LO53 Project Report
Wi-Fi Based Positioning System

Celian GARCIA Johann NOVAK Aurelie PELLIGAND
Donatien RABILLER

June 8, 2015

Introduction

The purpose of this project is to build an indoor WIFI-based positioning system. First of all, it is based on three different kinds of technologies : mobile phones (hosting an Android operation system), a computer (hosting a positioning server) and WIFI access points (TP-Link devices).

For this project, in order to be efficient, we had to divide the work to do into 3 categories, as mentioned above. Therefore, two persons were working on the server and the database, one working on the Android Application and finally one configuring the TP-Link devices.

Considering how the positioning system works, one have to go through two main steps. The first one is called the calibration whose aim is to stock inside a database, some data that the mobile phone can send and which helps the server to locate afterwards the phone's position. The second step, called positioning, is used by the phone in order to be located in an indoor environment.

The android application is what the user uses to perform the above-mentionned operations. It contains simple and intuitive graphical interfaces which are detailed in the user guide.

Concerning the Access Points[1], we used OpenWRT and C langage. To be able to locate precisly an indoor device, we need to make measurements of the distance between the device and at least three points in the area. To do these measurements, we have implemented a solution which consists in measuring the strength of a signal emitted by the device which is captured by the APs. This is done by running a program on the TP-LINK's OS which sniffs packets flooding the network allowing to compute the RSSI of the device.

The server can be considered as a link between the mobile phone, the TP-Link devices and the database. It communicates with each three of them and receive/send/insert/request data from/to entity.

The tools we used to create this positioning system are, the Eclipse IDE in order to create the server, the different classes required for the project and to connect, through Hibernate to a postgreSQL database. We also used Android Studio to program the android application.

We created a scheme to illustrate how our program works and the different interactions between the devices :

[1] Also known as AP

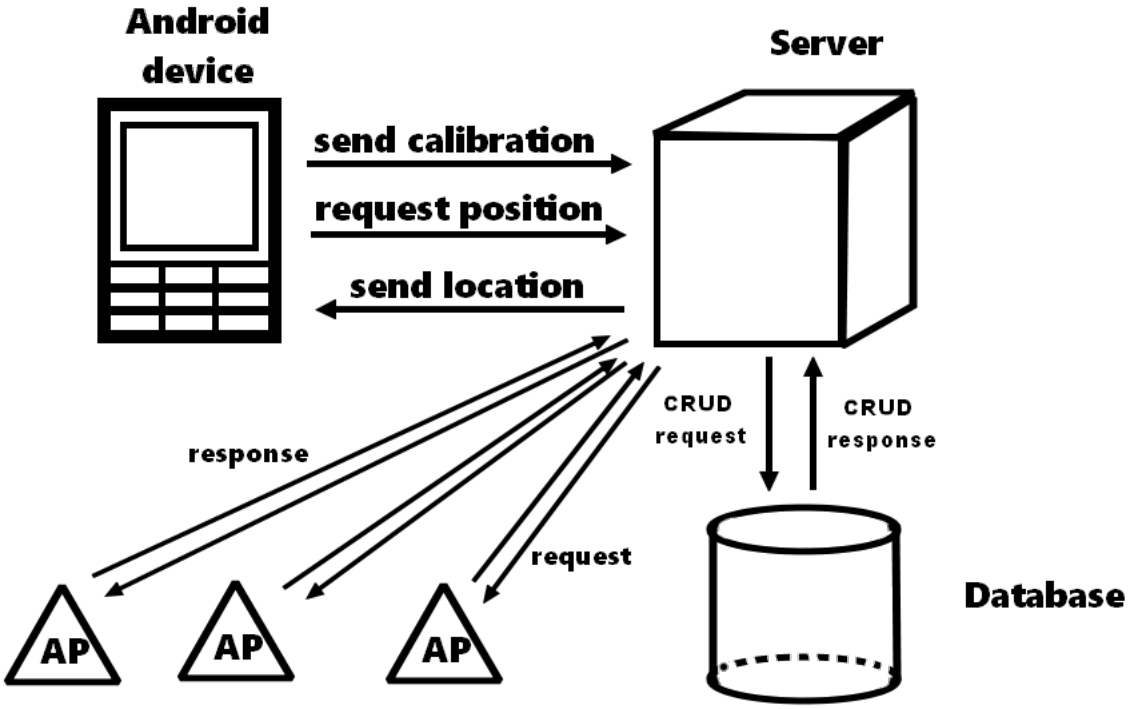

Figure 1: Overall communication scheme of the project.

Contents

Chapter 1

Graphical Indoor Positioning : the Android application

1.1 Purpose of the android application

The android application is the unique graphical interface the user can access for this project. It allows him to perform two operations:

→ calibrate the program by sending a fingerprint of points to the server;
→ enjoy seeing his position travel across the map.

Besides, we had in mind an application which allows to easily transmit calibration informations because creating a fingerprint map can quickly become fastidious for the user.

Another aim was to make the graphical interface easy to handle for the user. This is done by creating a viewport which is an object that can be scaled, translated and rotated. The way it is handled looks like how google maps are handled (this way, the user already know how to perform the operations).

1.2 Architecture

Most of the work done in the architecture is naturally divided in two part :
→ Calibration side : everything related to the sending of a calibration position in order to make the fingerprint ;
→ Location side : everything related to the reception of our current position.

Concerning the following class diagrams, their aim is to give an overall view of the architecture. The already implemented classes are not detailed and are represented in green.

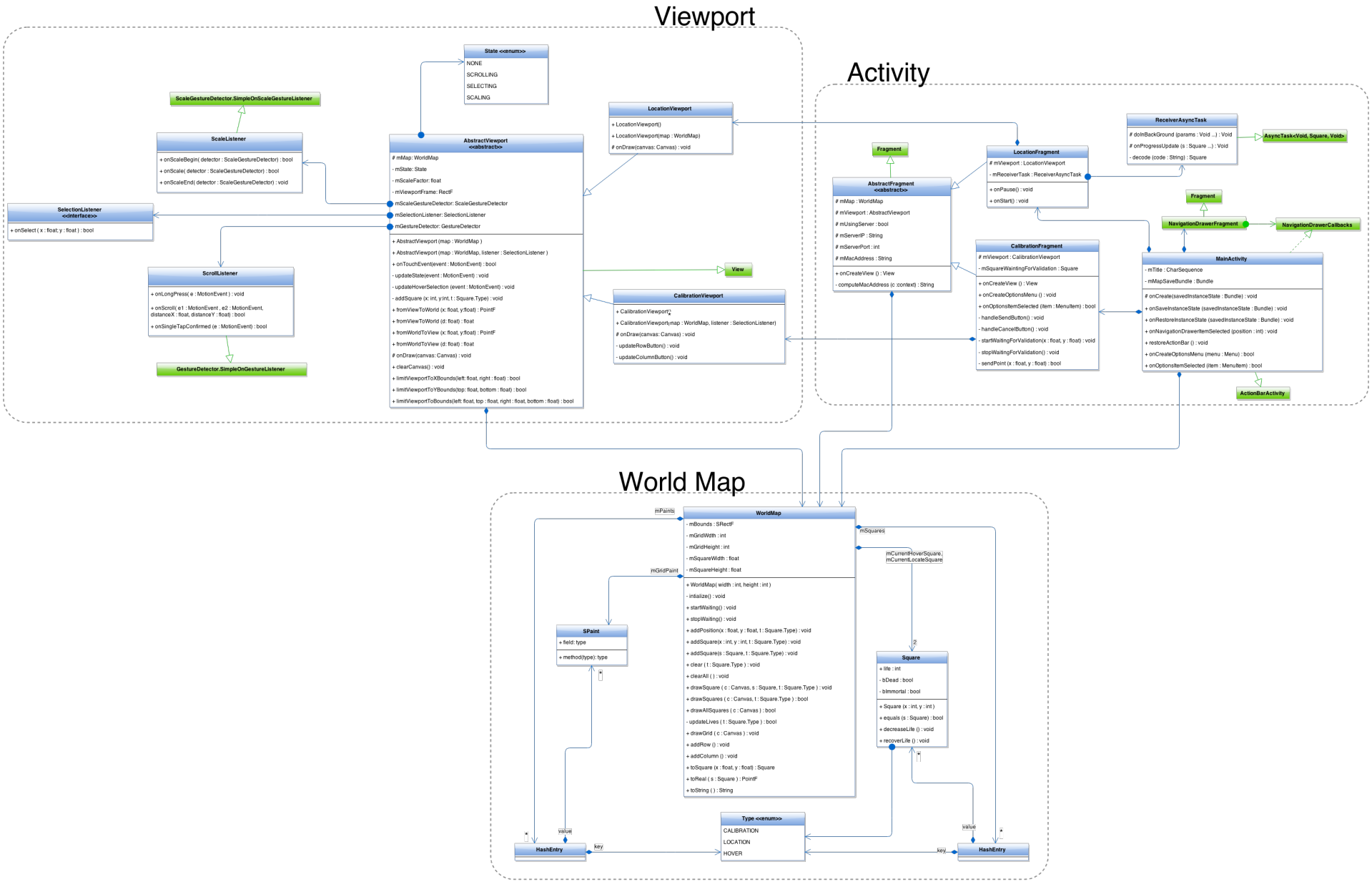

Figure 1.1: Android application global class diagram.

The global architecture can be divided into 3 main parts :

→ The Activity part, which concerns everything related to the activity and the fragments (this is very common in android programming) ;

→ The Viewport part which deals with the viewport (listening to translating or scaling events);

→ The WorldMap part which consists in one class, the world map which is a shared object between the Activity and the Viewport part.

1.2.1 Activity part

This part of the architecture is structured as something very specific to android : the Activity and its Fragments.

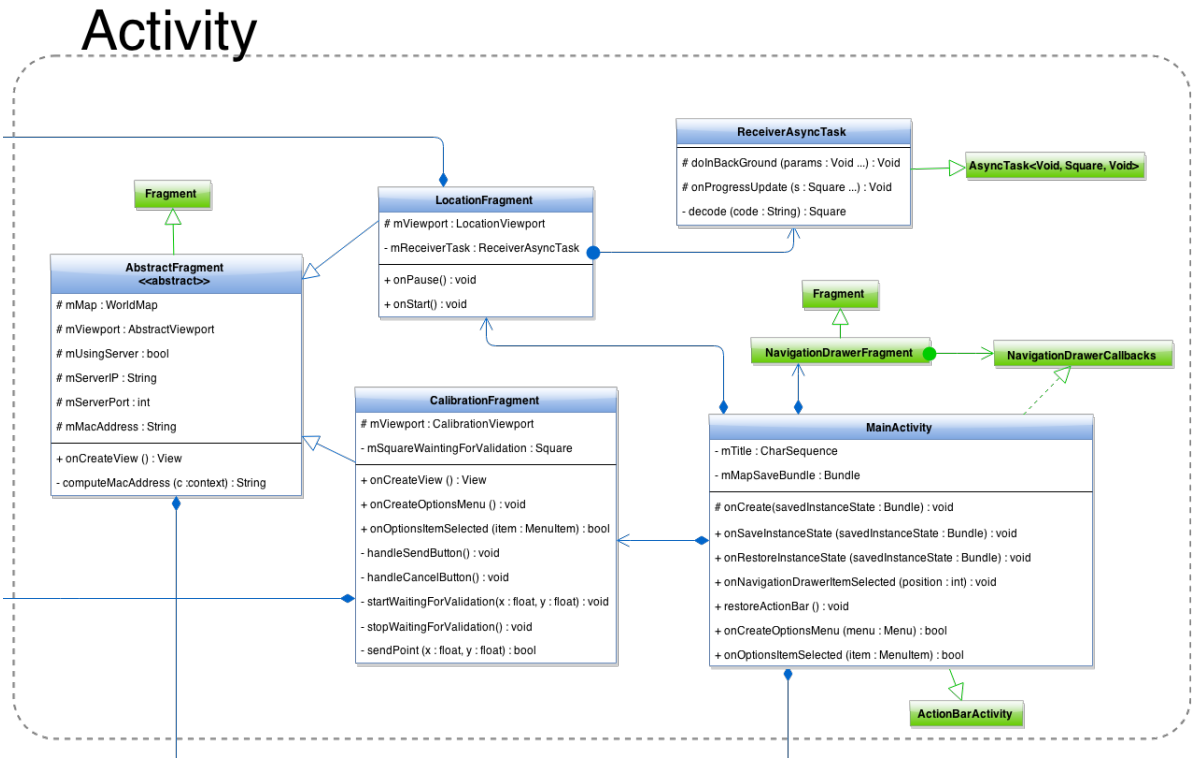

Figure 1.2: Activity class diagram.

The activity can be here assimilated as something near of a *main()* function in classical programming. This is important to mention because it is not always the case with the android API. Most of the time, one activity is related to one operation the user can perform.
It is made to handle four fragments :

→ the CalibrationFragment;

→ the LocationFragment;

→ the NavigationDrawerFragment : made for switching between the previous two ones ;

→ the SettingsFragment.

The main activity is composed of an action bar and a container. The container can contain either the calibration, the location, or the settings fragment. This is the navigation drawer fragment which handles the switch between possible contains of container. The NavigationDrawerFragment is not detailed in the report because it is irrelevant considering the main purpose of the application, unlike both other fragments which represent the core.

Moreover, the SettingsFragment was a last-minute inclusion so it does not appear in the class diagram.

The LocationFragment and the CalibrationFragment extend both of an AbstractFragment which defines methods to be implemented such as :

→ get the world map bundle;
→ get the mobile phone mac address;
→ fetch the server's ip and the 'using server' boolean from the preferences.

How the location and calibration points are respectively received and sent using sockets is detailed in the third chapter of this report.

1.2.2 Viewport part

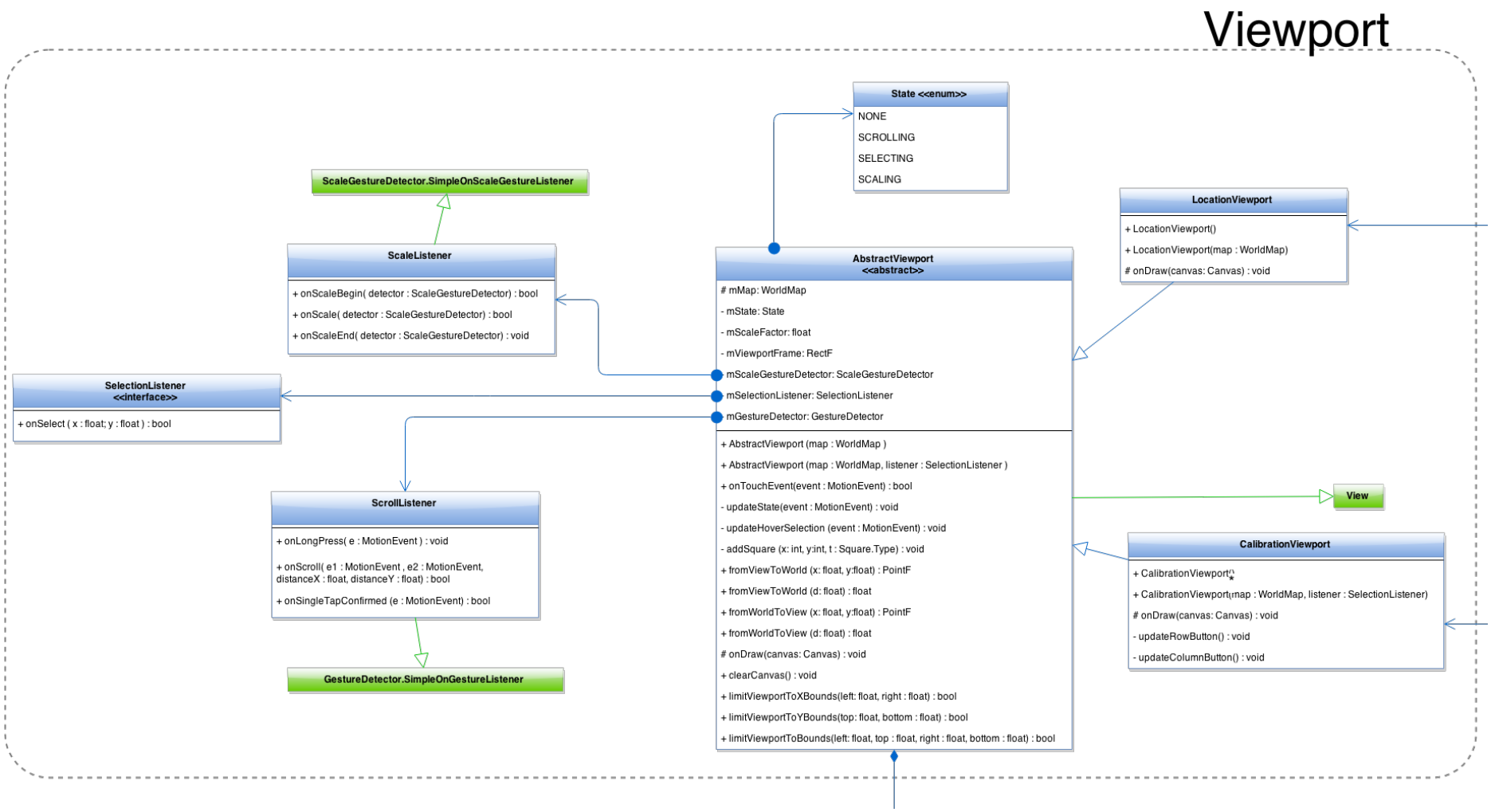

Figure 1.3: Activity class diagram.

Just like for the fragments, an AbstractViewport class have been done. This is where the scaling and dragging behaviours are defined. The Location and Calibration fragments are just here to draw points and, in case of calibration, draw the "add row" and "add column" buttons.

A state machine to handle finger's inputs

Creating a dynamic viewport is not an easy thing. The first we have to do is differentiate several states. All states are defined in an enumeration. Here is a transition-state diagram which defines how inputs are handled.

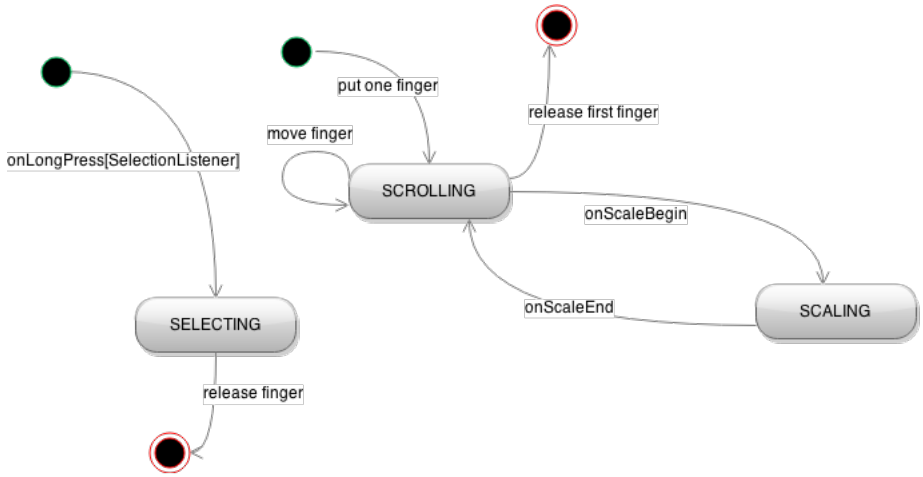

Figure 1.4: State diagram of the viewport.

→ *NONE* state : The application doesn't have any state, the user is not doing anything;

→ *SELECTING* state : the user is selecting a square to send to the calibration server. The hovered square is colored in green. The long press begins when the user touches the screen without moving and wait like this for one second. Besides, in order to reach the '*SELECTING*' state, the condition *onLongPress[SelectionListener]* to the diagram to indicate that if the SelectionListener is null (the case in LocationViewport constructor), then it is naturally impossible to reach the SELECTING state.

→ *SCROLLING* state : the user is translating the map with one finger.

→ *SCALING* state : the user performs a classic scale with two fingers on the screen.

It is important to mention that there is no way of switching directly from the

SCROLLING/SCALING states to the *SELECTING* state and inversely. It is a choice made in order for the user to be able to scroll and scale without preoccupying himself on how long his finger did not move.

Relativity

When the different states are defined, we have to handle relativity. Relativity in the application means that, if the user can zoom and translate the viewport, all graphical objects must have global coordinates (on the map, or the "world") and local coordinates (on the screen).

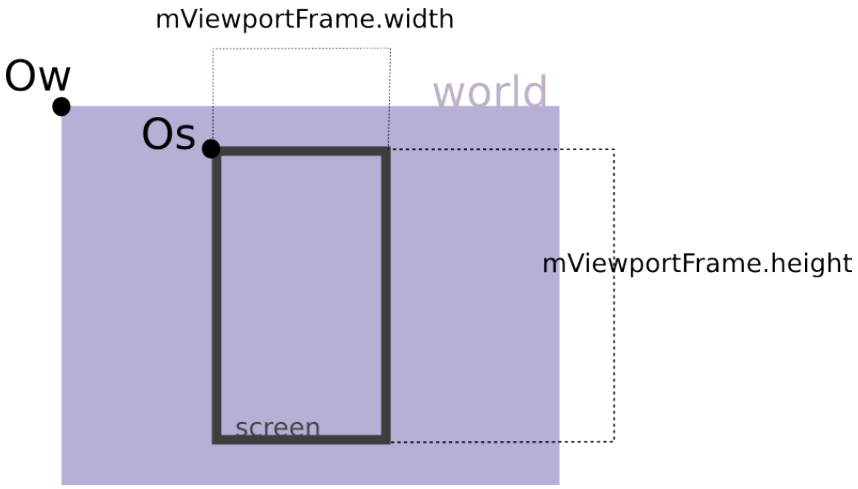

Figure 1.5: Viewport relativity diagram.

The **mViewportFrame** field is a RectF expressed in world coordinates. *Ow* is the world origin (ie (0, 0)), and *Os* is the mViewportFrame offset. This frame act like a screen avatar in the world system coordinates. We defined also a **mScaleFactor** which is the ratio between the screen size (ie canvas size) and the mViewportFrame size.

In order to make it more readable in the code, we created two indispensable functions which use these two elements (mViewportFrame and mScaleFactor) to convert points or distances:

→ $fromViewToWorld()$;
→ $fromWorldToView()$.

Either of them takes 2 coordinates as parameter and returns the transformed point as a PointF or as a distance (as a float value). Before creating these two methods, it was impossible to handle relativity problems with clarity.

Border limits of the viewport

Naturally, we had to introduce border limits in the application in order to offer more comfort for the user. Functions which handle this topic share the name

limitViewportToBounds. They reset the mViewportFrame according to the bounds given in parameters.

The selection listener

The selection listener is an inner class which have to be given in the Viewport constructor if we want to handle a selection behavior. If it is null or not given, the *SELECTING* state won't be reached. Indeed it is the case in Location viewport, where the user is not able to select a square.

1.2.3 WorldMap part

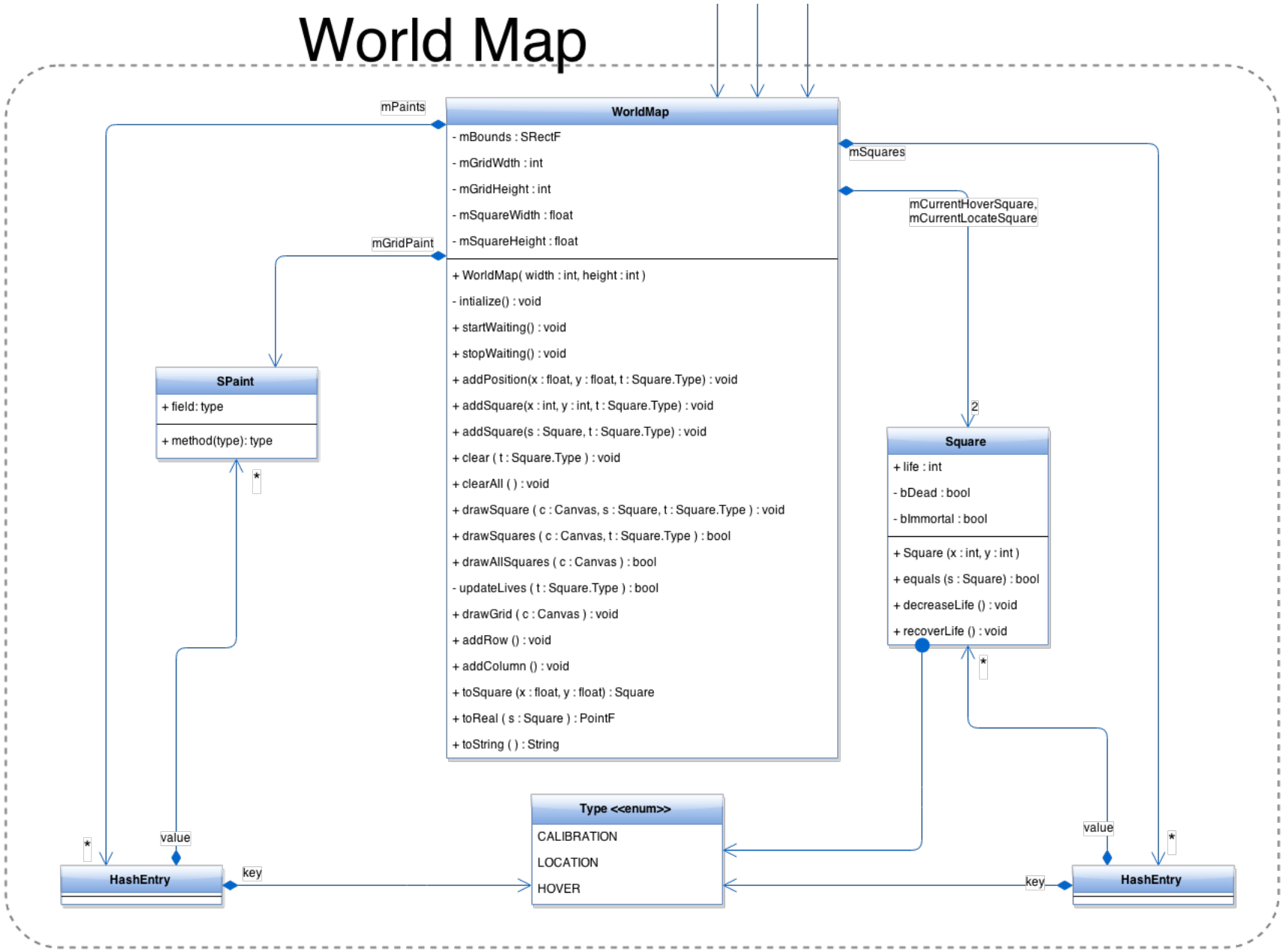

Figure 1.6: Class diagram representing the WorldMap.

This class his handled to store and draw squares. There are 3 types of squares : *CALIBRATION*, *LOCATION*, and *HOVER*.

Each square contains a *life* property which is used to handle a fade out behavior : Calibration squares are set as immortal, they are always drawn and never fade out whereas Location and Hover squares are mortal, they fade out if they are not the last added one.

1.3 Choices made and relevant details

1.3.1 Modifying the settings only at the beginning

We decided to disable the modification of the settings when either calibration or location have been launched because we wanted to have a certain consistency in the project. If we could change server IP between calibration and location sessions, it would have quickly confused the application. Moreover we use a unique server IP for both so it would have been totally irrelevant and useless to permit the application changing it when running.

A persistent map common to all fragments

As described in the purpose part, the world map has to be unique for all the application, it also has to :

→ be persistent : it means that if we let the program run in background or if we let the screen turn off, the map should not be removed;

→ be able to be shared to all fragments from the main activity.

There are several ways to do such things and the one selected is very efficient as it can do both, we are talking about the **Bundle**.

Bundle The aim of the Bundle class is to store data in order to serialize/deserialize it later. To store data inside, we instantiate it and insert data with a put function : for example we used here putSerializable : *mMapSaveBundle.putSerializable(TAG_WORLDMAP, new WorldMap(width, height));*

It takes :

→ a string tag, which is here a static field of MainActivity;

→ the serializable object that we want to put in.

Share a Bundle When we create a map bundle and in order to share it, we can use the *setArguments()* and *getArguments()* fragment's functions. **In activity** : we call the *setArguments(Bundle bundleToShare)* function just before replacing the fragment container, because the bundle argument has to be set before the call of *onCreateView()* fragment's function. **In fragment** : precisely in the *onCreateView()* function, we call *getArguments()* function to retrieve the bundle.

Store a Bundle When the application is run in background, when we touch the home button or even when the screen turns on, the main activity is instanciated again. So we have to save what we called the persistent data (ie. the map) in order not to flush the data already gathered. It needed to override two activity's functions to save or restore :

→ *onSaveInstanceState(Bundle savedInstanceState)*;
→ *onRestoreInstanceState(Bundle savedInstanceState)*;

What is interesting here is that we can put a bundle inside a bundle. In this way, we can save the map bundle inside the savedInstanceState bundle.

Preference settings more persistent

We could have used again a bundle to handle the user preferences, but at each launch of the application, the savedInstanceState bundle is flushed. This is why android provides something largely more attractive for this kind of persistent data : the SharedPreferences. These are stored into the mobile phone and so will persist even if we close the app.

Data reception in the Location fragment

As we saw, the location has to make a loop with send and receive statements. The problem here is a classic synchronization problem. As we have a viewport which can be dragged and scaled, the reception could not be performed in the same thread. If it was the case, the socket would wait for positions to receive, blocking the thread and so disabling for a moment the drag and scale possibilities. That is why we had to make the reception asynchronous.

We first thought of creating a thread, assign it a runnable Receiver and simply run it. But this is not how it is supposedly done through the Android API because of specific threading behaviors already running. Android discourages the creation of our own threads and provides a very efficient tool : the asynchronous task. It is efficient because we don't have to manage Thread or UIThread, etc ... and it provides a code as clear as possible.

Two detectors to handle two input types

There are ways to handle both scale and scroll inputs using only the scale gesture detector, but when we look at the functions provided by its interface listener, we see that it has not been done in this aim. Unlike the gesture detector interface which contains functions like *onScrolling*, *onLongPress*, or *onSingleTap* which are very useful to handle events. This is why we decided to take advantages from both and use a scale gesture detector to retrieve scale input and a gesture detector to retrieve scroll input.

1.4 User guide

The first screen the user sees is the setting screen. Here are the parameters the user can modify :

→ "Use the server" : Define if we really use a server or if we want a randomly simulated server. Basically it is quite useful for debug, when we have not necessarily a sever at proximity.
→ "Server IP" : the IP of the server to connect to.
→ "Location Port" : the port used by the server for location requests.
→ "Calibration Port" : the port used by the server for calibration requests.
→ "Default map width" : the width (in squares) of the map shown to the user.
→ "Default map height" : the height (in squares) of the map shown to the user.

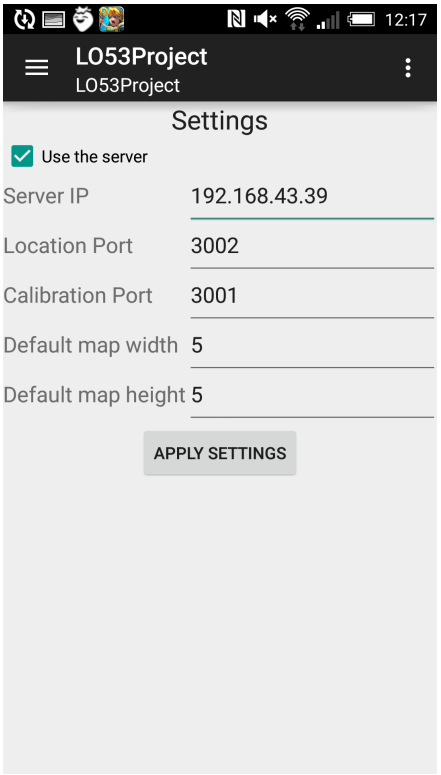

Figure 1.7: Settings screenshot.

When the user is satisfied with the settings, he has to click on the Apply Settings button, or the program will not take in count the changes. Beware, the settings can only be changed at the beginning of the program. Moreover, the next time the program is run, the newly created settings will be proposed.

The next step is to select either the calibration mode (if we want to make a fingerprint) or the location mode (if we just want to see our position). If no fingerprint have been registered yet, the location part will not be able to make any requests. The selection is done in the selector button upper left :

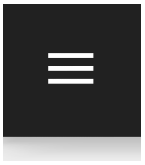

Figure 1.8: Selector button.

Location :

There is nothing specific to do here, just see your position in the room, represented by a white square on the map. The last position received is totally opaque, and fades with time.

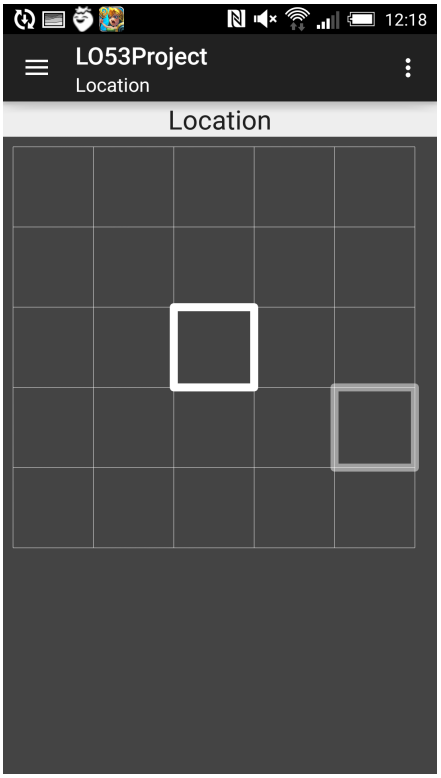

Figure 1.9: In location : White squares represent our position.

Calibration :

First of all, the user can add a column or a row to the map by performing a single tap respectively on the vertical or the horizontal white "button".

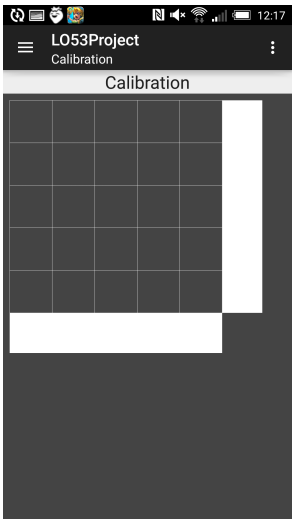

Figure 1.10: In calibration : Before add a row

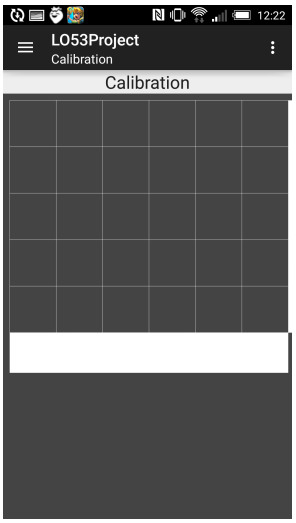

Figure 1.11: In calibration : After add a row

Secondly, and the most important, you can add a calibration position. To do this, select a square by touching it without moving during one second. When the square is filled with green and if you did not release your finger, you can even move across the map while sliding your finger.

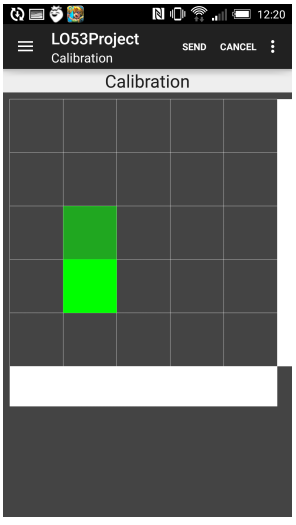

Figure 1.12: In calibration : Selection in progress.

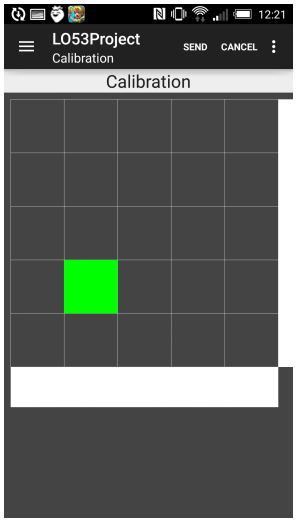

Figure 1.13: In calibration : Selected square persists after finger release

When you are satisfied by the square you selected (ie. it corresponds to your

current real position) you can send it to the server with the 'Send' button on the top of the screen. If you are not satisfied, you can either use the 'Cancel' button, or select another square just like before.

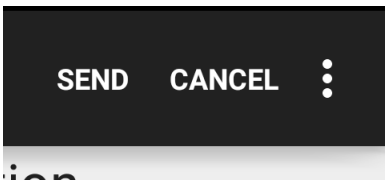

Figure 1.14: Send and Cancel buttons.

When you send the point, and if the server has well received the point, a message is shown and the square is displayed in black.

Eventually, in both calibration and location mode, you can translate or zoom the map respectively with one or two fingers the usual way.

1.5 Ameliorations

The application could certainly be more accurate on the positions given to the user. For example by asking to him :

→ a referential point which defines the map center;

→ a custom geographic scale (ie. width and height of each square corresponds to how many real life meters) decided beforehand by the user. This would improve the precision between the simulated grid map and the real world open space.

Honestly, the graphical design is very poor, not to say non-existent. This is also a point which can be largely improved. We concentrated our efforts on something able to be run and to work, not something beautiful.

In the end, there still are some problems with the zoom limit : unzooming quickly doesn't limit correctly the scale factor.

Chapter 2

TP-Link devices : the programmable WiFi Access Points

2.1 Goal of the TP-Link device

The aim of the TP-Link is to collect RSSI from connected wifi devices allowing the server to compute the actual position of the android device. To achieve that we created two programs, one which catches packets that are transiting through the network, then retrieves the RSSI to finally store its values. The other program creates a socket and waits for the server to connect. When connected, it sends an average value of all RSSI concerning one mobile device (identified with its MAC addres) to the server.

2.2 Sequence diagram of the communication

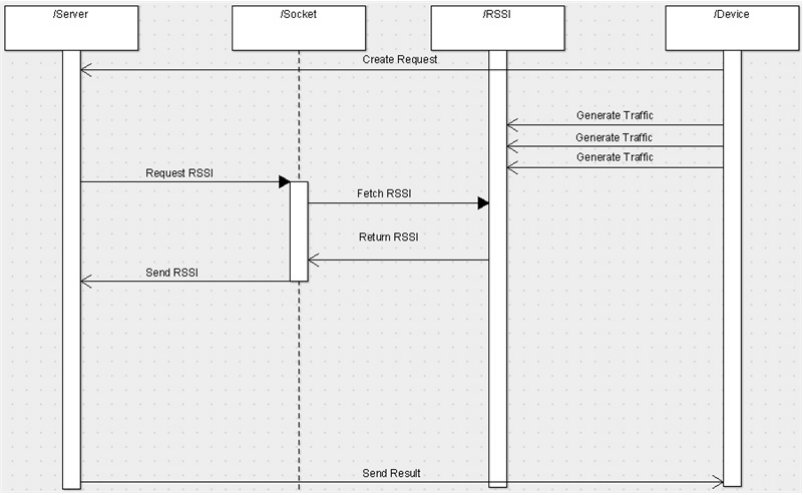

Figure 2.1: Sequence Diagram representing the communication between a TP-Link device and the server.

The Socket and RSSI processes are running on the AP OS, the server and the

mobile device are based on other environments. The device connects to the server for a specific request and then generates traffic which is caught by the AP. It then fills its RSSI list. Later on, the server asks the Socket process the average of the measured RSSIs, the socket accesses the RSSI list and returns the request value.

2.3 Algorithms

2.3.1 Rssi retrievement

Retrieving the RSSI for a packet is realized using the pcap library. Firstly we need to create the packet listener using the function *pcap_open_live* which takes the interface we want to listen to. Then we enter a loop where we catch a packet using *pcap_next*, we check if the packet is correct by checking the first two bits of the header frame control. Then we compute the offset of the RSSI in bytes by looking at the radio tap flags in the *it_present* array.

Once we have the RSSI, which is filtered if unreliable (value of 4dbm for example), and the MAC address we insert it to the RSSI list. So first we lock the semaphore to insure concurrent access with regard to the socket thread, and then we add the value for the right MAC address. We also clear all outdated values (which occurrs when one is in the list for more than one second). Finally the semaphore is unlocked.

2.3.2 Socket connections

To communicate the RSSIs to the server, a socket is opened on a separate thread listening to the port 3000. In order to do so, the standard socket library is used. First we create the socket using the *socket* function. Then one applies some setup to make the socket listen to the port 3000 and finally one loops where connections are accepted, handled and closed.

When a connection is accepted the handler first waits for a new string to arrive. It is then translated into a MAC address. A semaphore is then locked to insure concurrent access with the RSSI thread and the RSSI average value is computed for all RSSIs for one device. The lock is then released and a string is sent back through the socket according to this format : "MAC;RSSI". Finally the socket is closed and one loops again.

2.4 Encountered Problems

The main problem encountered when handling the TP-Link devices was, for an unknown reason, that the RSSI in the packet was not correctly positioned (at the good index inside the caught packet's header). So for certain configurations of the TP-LINK, RSSI measurement was impossible. However, for other TP-Link devices, everything worked out. We are still trying to find a solution for this bug.

Chapter 3

Positioning system : a two-threaded server

3.1 Why use sockets and not JavaEE ?

The server is linked with the other devices through TCP sockets. We use such a technology and not JavaEE because sockets are more appropriate for our needs. Indeed, as our server is communicating with the TP-Link devices with a custom protocol through sockets, the same could be applied for the mobile device. JavaEE is a wonderful technology and has many advantages but it must go through the HTTP/S protocol through a specific port. Besides, a JavaEE server application needs a lot of configuration beforehand (or Spring annotations) whereas only one configuration file is required for sockets.

Moreover, a socket-implemented server can be a standalone application, run as .jar file for example, unlike a JavaEE application, which first has to be packaged as a .war file[1] file, then deployed on a ServletContainer (such as Tomcat, JBoss, etc. . .). Proceeding as such makes tests and debug sessions much harder as it takes more time than to simply execute an application.

[1] **war** : **W**eb **AR**chive, structured archive composed of a /META-INF and a /WEB-INF root directories containing all additional libraries and mandatory .class files.

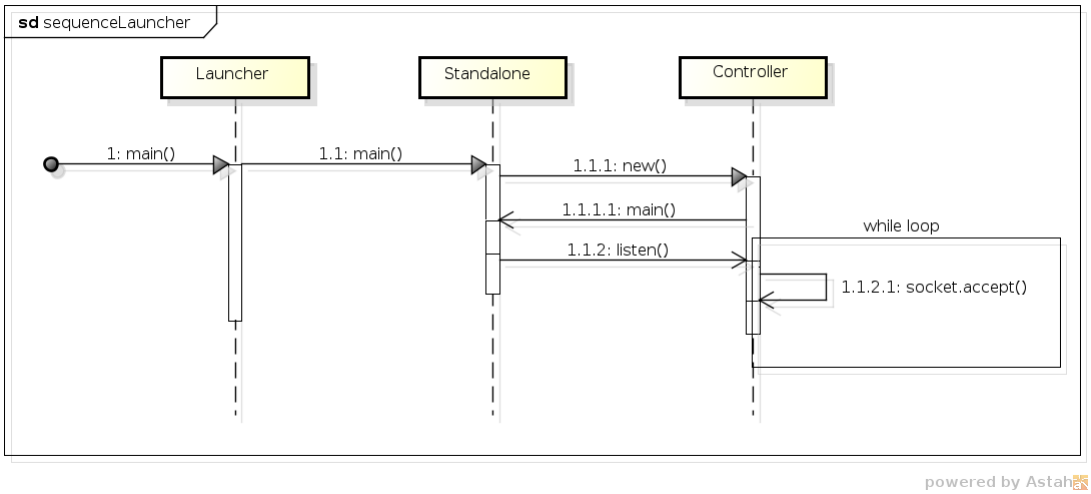

Figure 3.1: Sequence Diagram representing the initialization of the server.

3.1.1 Architecture of the server

Concerning the architecture, it is started through the Launcher.java class. It creates two threads that respectively instantiate two controllers : CalibrateController and LocateController. Our project respects the Service Oriented Architecture[2]. Each of these controllers possesses as attribute a ServerSocketChannel waiting for entering connections from mobile phones and each of these ServerSocketChannels are bound to a specific port. The first one is used to handle mobile devices calibration requests and the other one answers positioning requests.

The controllers use SocketChannels (with a Selector) and which are non-blocking socket (unlike normal TCP). The principle is simple, one registers a ServerSocketChannel inside the Selector object and then waits for any entering connections. When a client binds itself with the server, it then registers a new SocketChannel inside the selector which can be accessed afterwards. However when one handles these channels, the order cannot be respected as the selector is just an encapsulated set.

Each controller can access the database through the service layer with the classes CalibrateService and LocateService. Finally, these services are linked with the repository layer which has access to the database.

[2]Also known as SOA. Architecture model composed of layers. In our case, there are three layers : controller, service and repository.

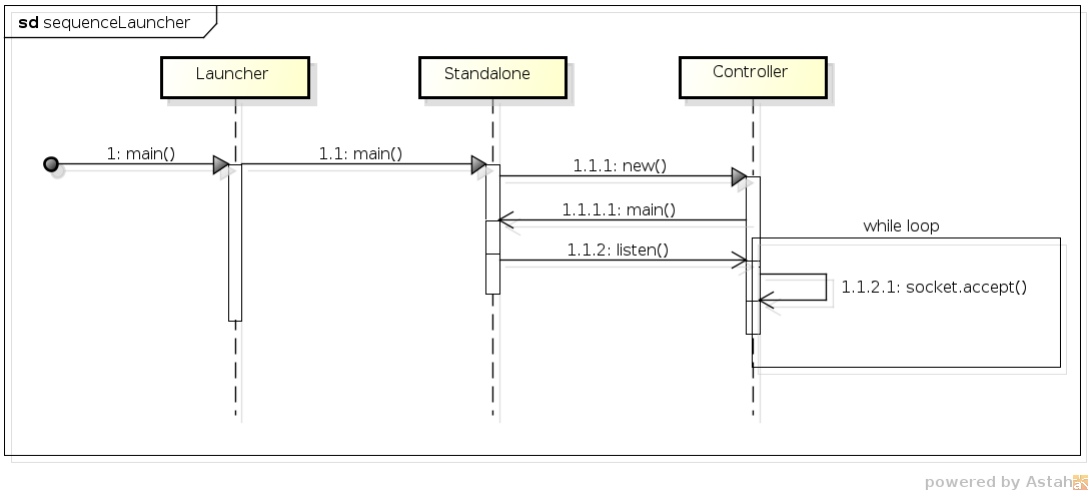

Figure 3.2: Class Diagram showing the architecture of the server.

3.1.2 Socket Channels replacing Threads

Advantage of Socket Channels

We first used normal TCP ServerSocket Java objects but the number of thread created was too important. It used to start as many threads as mobile devices' requests and for each request, for all APs, a new thread was started in addition to ask associated RSSI values. The performance was too dependent from the hosting computer's number of processors. This is why we changed the architecture thanks to the Java "*Selector*" object.

Handling of channels

Our implementation divides three types of channels :
→ connectable channels;
→ readable channels;
→ writable channels.

Connectable Channels They correspond to sockets which tried to bind themselves with the server. Such channels are immediately registered in the Selector as 'READABLE'.

Readable Channels They are the most important ones. When handling some, the data sent by the mobile device is parsed. It communicates then with the APs to have the associated RSSIs. Eventually, the database is accessed in order to query or insert values.

Writable Channels This channel is used to send back a response to the mobile device. When it is done writing, it finally closes the client's SocketChannel.

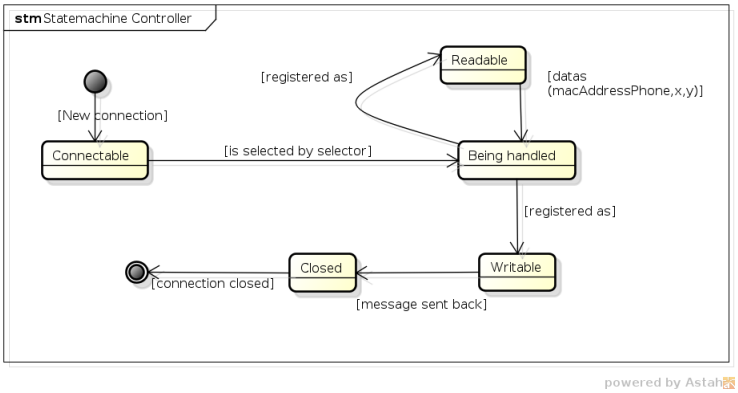

Figure 3.3: State machine representing the handling of channels.

3.1.3 Service Oriented Architecture

The SOA is an architecture whose purpose is to completely divide jobs to perform by the means of different layers. In this project, three layers have been created : the controller one, the service one and the repository one. Each one of them have a different role. The advantage of this architecture is that if someday, the way to access the database has changed for example, one will not have to modify the whole code but just the repository layer by adding a new DAO[3] class.

Controller layer It is where the endpoints of the server are stored. The role they have to carry out is to : listen to any entering connection from a client, control the consistency of the request, perform some mandatory tasks and finally relay the information to the service layer.

Service layer Intermediary layer linked to the controller and the repository layer. Its existence allows not to let the controllers have a direct access to the repository and access directly the database.

Repository layer Model-linked layer. It accesses the data which can be stored in different way (database, files, etc...) and can perform CRUD[4] operations.

These are the class diagrams of the communication between the different devices:

[3]**DAO** : **D**ata **A**ccess **O**bject. Interface providing a way of converting database objects into standard Java Objects.

[4]**CRUD** : **C**reate, **R**ead, **U**pdate, **D**elete, operations which can be performed on a database.

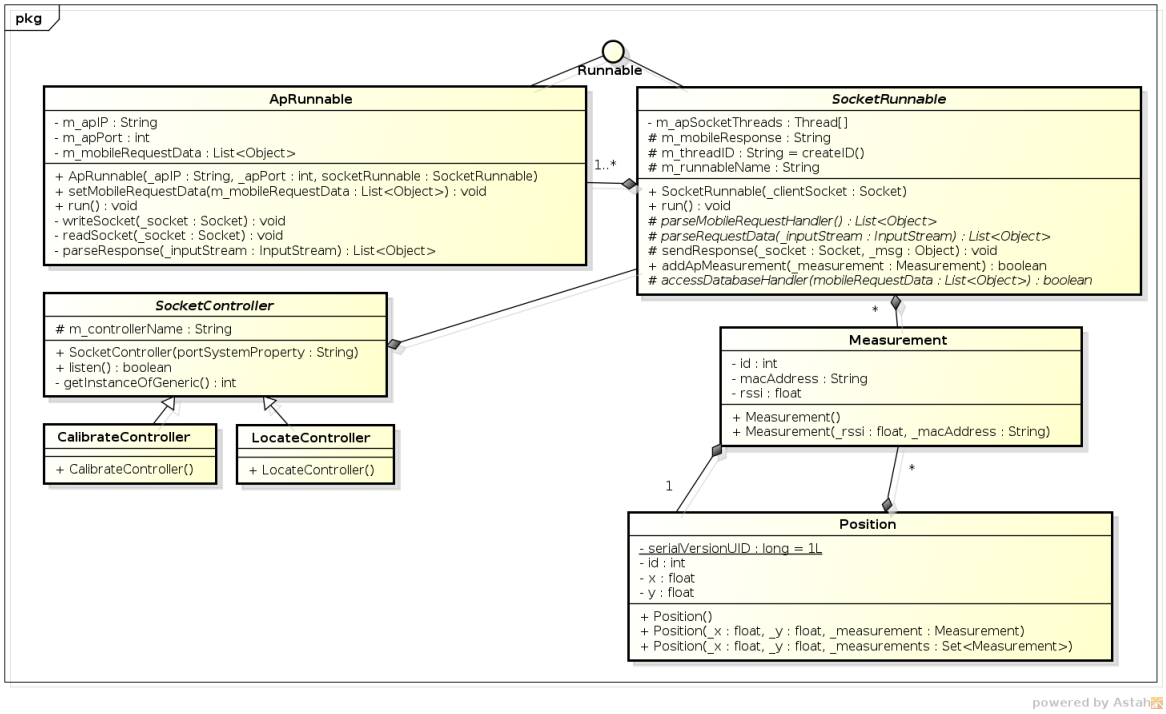

Figure 3.4: Class diagram showing the classes communicating with the controllers.

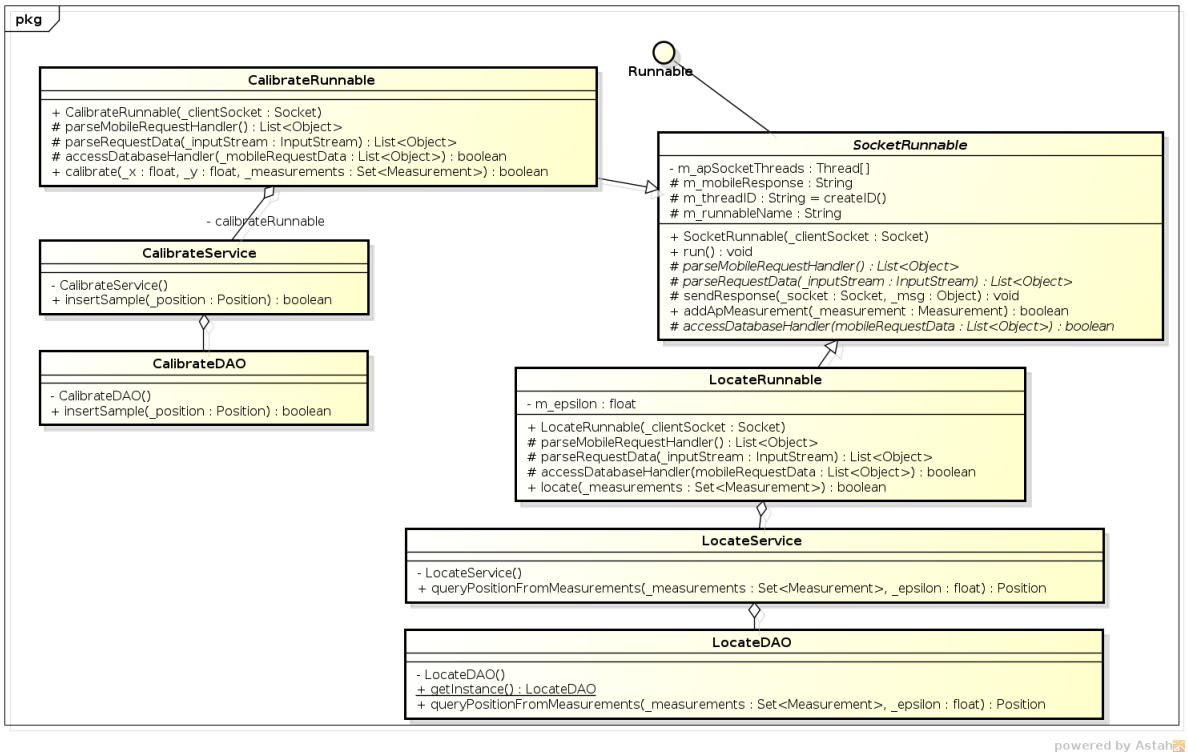

Figure 3.5: Class diagram representing the way the database is accessed from controllers.

3.1.4 A database accessed through ORM

Database side

The project's database is in PostgreSQL language because it has several interesting advantages. For instance, it is known to never crash, it is quite practical unlike other language concerning tables management and moreover, it is an open source software.

The database contains two main tables, "Position" and "Measurement", linked to each other with a one-to-many↔many-to-one relationship :

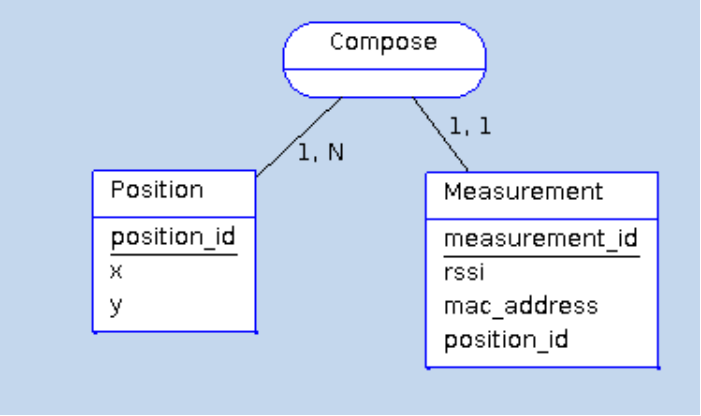

Figure 3.6: Merise MCD showing the database's table relationships.

Server side

Nowadays, there are two main ways to obtain server side (in Java), data from a database : JDBC and Hibernate.

Java DataBase Connectivity It consists in accessing directly the database by creating an instance of connection. CRUD operations can then be executed as statements and eventually the connection has to be closed at the end of the access. This way of interacting with the database is not very efficient when one needs to perform many connections as one has to open and close connections each time.

Hibernate API used for database object persistence in Java (or ORM[5].). It can convert by the means of configuration files, database objects into POJO[6]s. It is also able to reproduce relationships and linked tables. It works thanks to the "session" concept. Each time one has to access the database, one asks to get a new instance of Hibernate session where one will be able to perform CRUD operations more easily than JDBC. Some methods are already implemented in order to facilitate such operations. This is the technology used in this project. Three configuration files were required :

→ *hibernate.cfg.xml*, which defines how Hibernate can access the database and which tables have to be linked to which Java class (in our case, Position.java and Measurement.java);

[5]**ORM** : **O**bject **R**elationnal **M**apping

[6]**POJO** : **P**lain **O**ld **J**ava **O**bject, standard Java objects (in contrary of Java Beans for example).

→ *Position.hbm.xml*, which corresponds to how can the database table "Position" can be mapped to this Java class;
→ *Measurement.hbm.xml*, which corresponds to how can the database table "Measurement" can be mapped to this Java class;

3.1.5 How communication is performed between devices ?

An overall sequence diagram has been created in order to look how a mobile request is handled all the way to the end.

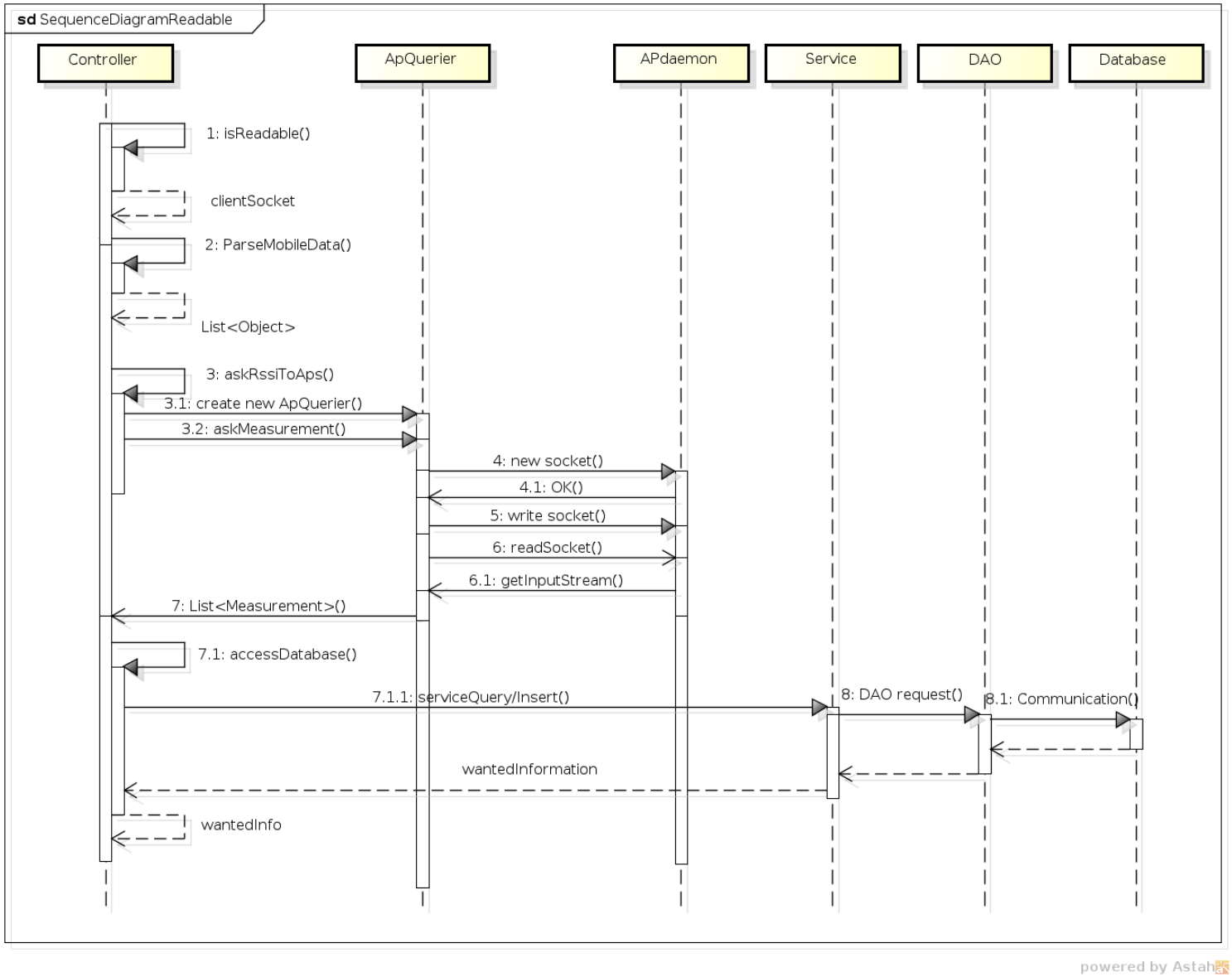

Figure 3.7: Sequence diagram of the handling of a readable socket.

3.1.6 How are the errors managed ?

The server is running until the application is shut down by the server's administrator. So it must absolutely not crash. Hence when errors occurr, they are all cought and handled in a way allowing the server to continue running. In order to create such a error-proof architecture, one must be aware of all the states of the server, one must know the values of the different attributes when passed from methods to methods and the values of the returned objects.

Conclusion

We can conclude that this project was first, hard to achieve. There were several steps for us in order to comprehend the big picture :

→ what were the different steps of the communication between the devices;
→ how can the pcap library be used;
→ what protocols can be used in order to transport the information between all devices;
→ how the android graphical interface can be designed;

As the server required sockets in order to communicate with the Access Points, the same technology has been applied to the communication between the mobile phone and the server.

The whole structure of the project relies on the fact that our database has to be filled through a calibration beforehand. It compels the application user to first perform a calibration. This had an impact on both the android application and the server as it needs an extra step before locating.s

The application was globally an enriching first android development experience, because we had the occasion to have an overview of a large panel of its specificities. Thanks to the Android API (written in Java but is a very specific use of Java language), very efficient shortcuts, were at our disposal and it was quite easy to understand all base concepts.

Some improvements can be made on the server side. First, the handling of channels is made with 3 steps but it could have been separated into additional steps : one for the request parsing, one for the APs' RSSI requests and one for the database access for example. Also, a trajectory deduction could have also been implemented by inserting in the database past positions concerning a mobile phone and by deducing server-side, a global direction. Finally, by adding a new table inide the database related to the mobile phone, we could have implemented a system to give to the phones, the other phones location along with the requested one.

This project has made us aware on the fact that nowadays, devices can interact together by many means and can communicate all kind of information to each other. We also discovered new technologies and how to use them such as the Android API, non-blocking TCP sockets or the pcap library. Furthermore, this project was not made individually but on the contrary, it was achieved as a team helping each other out.

Appendix

Sources

→ Software used in order to create the server in Java : Eclipse
http://www.eclipse.org/
→ Software used to create the Android Application in Java : AndroidStudio
https://developer.android.com/sdk/index.html
→ Database we used for the positioning system : PostgreSQL
http://www.postgresqlfr.org/
→ Software used to code the report in LaTeX : TexMaker and ShareLatex
http://www.xm1math.net/texmaker/index_fr.html
www.sharelatex.com
→ Software used to connect to the TP-Link devices : Putty
http://www.putty.org/