A Mobile Application to Remotely Control Humanoid Robots

Henrique Lima Cará de Oliveira *Esther Luna Colombini*

Relatório Técnico - IC-PFG-16-12 - Projeto Final de Graduação

December - 2016 - Dezembro

A Mobile Application to Remotely Control Humanoid Robots

Henrique Lima Cará de Oliveira[*]

Esther Luna Colombini[†]

1 Abstract

In this work, we developed an easy-to-use Android OS remote controller mobile application for NAO - an autonomous, programmable humanoid robot developed by Aldebaran Robotics - following the principles of simplicity and design constraint [1]. The remote controller allows the user to send some basic commands to a robotic device, and it also keeps the user informed about the positional aspects of the robot in relation to its environment by displaying to the user a 3D model of the robot, which mimics the current pose of the robot. The app also displays the images obtained from the vision sensor of the robot. We developed an app structure that encapsulates some predefined actions to be executed on the robot, and we also implemented a high level protocol for the communication between the controller and the robot. By using this protocol, we can easily extend those actions to control a real bipedal robot, commanding it to execute predefined actions in an exhibition context of robot soccer. Additionally, we have also implemented all the server-side interfaces that allow us controlling a simulated robot. All tests in this work were performed in the robotics simulated environment V-REP.

2 Introduction and revised bibliography

In this work, as for the controlled robot device, we have chosen to use NAO due to its well documented use in related projects. However, as the project was conducted, it is possible to control any kind of humanoid robot that is capable of performing the predefined actions by following the logical protocol established, and also has compatible API functions. Therefore, it would be necessary to rewrite the API function caller code (client for the robot's API).

Along the app development, we have chosen to work with an open source Java development framework called libgdx [2]. Libgdx has proved to be an excellent tool regarding its capability of providing many resources and a high level of customization, along with the facilitated portability to various operational systems (Android and iOS, for example). It

[*]Unicamp

[†]Institute of Computing, University of Campinas, 13081-970 Campinas, SP.

also has demonstrated good execution performance when tested in different devices. However, as we have noted, working with libgdx requires some low-level graphic detailed control and knowledge, what relatively requires more develop time in comparison to other available commercial solutions.

We used GIMP (GNU Image Manipulation Program) [3] to create the 2D resources. For the robot 3D model (Nao 3D model), we first obtained a non-textured model of NAO from SoftBank Robotics Community [4] and, then, we used Blender [5] - a free and open source 3D creation suite - for texturing and rigging the model. Also, during the rigging process, we relied on NAO Software documentation [6] for obtaining good estimations of the positions of the joints in the model.

The app graphics consists of 2D resources - that is, screens, buttons and textures in general - and also a 3D model of the robot, which is displayed in the Controller Screen. In the operational context of the app, the model mimics the current pose of the real/simulated robot, and it may also be used as a controller interface, i.e., the user can directly perform control actions by controlling its joints individually. All simulations were performed using V-REP (Platform Virtual Robot Experimentation) [7] and Choregraphe Suite. The communication interfaces between the local server (for the app) and Choregraphe Suite (NAO qi API [8]), and between the local server and V-REP server API were based on Pierre Jacquot's work "NAO Control Project" [9].

3 Justification

On robot soccer, usually the robots do not rely on direct human control over the robot's actions. However, the same robots are usually presented to the general public. In this sense, allowing them to directly interact with one of this robots through an easy-to-use remote controller, seems like a good way of increasing the general interest of those in the competitions and, moreover, in the field of robotics. In this sense, the main motivation of this work is to create a tool for remote controlling a humanoid robot that can help promoting science communication for the general public.

4 Objectives

This work aims to develop a mobile app for Android OS, which allows anyone to control a real or a simulated bipedal robot with 25 degrees of freedom (DOF), commanding it to execute predefined actions in the context of robot soccer.

5 Work development

5.1 General architecture and protocol

The system built in this work is composed of four main components: the mobile app, the local server, the NAO qi server and the V-REP server.

The communication between the mobile app and the local server is done via TCP stream sockets and the others - between the local server and the NAO qi server (NAO qi), and between the local server and the V-REP server - are done via Java API methods calls.

All of the commands that the mobile app may send to the local server are better understood when thinking about use cases - connect, get joints angles and vision sensor image, walk/turn, go to posture and change joints angles - with their own specifications.

In order to allow the controller to communicate to simulated or real robots, a logical communication protocol was defined, as described in Figure 1.

Also, all the communications commands/responses between the Mobile app and the local server have default timeout of 5 seconds. Then, whenever the waiting entity runs out of time, it causes the app to disconnect from the local server and to go back to the connection screen; and the local server to close the current connection socket and to wait for new incoming connection requests. We choose this approach in order to reduce inconsistencies.

5.2 Use cases

5.2.1 Connect

The basic flow of the Connect use case is presented next. Please refer to Figure 2 to the equivalent sequence diagram.

1. The mobile app sends a connection request to the local server with the server's ip address and port number; the server accepts the request and establishes a connection channel with the client. In Figure 2, the connection request call made by the client is an abstraction of this process.

2. The local server sends a connection request to NAO qi, with NAO qi's ip address and port number, then, NAO qi returns a reference to its application object. In Figure 2, the connection request call made by the local server to NAO qi is an abstraction of this process.

3. Using NAO qi's reference, the local server tries to send the command start to NAO qi, which, if succeeded, initializes NAO qi framework and creates a session. A Session is what allows the server to connect NAO qi's services together locally or over the network.

4. Now, if the session was successfully created, the local server sends a connection request to the V-REP server, with V-REP server's ip address, port number and some others connections options; if the request was successfully accepted, the V-REP server API returns an integer value - the client ID - different than -1. In Figure 2, the connection request call made by the local server to the V-REP server is an abstraction of this process.

5. Next, the local server needs to get the V-REP objects handles for each NAO simulated robot's children objects - its joints -, plus, the objects handles for NAO simulated robot itself and its head vision sensor. In Figure 2, the call get objects handles(clientID) is an abstraction of this process.

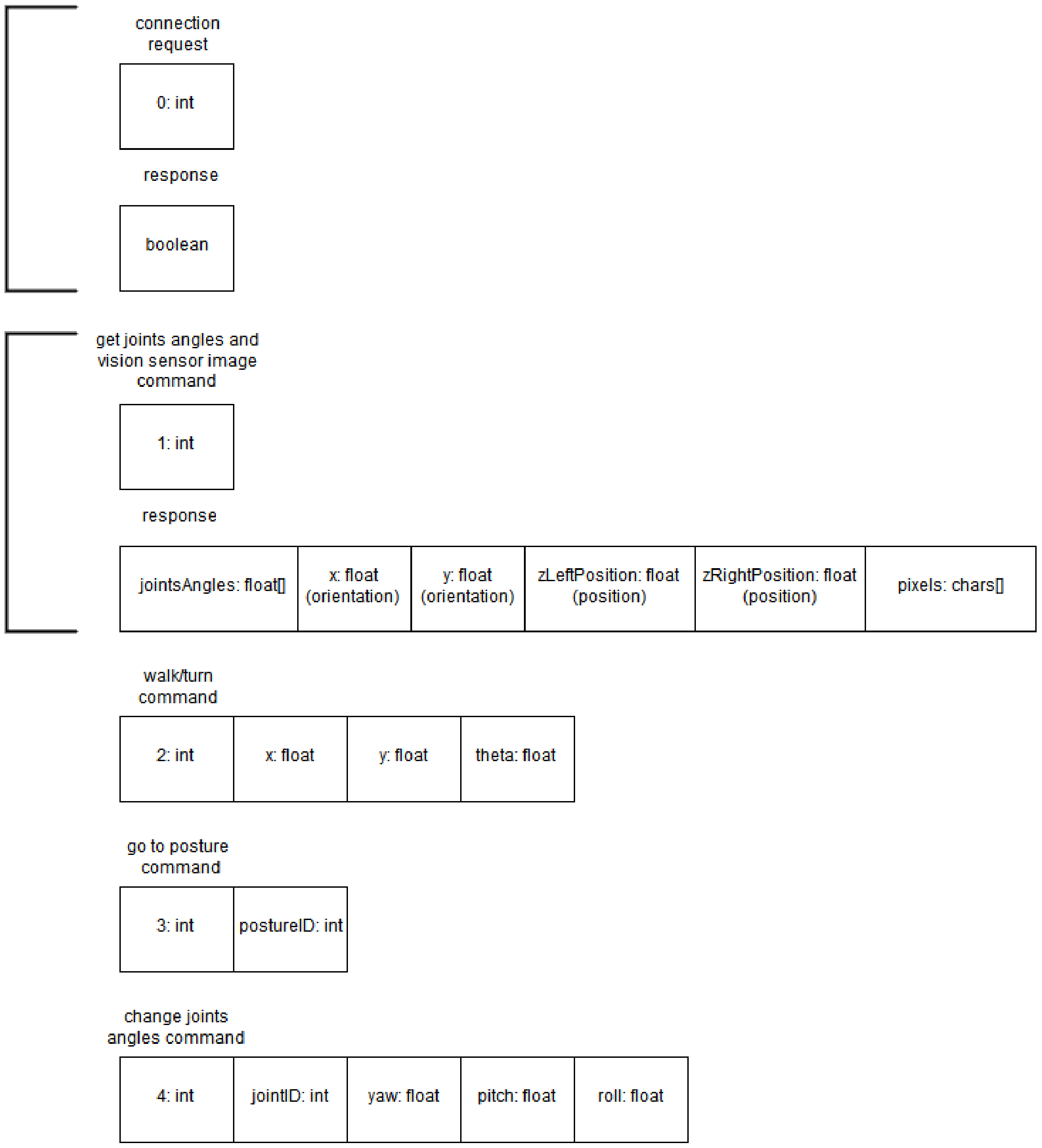


Figure 1: Protocol

6. Next, the local server enables the streaming operation mode for each command and object handle the command applies to; this can be seen as a command/message subscription from our local server to the V-REP server, where the V-REP server will be streaming the data to our local server. The following pair of commands and objects' handles are set to streaming operation mode:

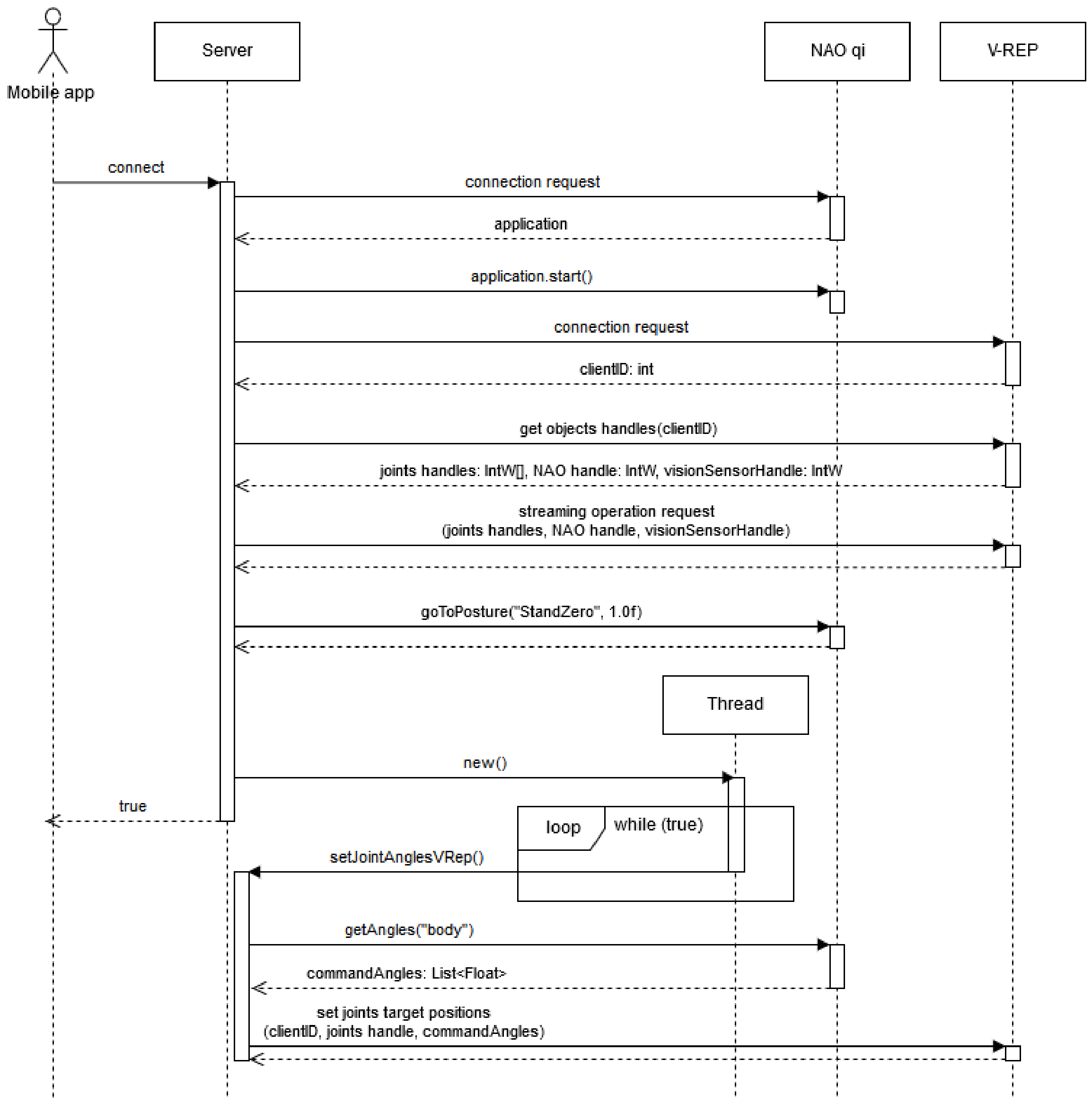

Mobile app
Server
NAO qi
V-REP
connect
connection request
application
application.start()
connection request
clientID: int
get objects handles(clientID)
joints handles: IntW[], NAO handle: IntW, visionSensorHandle: IntW
streaming operation request
(joints handles, NAO handle, visionSensorHandle)
goToPosture("StandZero", 1.0f)
Thread
new()
true
loop
while (true)
setJointAnglesVRep()
getAngles("body")
commandAngles: List<Float>
set joints target positions
(clientID, joints handle, commandAngles)

Figure 2: Connect sequence diagram

- simxGetJointPosition for each joint of NAO,
- simxGetObjectOrientation for NAO model,
- simxGetObjectPosition for both pelvis joints of NAO (later used to calculate NAO's z-axis position relative to the ground), and
- simxGetVisionSensorImage for NAO's head vision sensor.

All of those operations are non-blocking (Figure 3). In Figure 2, the call streaming operation request(clientID, body, visionSensorHandle) is an abstraction of this

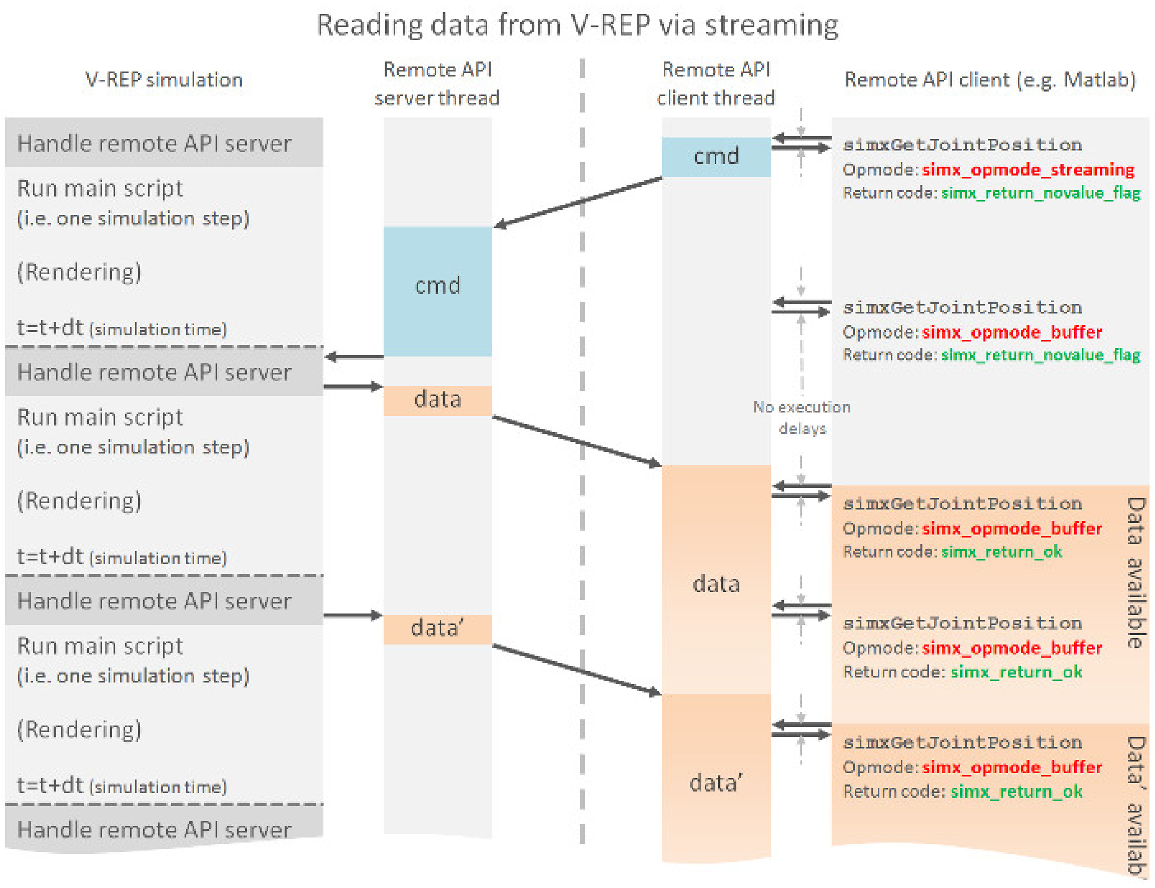

Reading data from V-REP via streaming
V-REP simulation
Remote API server thread
Remote API client thread
Remote API client (e.g. Matlab)
Handle remote API server
Run main script (i.e. one simulation step)
(Rendering)
t=t+dt (simulation time)
cmd
data
data'
simxGetJointPosition
Opmode: simx_opmode_streaming
Return code: simx_return_novalue_flag
Opmode: simx_opmode_buffer
Return code: simx_return_ok
No execution delays
Data available
Data' available

Figure 3: Data streaming (obtained from [10])

process.

7. The local server tries to send the blocking-call command goToPosture("StandZero", 1.0f) to NAO qi, which, if succeeded, makes the NAO qi simulated robot go to the predefined posture asked in the first parameter, i. e., "StandZero", which is the initial position of the V-REP simulated NAO. The second parameter of this method is the relative speed of the movement whose value is in the range between 0.0 to 1.0.

8. Now, at this point, the local server creates a new thread that is responsible for updating V-REP simulated NAO's joints angles, and, finally, returns true to the client. About the newly-created thread (synchronization thread), it runs repeatedly calls to the server thread-safe private method setJointAnglesVRep(); at each execution, this method tries to send the command getAngles("Body", false) to NAO qi, which, if succeeded, returns a list of joints angles of NAO qi simulated robot. Then, the local server calls the method simxSetJointTargetPosition for each joint of NAO, setting its angle - of the V-REP simulated NAO joint - to the same value of the respective joint in the previous returned list; this way, both simulated robots - nao qi's and V-REP's

- shall express almost synchronized poses.

5.2.2 Get joints angles and vision sensor image

The basic flow of get joints angles and vision sensor image use case follows below. Please refer to Figure 4.

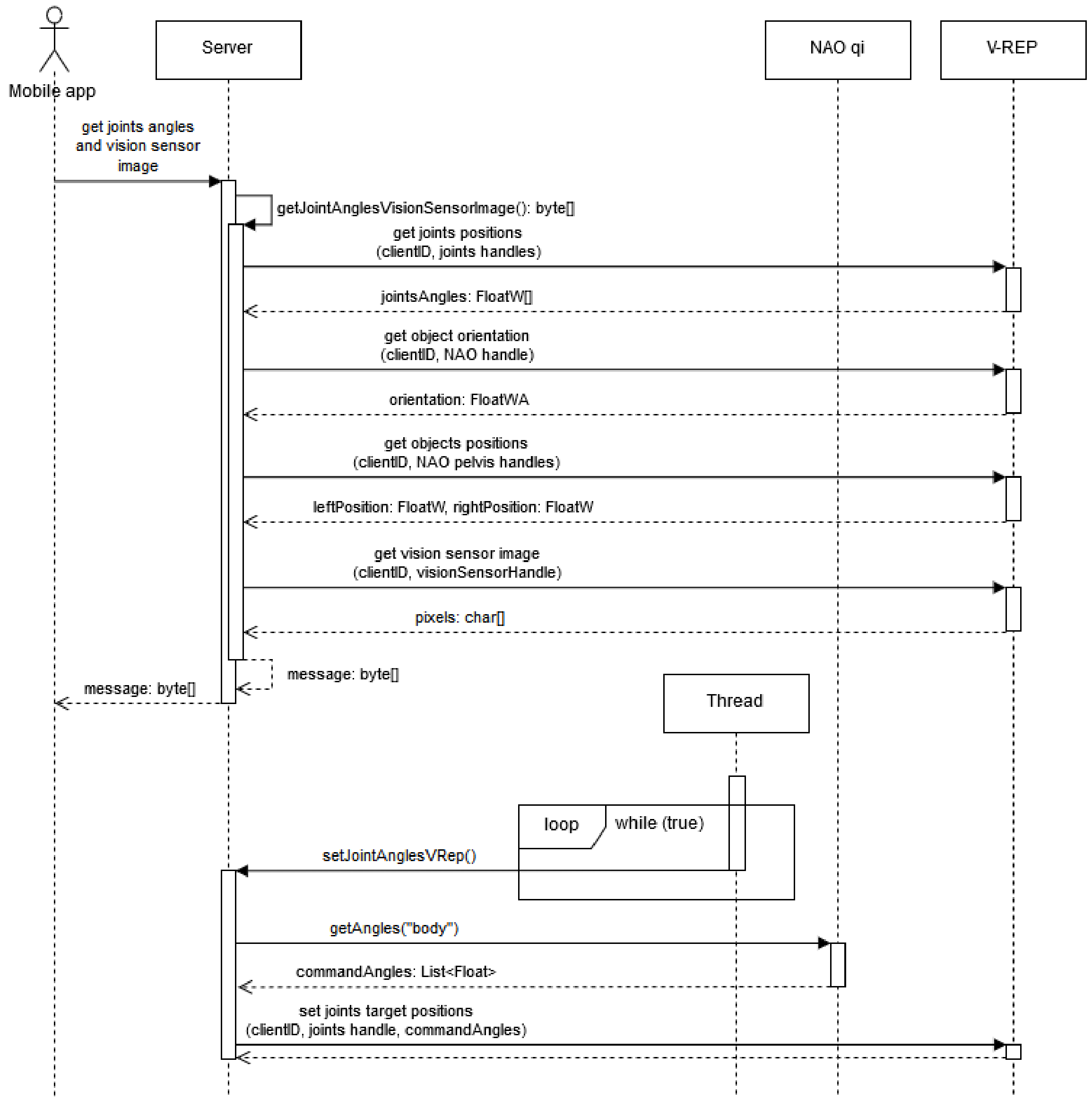

Mobile app
Server
NAO qi
V-REP
get joints angles and vision sensor image
getJointAnglesVisionSensorImage(): byte[]
get joints positions (clientID, joints handles)
jointsAngles: FloatW[]
get object orientation (clientID, NAO handle)
orientation: FloatWA
get objects positions (clientID, NAO pelvis handles)
leftPosition: FloatW, rightPosition: FloatW
get vision sensor image (clientID, visionSensorHandle)
pixels: char[]
message: byte[]
message: byte[]
Thread
loop
while (true)
setJointAnglesVRep()
getAngles("body")
commandAngles: List<Float>
set joints target positions (clientID, joints handle, commandAngles)

Figure 4: Get joints angles and vision sensor image sequence diagram

1. The mobile app sends the command get joints angles and vision sensor image to the

local server, then, the local server calls its thread-safe private method getJointsAnglesVisionSensorImage.

2. In this method, the local server needs to get the angles of the V-REP simulated NAO joints, for each one of then, by individual calls to the V-REP server API method simxGetJointPosition. Each one of these calls are made using the buffer operation mode; i. e., a non-blocking mode where no command is sent to the local server, but just a reply to a same command, previously executed, is returned from a local buffer, if available. This mode is often used in conjunction with the simx_opmode_streaming (Figure 3). That being done, the local server obtains an updated array of the joints angles. In Figure 4, the call get joints positions(clientID, joints handles) is an abstraction of this process.

3. Next, the local server needs to get the orientation of the V-REP simulated NAO by calling the V-REP server API method simxGetObjectOrientation, passing NAO object handle as parameter. The local server obtains the updated values of yaw, pitch and roll values of the robot relative to the scene's coordinate system. In Figure 4, the call get object orientation(clientID, NAO handle) is an abstraction of this process.

4. Now, at this point, the local server needs to get the z-position of the V-REP simulated NAO by individually calling the V-REP server API method simxGetObjectPosition, passing NAO's pelvis objects handles as parameters. The local server obtains the float values zLeftPosition and zRightPosition. In Figure 4, the call get objects positions (clientID, NAO pelvis handles) is an abstraction of this process.

5. Next, the local server needs to get the array of values representing the pixels of the V-REP simulated NAO head vision sensor image by calling the V-REP server API method simxGetVisionSensorImage, passing visionSensorHandle as parameter. The local server obtains an array of chars whose elements represents the image pixels. In Figure 4, the call get vision sensor image(clientID, visionSensorHandle) is an abstraction of this process.

6. Finally, the local server first reduces the size of the pixels array - from 640x480 to 160x120 - in order to optimize the overall communication speed - and then, following the communication protocol specification (Figure 1), puts all of those updated data together in an response array of bytes - message -, returns it to the caller and then to the Mobile app.

5.2.3 Walk/turn

The basic flow of walk/turn use case follows below. Please refer to Figure 5 for the sequence diagram.

1. The mobile app sends the command walk/turn(x, y, theta) to the server with the velocities parameters along X-axis, along Y-axis, both in meters per second, and around Z-axis, in radians per second; the local server checks if NAO qi simulated

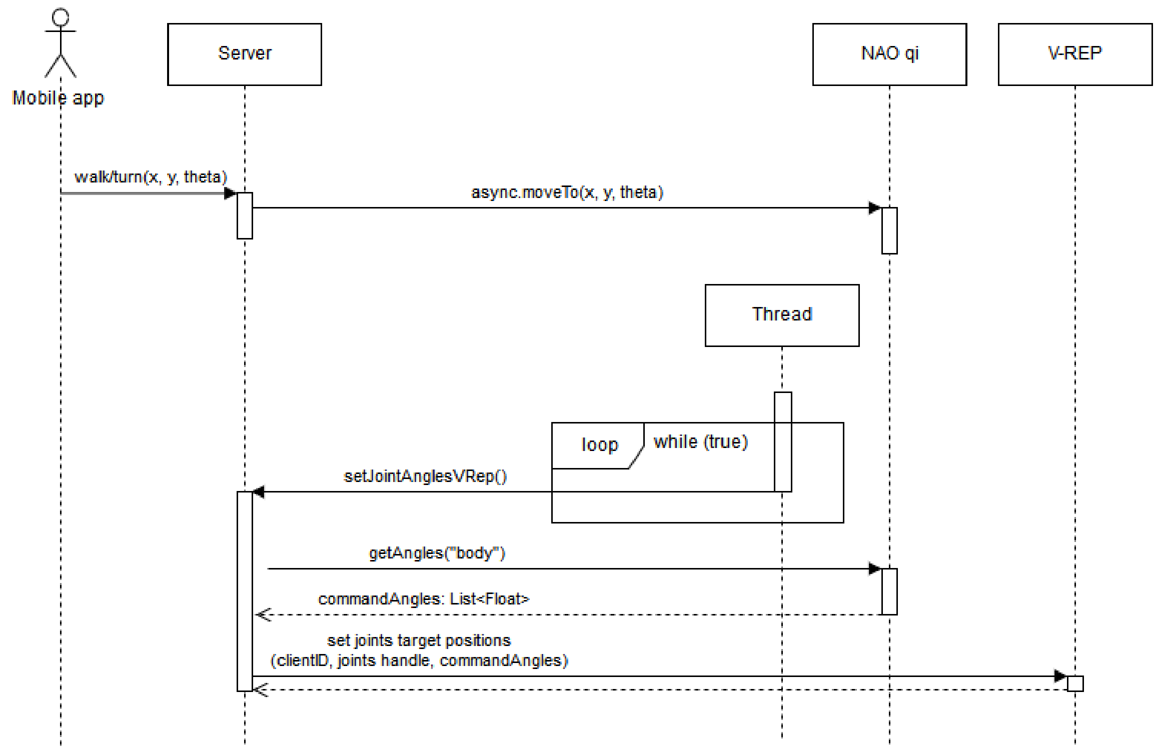

Mobile app
Server
NAO qi
V-REP
walk/turn(x, y, theta)
async.moveTo(x, y, theta)
Thread
loop
while (true)
setJointAnglesVRep()
getAngles("body")
commandAngles: List<Float>
set joints target positions
(clientID, joints handle, commandAngles)

Figure 5: Walk/turn sequence diagram

robot is moving, and, if is not, it tries to call the NAO qi method moveTo in a non-blocking way - async -, passing x, y and theta as parameters.

2. If succeeded, NAO qi asynchronously executes the operation by using its threadpool.

5.2.4 Go to posture

The basic flow of go to posture use case follows below. Please refer to Figure 6 for the corresponding sequence diagram.

1. The Mobile app sends the command go to posture(postureID) to the local server with the predefined posture id as parameter; the local server stops any current move action and tries to call the NAO qi method goToPosture in a non-blocking way - async -, passing posture name as parameter.

2. If succeeded, NAO qi asynchronously executes the operation by using its threadpool.

5.2.5 Change joints angles

The basic flow of change joints angles use case follows below, which corresponds to the diagram presented in Figure 7.

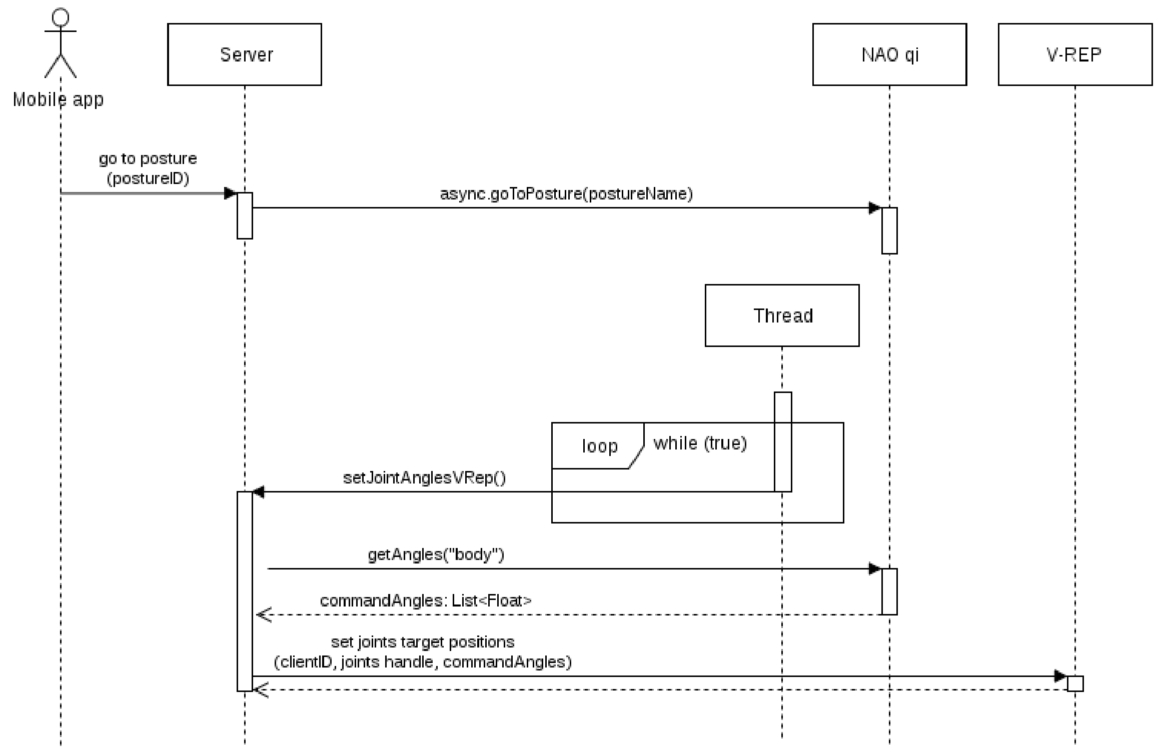

Mobile app
Server
NAO qi
V-REP
go to posture (postureID)
async.goToPosture(postureName)
Thread
loop
while (true)
setJointAnglesVRep()
getAngles("body")
commandAngles: List<Float>
set joints target positions (clientID, joints handle, commandAngles)

Figure 6: Go to posture sequence diagram

1. The mobile app sends the command change joints angles(jointID, yaw, pitch, roll) to the server, then, the local server calls its thread-safe private method changeJoints Angles(jointID, yaw, pitch, roll).

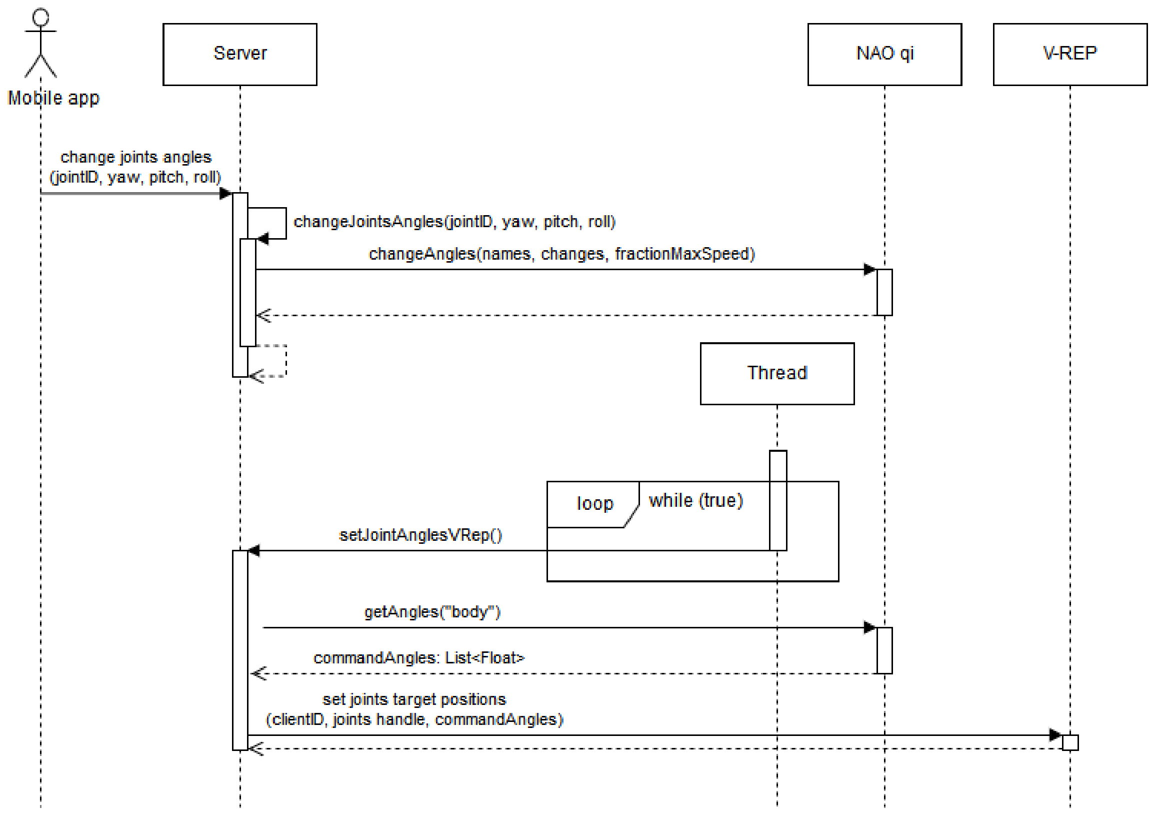
2. In this method, the local server needs to set the angles of the NAO qi simulated robot joint specified by the jointID by calling the NAO qi method changeAngles (names, changes, fractionMaxSpeed), where names are the joint's yaw, pitch and/or roll ids, changes are the yaw, pitch and/or roll angles increases (postive) or decreases (negative) in radians and fractionMaxSpeed is the fraction of the maximum speed to use. If succeeded, the method returns.

If the real robot is in place, a direct link between the app and the robot would be established, with no need for invoking Nao qi methods.

5.3 App architecture

5.3.1 Basic concepts

All of the following concepts were implemented on Android OS, using libgdx as our development framework. Libgdx is a cross-platform game and visualization development framework that currently supports Windows, Linux, Mac OS X, Android, Blackberry, iOS, and HTML5



Mobile app
Server
NAO qi
V-REP
change joints angles (jointID, yaw, pitch, roll)
changeJointsAngles(jointID, yaw, pitch, roll)
changeAngles(names, changes, fractionMaxSpeed)
Thread
loop
while (true)
setJointAnglesVRep()
getAngles("body")
commandAngles: List<Float>
set joints target positions (clientID, joints handle, commandAngles)

Figure 7: Change joints angles sequence diagram

as target platforms. Libgdx also goes low-level, giving direct access to file systems, input devices, audio devices and OpenGL via a unified OpenGL ES 2.0 and 3.0 interface [11].

Our app uses the screen structure model of libgdx, in which each of the screens encapsulates its own graphic and logic.

In the screen lifecycle (Figure 8), the method show() is called when the screen is set as the current one, while hide() is called on changing screens; resize() is called right after show() is called, and right after the screen size/orientation changes. The methods resume() and pause() are set as callbacks of Android's respective methods, and the method dispose() should be explicitly called; then, we are responsible for disposing the disposable resources after using then, except the ones managed by the asset manager [12]. Finally, the rendering thread of libgdx calls the render() method of the current screen continuously, with a frequency that depends on you the hardware.

5.3.2 Loading screen

In this screen (Figure 9), all of the packed textures - loading.pack, connection.pack and controller.pack - [13], fonts and 3D models are loaded by the asset manager.

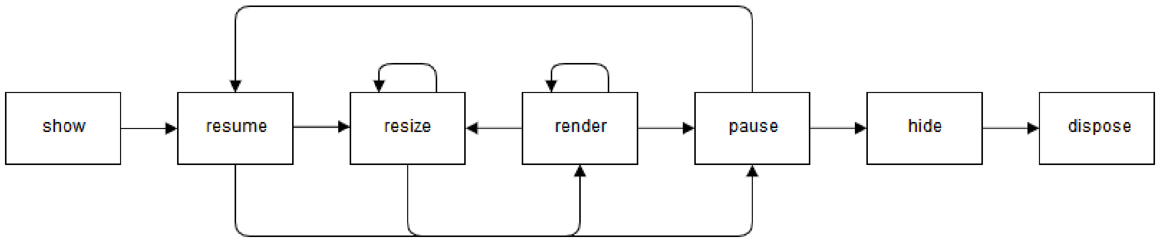


Figure 8: Screen lifecycle

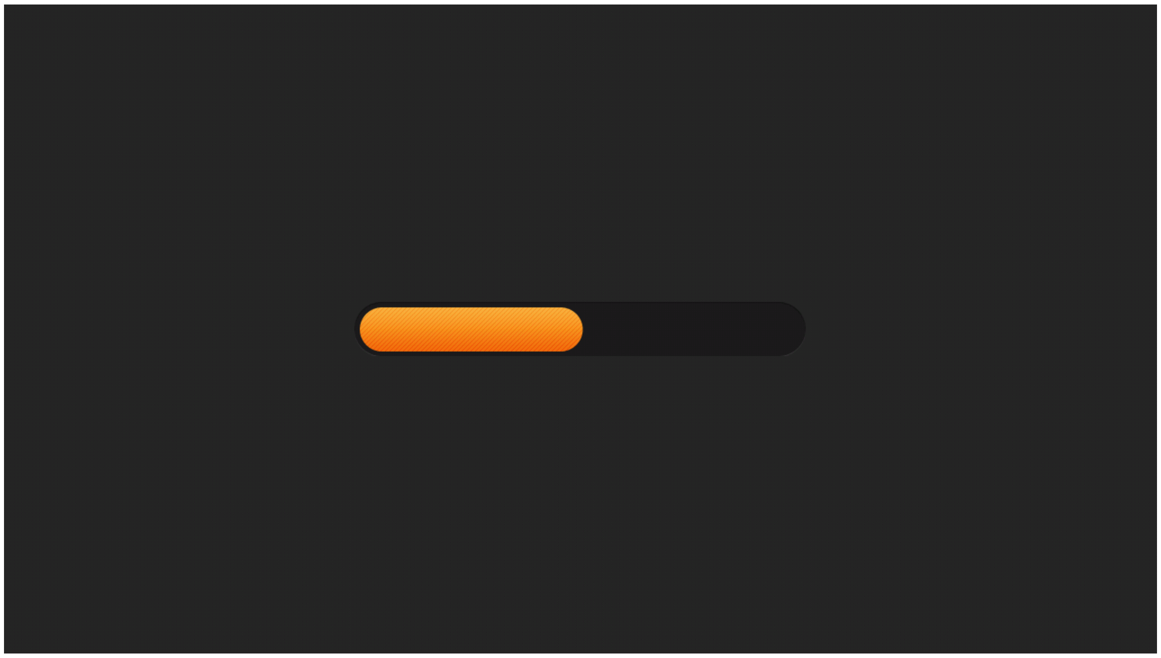


Figure 9: Loading screen

5.3.3 Connection screen

In this screen (Figure 10), the user chooses between connecting to the local server using Wifi or Bluetooth. However, only the Wifi communication is implemented until the date of writing.

5.3.4 Controller screen

This is the app main screen, in which all of the control commands can be performed. Control buttons are grouped in two sets; i. e., default operation mode (Figure 11) and individual joints adjustment operation mode (Figure 12).

In the first mode, we can command the robot to walk forward, to the left, to the right and turn around - by using the directional joystick; also, there are actions buttons, each of

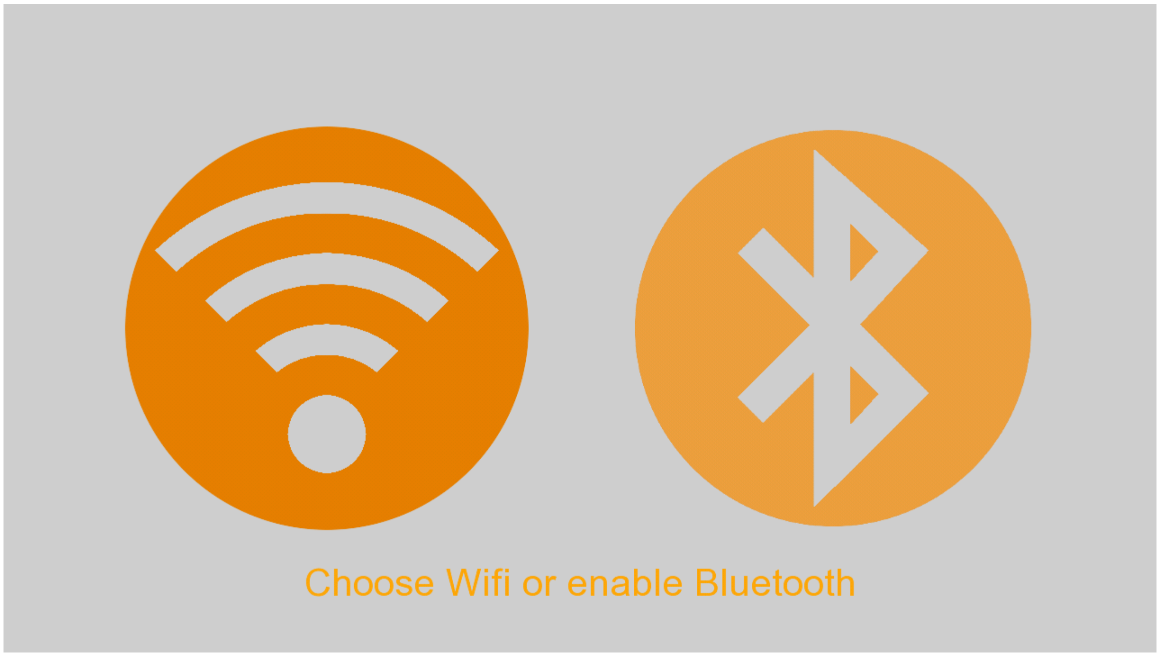


Figure 10: Connection screen

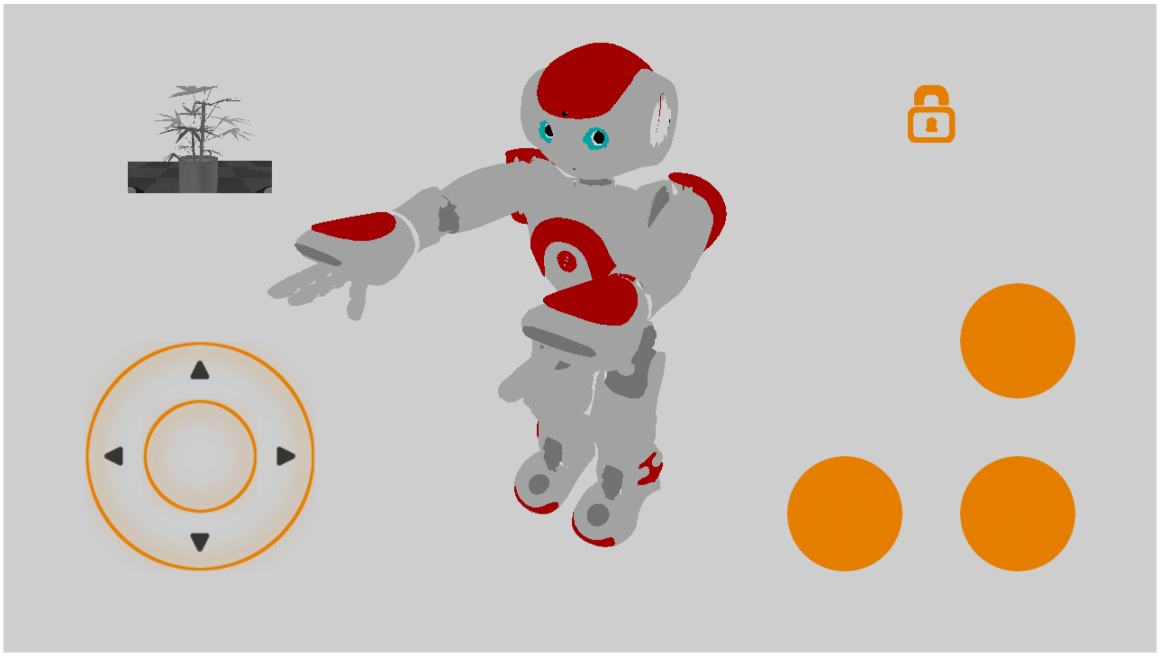


Figure 11: Default operation mode

which for commanding the robot to go to one of the following postures: lay back, sit relax and stand zero.

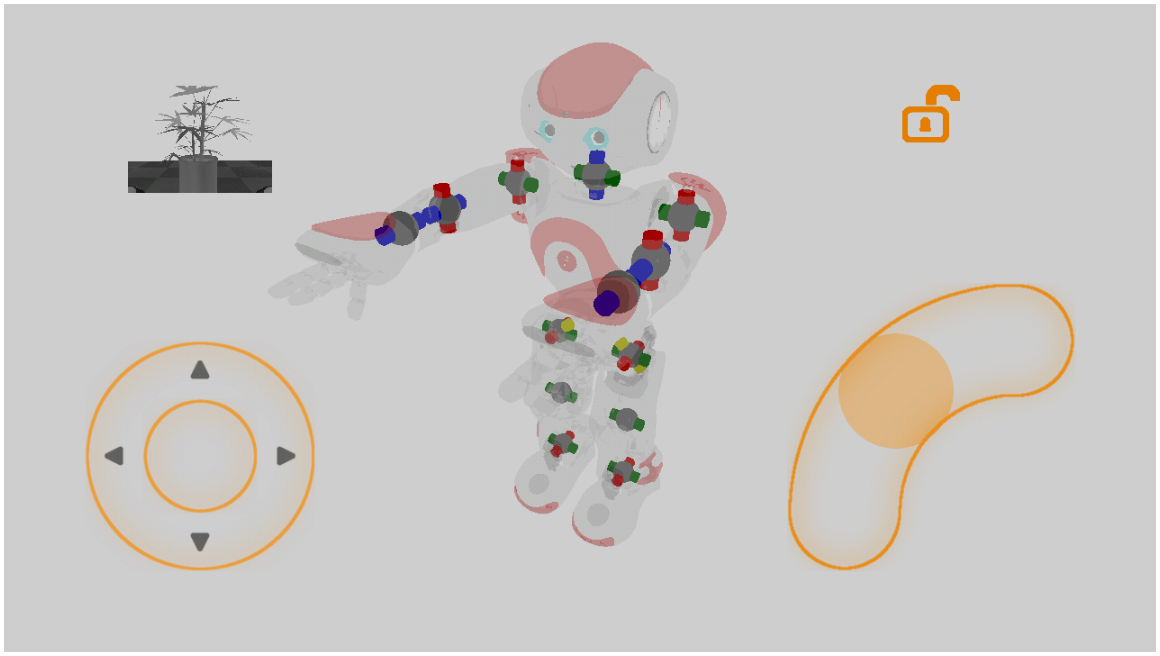


Figure 12: Individual joints adjustment operation mode

In the second mode, we can select any of the represented internal joints of NAO 3D model - corresponding to the real/simulated robot - by clicking on it and increasing or decreasing its yaw, pitch or roll angles by using the directional joystick and/or the slider joystick (Figure 13).

Graphically, a 3D model of NAO is displayed at the center of the screen and it mimics the current pose of the real/simulated robot, and, as we already saw, may also be used as a controller interface when in the individual joints adjustment operation mode. Moreover, in this operation mode, the model has a level of transparency that makes it easier to view and select any of its represented internal joints.

The screen also has an image located on the left upper side that displays the image from the robot's head vision sensor sent by the local server.

Finally, the screen has a locker image button located on the right upper side. Clicking on this icon alternates between default operation mode and individual joints adjustment operation mode.

Operationally, on the show method of the screen, newSingleThreadExecutor() returns ExecutorService with one thread pool size. ExecutorService uses single thread to execute the task. Other task will wait in queue. If the thread is terminated or interrupted, a new thread will be created.

On the render method of the screen, the task GetJointAnglesAndVisionSensorImage is submitted to the ExecutorService at, approximately, every 0.1 seconds. In this task, the worker thread first tries to get the lock, then, if succeeded, it sends the command get joints angles and vision sensor image to the local server, receives an response array of bytes, up-

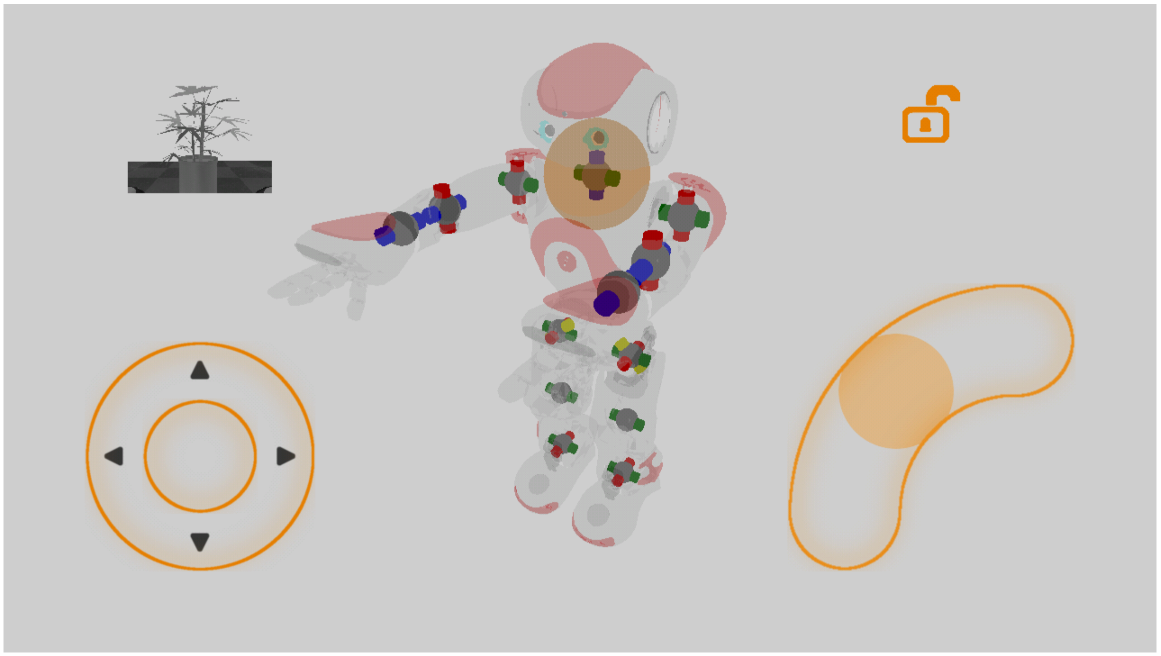


Figure 13: Individual joints adjustment operation mode with the head joint selected

dates NAO 3D model joints angles, orientation and z-position, sets the calculateTransforms variable to true, updates the image from the robot's head vision sensor sent by the local server and, finally, releases the lock. On the main thread, at every execution of the render method, it tries to get the lock, and, if succeeded, then, if the calculateTransforms variable is set to true, it calculates the transforms on NAO 3D model and resets the calculateTransforms variable to false. We chose to use this logical structure in order to achieve good execution performance due to the heavy cost of the transformation calculation operations.

Approximately every 0.2 seconds, the joysticks of the current operation mode are polled and, if not in resting position, the task WalkTurn - corresponding to the move actions - or ChangeJointsAngles - corresponding to the control of the individuals joints of the robot - with its respective parameters is submitted to the ExecutorService. In each of those tasks, the worker thread sends the corresponding command to the local server.

Finally, clicking on any of the action buttons submits the task GoToPosture with its respective parameters, i.e., the posture id, to the to the ExecutorService. Then, the worker thread sends the corresponding command to the local server.

6 Results

When testing the control of a simulated robot, all of the commands performed well, except by some occasions when the walk action and the go to posture command leaded to divergent results between NAO qi simulated robot and the V-REP simulated robot. The main reason for this behaviour is the lack of synchronization of the two robots caused by some delay on the synchronization thread. Its worth noting, however, that this problem only occurs because of our exceptional scenario of having to synchronize between the two simulated robots.

Regarding to the movements of NAO 3D model, considering that the model is large and that the calculate transforms operations are potentially slow, with an update rate of 0.1 seconds, our communication protocol and the in app concurrence control of threads proved to be efficient, since scenarios of slowness or long time desynchronization were rarely observed and were most of the time related to the behaviour of the synchronization thread. Additionally, updating the displayed image obtained from the robot's head vision sensor image is another potentially slow operation, that, however, did not impact the overall performance of the app.

7 Conclusion

In this work, we were able to implement generic controls interfaces, our own logical protocol of communication, optimized for our needs, and graphic interfaces following the design principles of simplicity, which makes designs and mechanisms easy to understand and aware of inconsistencies; and restriction, which minimizes the power of an entity and its communication needs [1].

Despite the proposed actions in the context of robot soccer were not yet implemented (kick, defend, etc.) and the no-use of the mobile device's sensors, the designed app structure encapsulates actions, generic enough, for serving as basis for introducing of all of those specifications with minimal code modifications.

8 Future work

For future work, we propose the implementation of actions in the context of robot soccer, using the designed app structure as an initial point, and expand its uses for real humanoid robots.

References

[1] Matt Bishop. *Computer Security: Art and Science.* Addison-Wesley Professional, 1 edition, 2015.

[2] libGDX documentation.
https://libgdx.badlogicgames.com/documentation.html (Retrieved 12/10/2016).

[3] Gimp tutorials.
https://www.gimp.org/tutorials/ (Retrieved 12/10/2016).

[4] Softbank Robotics resources.
https://community.ald.softbankrobotics.com/en/resources/software/language/en-gb (Retrieved 12/10/2016).

[5] Blender tutorials.
https://www.blender.org/support/tutorials/ (Retrieved 12/10/2016).

[6] NAO joints.
http://doc.aldebaran.com/1-14/family/robots/joints$_r obot.html (Retrieved$12/10/2016).

[7] V-REP user manual.
http://www.coppeliarobotics.com/helpFiles/ (Retrieved 12/10/2016).

[8] NAOqi APIs.
http://doc.aldebaran.com/2-1/naoqi/ (Retrieved 12/10/2016).

[9] Jacquot Pierre. Project-nao-control. https://github.com/PierreJac/Project-NAO-Control, commit = 034f6c0c1777632014ba6d890c394b787d0947f5 (Retrieved 12/10/2016).

[10] Remote api modus operandi.
http://www.coppeliarobotics.com/helpFiles/en/remoteApiModusOperandi.htm (Retrieved 12/10/2016).

[11] libGDX.
https://github.com/libgdx/libgdx/wiki (Retrieved 12/10/2016).

[12] Managing your assets.
https://github.com/libgdx/libgdx/wiki/Managing-your-assets (Retrieved 12/10/2016).

[13] Tutorial: How to use the libGDX TexturePacker.
http://www.programmingmoney.com/libgdx-texturepacker-tutorial/ (Retrieved 12/10/2016).