ###