7MR10070: Software and Robotic Integration

Semester 2

Assignment 2

Written by Alexandros Megalemos

# Robotic Simulation

# Introduction

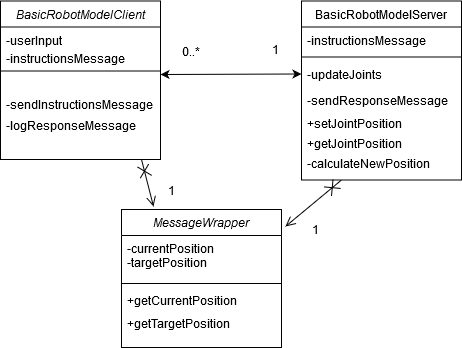


Figure 1: Pseudo UML of the relationship between Client and Server

*Figure 1: Pseudo UML of the relationship between Client and Server* shows a “pseudo” UML diagram of the system. Some of the functions were omitted for clarity. The basic idea of the system is to have a client-server interface, where the client sends commands to the server and if possible, the server executes them. While only one server should exist, we could theoretically multiple clients sending requests to the server. Both nodes use the MessageWrapper class which hides ugly parts of the message and provides utility methods for it.

## BasicRobotModelClient

In general, we don’t want the Client to include any logic (in regards to kinematics) and so it has no access to any actual logic in terms of calculating kinematics and estimating their validity. The only thing it can do it receive input from the user (i.e. using a terminal or a proper GUI) and pass (publish) it over to the Server. In order to be able to pass over more input, it needs to receive a reply from the Server, telling it that either: a) The input was valid and the robot has moved to the desired position or b) The input was invalid and it should request a different set of inputs from the user.

The client allows to user to select between two modes. The first allows the user to specify the joint angles directly and the second one allows the user to specify the end effector’s desired coordinates.

## BasicRobotModelServer

This is where the kinematics are calculated. It waits to receive a message that contains all the relevant info (*xyz* or *joint position*) from the Client, checks that the position is valid and then sends a request to the joint state publisher to move the robot. Once the robot has moved to desired position, it sends a message back to the Client to it know of the current status of the robot.

## Robot Model

The model is a simple robot comprised of a base, two joints and two links. Its .xml can be found in the Appendix under Figure 2: Robot URDF/XARCO

# Methods

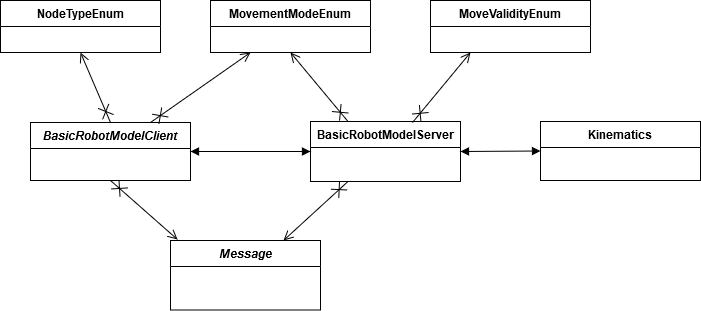


Figure 2: Pseudo UML of the whole system

## Client

The client/controller node is called **BasicRobotModelClient.py**. It contains a single class called **RobotModelClient** which itself is comprised of the **init()** function, the **start()** function, a **callback()** function and a **getInput()** function

#### Init()

* Initialises the node
* Initialises the publisher for a two way communication between client and server
* Creates an instance of the message wrapper

#### Start()

* Runs as long as the node is running (i.e. indefinitely)
* The main function where messages are presented to the user and input is requested

#### Callback()

* Simple callback method to log the instruction message

#### getInput()

* Performs checks to make sure that the input is inserted in a numerical format. If it’s not, it rejects it and asks the user to repeat

## Server

The server/subscriber node is called **BasicRobotModelServer.py**. It contains a single class called **RobotState** which itself is comprised of the following functions:

#### Init() and its sub-functions initJointState() and initPublishersAndSubscribers

* Initialises the node
* Initialises the joint state variables which contain the name of the joints and their positions
* Initialises the publishers and subscribers so that the server can communicate with the JointState and the Client
* Creates an instance of the message wrapper

#### setResponseMessage()

* Sets the response message to send back to client

#### Start()

* Initiates the program
* Calls the updateJoints() function
* Calls the publishMessage() function

#### Callback()

* Retrieves the new position of the joints
* Calls the setResponseMessage() function

#### setJointStatePosition()

* Sets the new position of the joints

#### getJointStatePosition()

* Calculates the new position of the joints using the Kinematics class

#### publishMessage()

* Helper function to publish the message to the subscriber

#### updateJoints()

* Publishes the new positions to the JointStatePublisher node

#### isValidPosition()

* Checks if the requested position is valid

#### setMessage()

* Sets the attributes for the message wrapper class

## Message Wrapper

The message wrapper class is called **Message.py**. It used in order to hide some functions from the server and client nodes. It acts as a bean-like class with setters and getters to satisfy the fields provided by the **InstructionsMessage.message** file. Furthermore, it contains some utility methods such as:

#### clearMessage()

* Clears all fields of the message

#### isUsingJointPosition()

* Checks if the joint position mode is active by calling the **MovementModeEnum** class

#### isUsingEndEffectorPosition()

* Checks if the end effector mode is active by calling the **MovementModeEnum** class

## Enums

Various enums are used to remove hard coded strings and hide some logic from the client and server nodes.

### MoveValidityEnum

Simple enum that contains 3 fields. One to specify that a valid position was requested, one to specify that an invalid position was requested and a not available position in case it is not available yet.

Furthermore, it contains:

#### isValid()

* Utility function to quickly check if the position is set to valid

#### isInvalid()

* Utility function to quickly check if the position is set to invalid

### NodeTypeEnum

Simple enum that contains node names that would otherwise be hard coded. These fields could have been included as constants in their respective .py files, but this implementation removes unnecessary clutter from those files.

### MovementModeEnum

Enum to signify the mode selected by the user. It contains 3 fields. One to specify that node mode has been set yet, one to signify that the joint position mode was selected and one to signify that the end effector mode was selected.

Furthermore, it contains:

#### isJointPosition()

* Utility function to quickly check if the joint position mode is set

#### isEndEffectorPosition()

* Utility function to quickly check if the end effector position mode is set

It also contains a **Utils** class to help get the enum entry through a label or id. This is useful when translating between the server/client response.

## Kinematics

The kinematics class contains methods to calculate the kinematics required based on the user’s requests. It is used to separate logic that is not needed to be seen by the server.

#### getForwardKinematics()

* Calculates the forward kinematics

#### getInverseKinematics()

* Calculates the inverse kinematics

#### isValidHeight()

* Checks if the requested position is valid based on the robot’s height

# Validation

summarizing how the server-client interface was tested. Clear tests for both valid and invalid robot positions should be described with corresponding figures where appropriate.

The interface was tested by manual testing.

## Input Validation

### Valid Input

### Invalid Input

## Position Validation

### Valid Position

### Invalid Position

# Code / Git Repository

<https://github.com/Meldanen/kcl/tree/master/robotics/ros/assignment_ws>

# Appendix



Figure 3: Robot URDF/XARCO

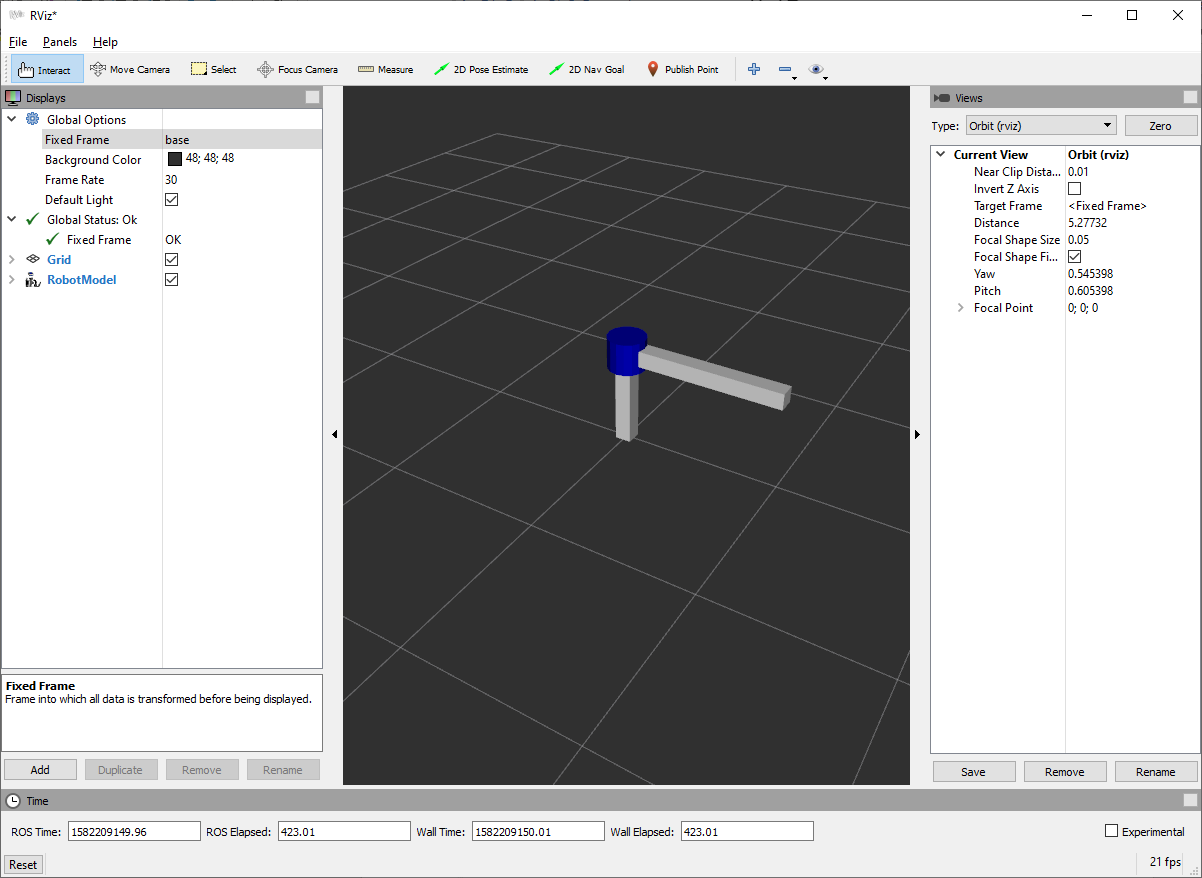


Figure 4: Robot in RVIZ

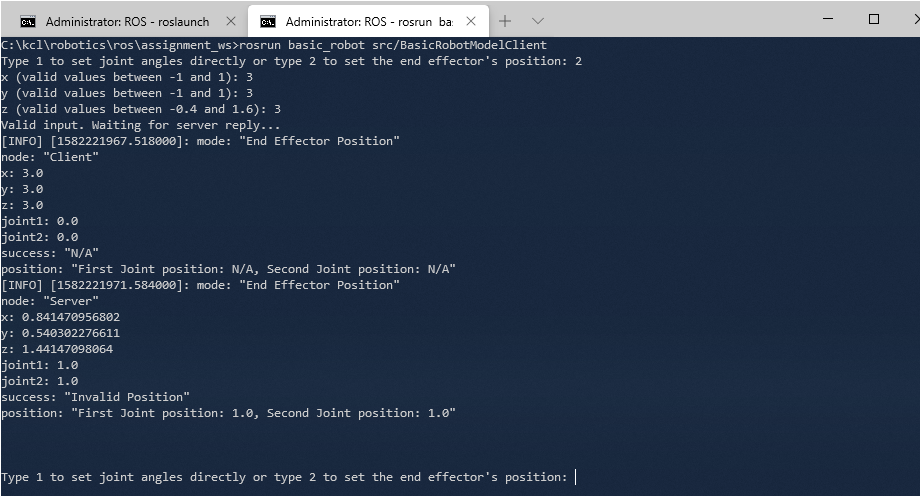


Figure 5: Request invalid position (end effector mode)

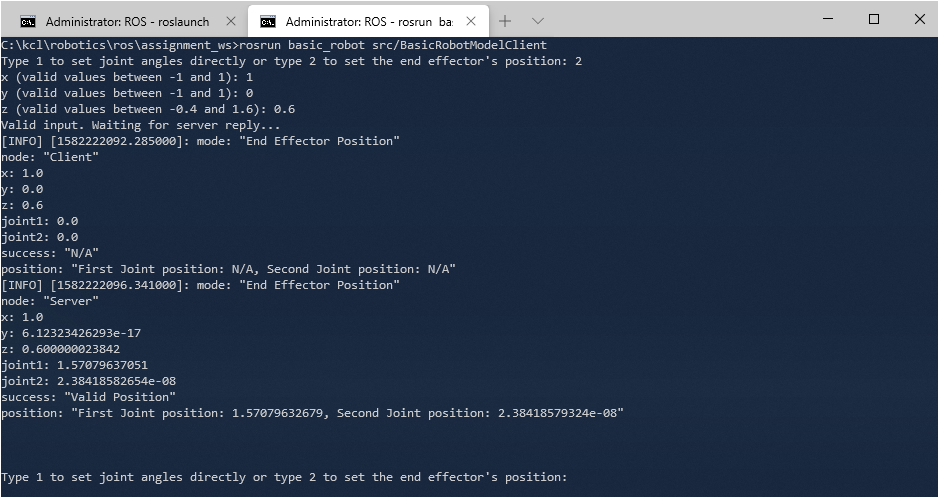


Figure 6: Request valid position (end effector mode)

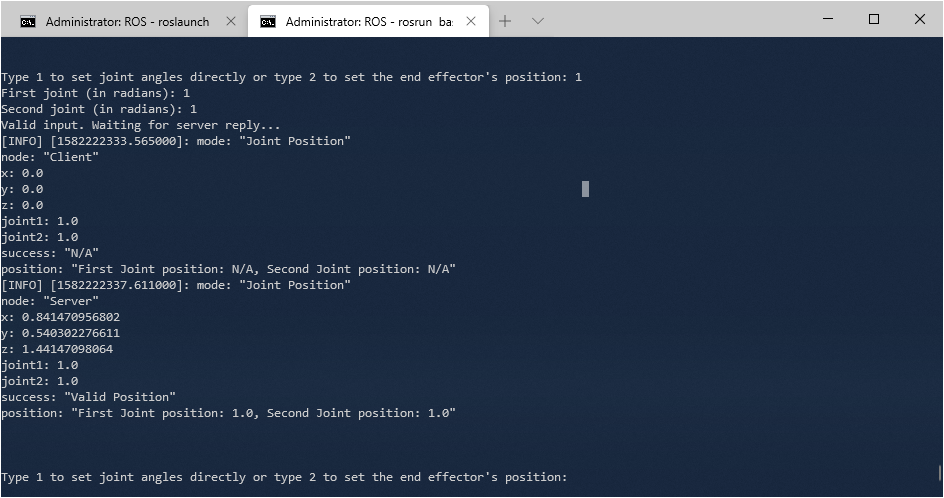


Figure 7: Request valid position (Joint position mode)

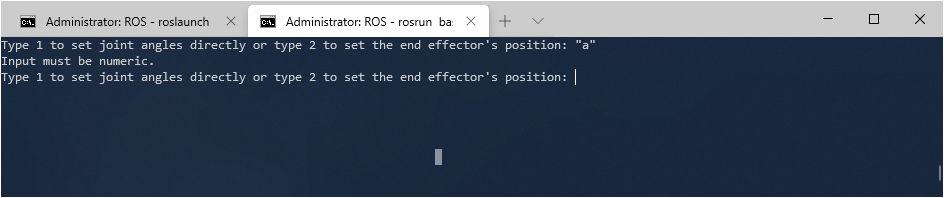


Figure 8: Request non-numeric node (same check used for all inputs)

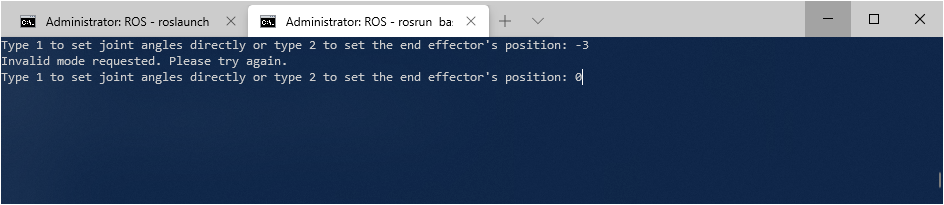


Figure 9: Request invalid mode