LO53 Project
WiFi positioning system

CADORET Luc – COUSSANES Jérôme – FELLAH Djamel – ROBERT Julien – SCHULZ Quentin

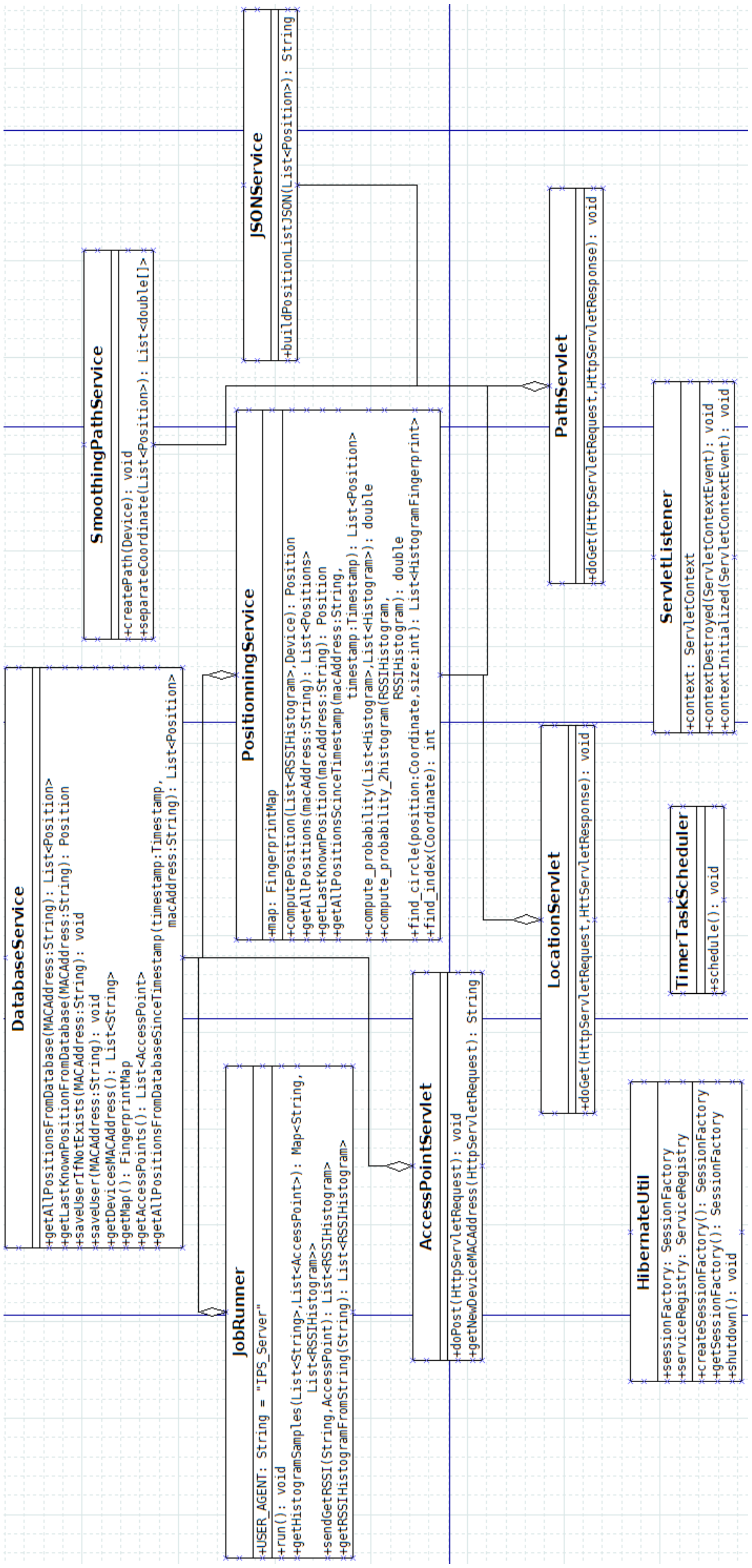
Summary

The server

I – The role of the positionning server

The positionning server is the core of the whole process. It regularily asks for the RSSI that the APs are capturing with a timed task, it registers the news devices MAC addresses in the database when an AP detects one, it computes the devices' positions, and delivers it whenever a user requests it. The positionning server keeps tracks of the devices and records positions even though the users didn't request it.

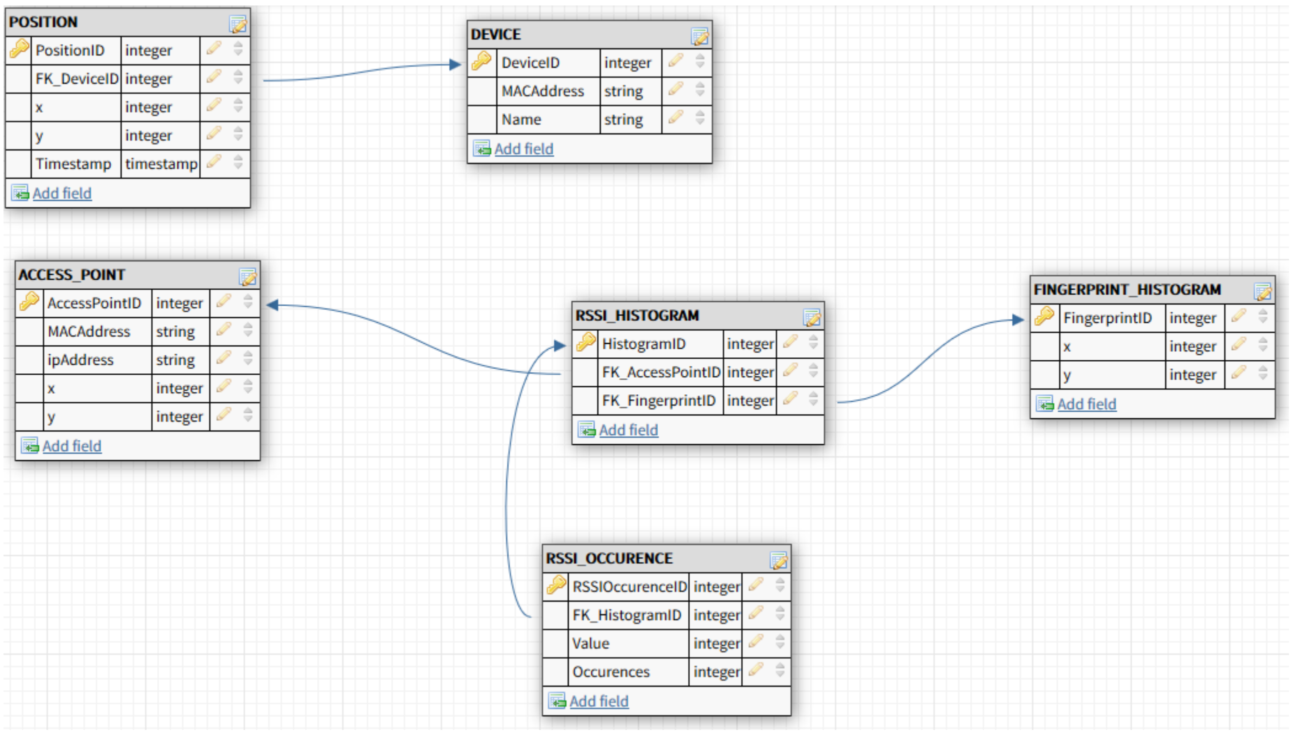
We decided to use a *tomcat* server with a *MySQL* database, we also used technologies such as *Maven* to manage our dependencies, *Hibernate* to fasten our manipulation of the database, *Javax servlets* to make an HTTP interface and *Timer Task* to run a scheduled job.


DatabaseService
+getAllPositionsFromDatabase(MACAddress:String): List<Position>
+getLastKnownPositionFromDatabase(MACAddress:String): Position
+saveUserIfNotExists(MACAddress:String): void
+saveUser(MACAddress:String): void
+getDevicesMACAddress(): List<String>
+getMap(): FingerprintMap
+getAccessPoints(): List<AccessPoint>
+getAllPositionsFromDatabaseSinceTimestamp(timestamp:Timestamp, macAddress:String): List<Position>
SmoothingPathService
+createPath(Device): void
+separateCoordinate(List<Position>): List<double[]>
JSONService
+buildPositionListJSON(List<Position>): String
PositioningService
+map: FingerprintMap
+computePosition(List<RSSIHistogram>,Device): Position
+getAllPositions(macAddress:String): List<Positions>
+getLastKnownPosition(macAddress:String): Position
+getAllPositionsScinceTimestamp(macAddress:String, timestamp:Timestamp): List<Position>
+compute_probability(List<Histogram>,List<Histogram>): double
+compute_probability_2histogram(RSSIHistogram, RSSIHistogram): double
+find_circle(position:Coordinate,size:int): List<HistogramFingerprint>
+find_index(Coordinate): int
PathServlet
+doGet(HttpServletRequest,HttpServletResponse): void
ServletListener
+context: ServletContext
+contextDestroyed(ServletContextEvent): void
+contextInitialized(ServletContextEvent): void
LocationServlet
+doGet(HttpServletRequest,HttServletResponse): void
TimerTaskScheduler
+schedule(): void
AccessPointServlet
+doPost(HttpServletRequest): void
+getNewDeviceMACAddress(HttpServletRequest): String
JobRunner
+USER_AGENT: String = "IPS_Server"
+run(): void
+getHistogramSamples(List<String>,List<AccessPoint>): Map<String, List<RSSIHistogram>>
+sendGetRSSI(String,AccessPoint): List<RSSIHistogram>
+getRSSIHistogramFromString(String): List<RSSIHistogram>
HibernateUtil
+sessionFactory: SessionFactory
+serviceRegistry: ServiceRegistry
+createSessionFactory(): SessionFactory
+getSessionFactory(): SessionFactory
+shutdown(): void

II – The database

a) Schema

To be able to know communicate with the known Aps, to compute the devices' positions, to register the devices' path and the devices themselves, we need to register many informations in the database. Here is the schema of ours:



POSITION	
PositionID	integer
FK_DeviceID	integer
x	integer
y	integer
Timestamp	timestamp

DEVICE	
DeviceID	integer
MACAddress	string
Name	string

ACCESS_POINT	
AccessPointID	integer
MACAddress	string
ipAddress	string
x	integer
y	integer

RSSI_HISTOGRAM	
HistogramID	integer
FK_AccessPointID	integer
FK_FingerprintID	integer

FINGERPRINT_HISTOGRAM	
FingerprintID	integer
x	integer
y	integer

RSSI_OCCURENCE	
RSSIOccurenceID	integer
FK_HistogramID	integer
Value	integer
Occurences	integer

The DEVICE table is used to register all the known devices that the APs previously posted.

We store all the computed location of the device in the POSITION table. If a device disconnects from the application and then come back late, we will recognize its MAC address and be able to keep recording the positions for this device. The position table just contains the coordinates of the device with a timestamp, to know when it was recorded. There is a one-to-many relationship with the DEVICE and the POSITION with the field "FK_DeviceID".

The FINGERPRINT table contains all the RSSI_HISTOGRAM associated coordinates. Each coordinates corresponds to several RSSI_HISTOGRAM, and each RSSI_HISTOGRAM is composed of several RSSI_OCCURENCEs, which are the values of an RSSI and the number of occurences it has in the histogram.

Each RSSI_HISTOGRAM comes from an ACCESS_POINT. An access point can have several RSSI_HISOGRAMs. In the ACCESS_POINT table, we also have some necessary information registered such as its ip address (so the server knows which addresses to reach in the timed job), and its position (mandatory in order to compute the devices' positions).

b) Interaction with the sever

To interract with the database, we are using the *Hibernate plugin*, which allows us to quickly query the database by mapping java classes to database tables (file terminating by *hbm.xml*). For example, to insert a device from the server into the table device in the database, we just have to instantiate the java object, and then save it by using a Session object (which is configured to be able to connect to our database with the *hibernage.cfg.xml* file). We don't even have to write queries to insert it. To extract info from the database however, we use the HQL query language, which looks like SQL but is more intuitive.

The class DatabaseService is a class that contains all the functions related to the communication between the server and the database. There is some functions to seed the database for testing purposes, such as the *seedDeviceAndPositions* method or the *seedFingerprint* method.

But the main methods of this class are the following:

- getAccessPoints, which gets all the access points from the database where the ip address is known, and returns it as a list of AccessPoint objects. It is used by the scheduled job to know which ip addresses to query
- getAllPositionsFromDatabase, which takes all the positions correponding to a device's MAC Address (given as parameter), sorted in ascendent order
- getLastKnowPositionFromDatabase, which returns the last known position (hihest timestamp) corresponding tu a device's MAC Address (given as parameter)
- getAllPositionsFromDatabaseSinceTimestamp, which executes the same things as the method above, but this time returns only the positions since a given timestamp
- saveUserIfNotExists, which takes as parameter a MAC Address. This function counts the number of times this MAC Address is in the Device table, and if it isn't there, it inserts this new MAC Address. This is the function that we use when an AP posts a newly detected device
- getMap, which is a very important method since it returns the whole fingerprintMap so we can compute the devices location. It is loaded when the JobRunner is instanciated
- getDevices, which returns all the known MAC Addresses of the devices. It is used by the JobRunner so that he knows which MAC Addresses' RSSI to ask to the AP

III – The communication interfaces

In order to have a positionning system, the server must coordinate all the other organs by communicating with them. We will see here how this is done.

a) Server to AP communication

1) The scheduled job

To know the RSSIs of the devices that the AP is receiving, we have created a class

JobRunner that extends the TimerTask class and that runs depending on a defined time interval. This JobRunner class is scheduled withing the TimerTaskScheduler class, which itself is instanciated by the ServletListener class. The ServletListener class implements the ServletContextListener class, and is able to be instanciated when the servlet context is initialized. To be more brief, this allows us to start the JobRunner automatically every time the server starts.

The role of the JobRunner class is to:

- Extract the known devices MAC addresses and the known access points' IP addresses
- Query one by one each AP to get the RSSI for the previously extracted devices MAC addresses. We decided to explicitely ask for the MAC addresses in the case where we don't want all the RSSIs. For example to have a maximum number of users to track...
- Get the results from the APs and build RSSI Histograms with it
- Compute the location of the devices with the PositionningService class
- Store the previously computed location in the database

We use and HttpURLConnection to query the APs, by writing a simple HTTP GET request to the AP's MAC Address, with the devices' MAC Addresses in the URL as parameters. The AP answers back with a JSON that is parsed with the help of the org.json package.

2) The Access point servlet

Whenever the access point detects a device that is not already known (via its MAC Address), it sends to the server and HTTP POST request with as parameter the MAC Address of the newly detected device. The server will then catch this request, get the MAC Address out of the query parameters – which are regular HTTP POST parameters, not JSON - and then register it to the database (with checking if this MAC Address isn't already registered).

To create this HTTP interface, we use the class AccessPointServlet, inheriting from the HttpServlet class. This servlet is mapped within the web.xml file, and will be called whenever a client requests the specified mapped address. The convention is that we call the method that intercepts the query *doX*, where X corresponds to an HTTP request method. In our case, it is called *doPost*. Before beeing registered in the database, we use a regex to check if the MAC address has the right format.

By beeing registered in the database, it now allows the scheduled job to ask the APs the RSSI of this new device.

b) Server to Android device communication

Because the scheduled job is running all the time, the locations of the devices are all already computed and registered in the database. The user can then connected to its app and request for its positions. For that there is two different servlet.

}

1) The Location servlet

The location servlet corresponds to the LocationServlet class, inheriting from the HttpServlet class. It allows the devices to query the "getLocation" URL (mapped in the web.xml file) with a simple HTTP get request and its MAC Address as a get parameter, and the server will respond with the last known location extracted from the database. It answers to the device with the following JSON string (as example):

```
{
      "x":100,
      "y":200,
      "timestamp":"2015-06-05 14:55:23.0"
}
```

Which corresponds to a simple position.

2) The Path servlet

The path servlet corresponds to the PathServlet class, also inheriting from the HttpServlet class. It allows the devices to query the "getPath" URL with also an HTTP GET method with as parameter also its MAC address and a timestamp.

If the user didn't put the timestamp parameter, the servlet will respond with an sorted list of all the positions corresponding to the requested MAC Address that are registered in the database, which gives a JSON like the following:

```
{"positions":
      [
            {"x":200,"y":200,"timestamp":"2015-06-02 17:24:56.0"},
      {"x":120,"y":170,"timestamp":"2015-06-02 17:24:57.0"},
      {"x":120,"y":100,"timestamp":"2015-06-02 17:24:58.0"},
      {"x":100,"y":100,"timestamp":"2015-06-02 17:24:59.0"}
      ]
}
```

However, if the user did put the timestamp parameter within the URL, it will return *only the positions since this given timestamp*. We decided to put a timestamp following this format "yyyy-MM-dd_HH:mm:ss.SSS" and not as a "long", because the "long" value depends on the epoch, which depends on the operating system (on UNIX, it is the 1st January 1970, on Windows it's the 1st January 1601), which is why we prefer to communicate with an explicit date which we will have to parse. For example, if the device requests:
http://ServerAddress/LO53_IPS/getPath?MACAddress=A1:B2:C3:D4:E5:F6&Timestamp=2015-06-02_17:24:58.0
It will return:

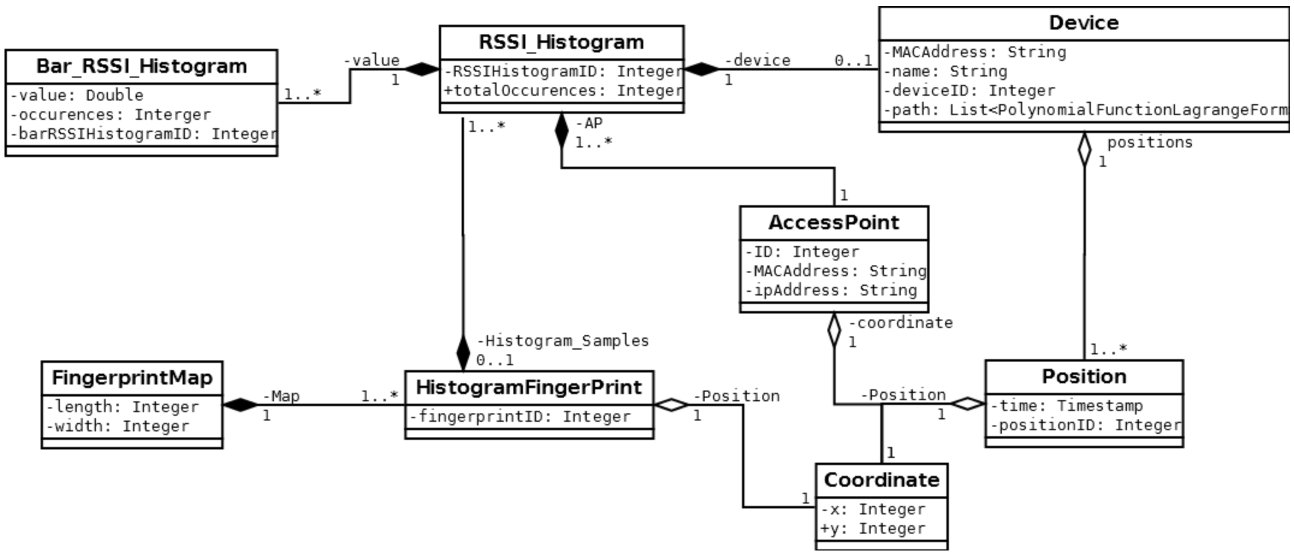
```
{"positions":
      [
            {"x":120,"y":100,"timestamp":"2015-06-02 17:24:58.0"},
            {"x":100,"y":100,"timestamp":"2015-06-02 17:24:59.0"}
      ]
}
```

This way, the android device can either ask for its complete path (because he just restarted the app for example), or only the path he made since a certain time.

IV – The location and path computations

The RSSI measurement isn't stable. To avoid this problem, we decide to compute a series of measurement on a histogram, this technique permit to calculate a probability for a panel of measurement of a device correspond to another panel of measurement of a fingerprint.

a) Data structures used



To represent the fingerprint map, we used following objects:

Bar_RSSI_Histogram corresponds to a bar of a histogram, contains a value of bar and number of occurrences.

RSSI_Histogram is a histogram complete, the totalOccurences permit to accelerate the computing of position who use many time this information. For each RSSI_Histogram, measurements corresponds at the same access point, this information is represents by an AccessPoint.

HistogramFingerprint corresponds to the vector of histogram for a point of FingerprintMap. This point is represent by a Coordinate.

FingerprintMap is a map contains all HistogramFingerprint.

To represent a device we used following objects:

Position is composed by a Coordinate and the associate time who the device was at this Coordinate.

Device is the represents of a device, it composed by a list of position, its name and its MACAddress. It have a list of PolynomialFunctionLagrangeForm to represents its path, each polynomial function corresponds to a part of device's path.

An access point is represents by an object with MACAddress, ID, ipAddress and a Coordinate.

The coordinate is just an object with two integers (x and y)

b) The location algorithm

The location algorithm is implements in PositionningService object, see the UML diagram in the part “The role of positioning service”

This algorithm search to have a great probability between a vector of RSSI’s histogram measure and a vector of RSSI’s histogram correspond at a Coordinate.

It used few functions:

- **find_index**: this function take a Coordinate, and give the integer correspond at this position in a list.

- **find_circle** : this function take a Coordinate, position, and an integer, I, and give a list of HistogramFingerprint correspond at all HistogramFingerPrint who are at I of distance to position.

- **compute_probability_2histogram** : this function take two RSSI_Histogram, histogram1 and histogram2, and run through the histogram and calculate for each Bar_RSSI_Histogram, value_histogram1_i and value_histogram2_j :
  ```
  prob = prob + Math.min( value_histogram1_i.getOccurences() /
  histogram1.getTotalOccurences() , value_histogram2_j.getOccurences() /
  histogram2.getTotalOccurences())
  ```

  prob can’t exceed 1 because the sum of all bar on an histogram divide by the total occurences is equal to 1. Prob it’s the output of function.

- **compute_probability**: this function take on input two list of RSSI_Histogram and start to compare their sizes, if sizes are different the probability is equal to zero because we have 0% of probability to find an histogram in an histogram null.

  We compare histogram two at a time, if we MACAddress are the same.

  If MACAddress is different we can stop and return 0% of probability, because the list of RSSI_Histogram haven’t detect by the same access point indeed it’s not possible to this position is the good one.

  If MACAddress is equal, we call compute_probability_2_histogram with this two RSSI_Histogram and we multiply actual probability by result of this function.

- **computePosition** : it’s the main function. It take a list of RSSI_Histogram, called vector_RSSI_Histogram, and a Device, called device. The goal it’s to compare this list with the HistogramFingerprint near the last position of the device.
  It start by use find_circle with I = 1, and use compute_probability to find the best probability if it find a probability superior to 0.95, it returns this position, else it use find_circle with I=2 and it restart.

c) The path smoothing method

The path smoothing method it used to create a true path between all positions and don’t have only a segment between each positions.

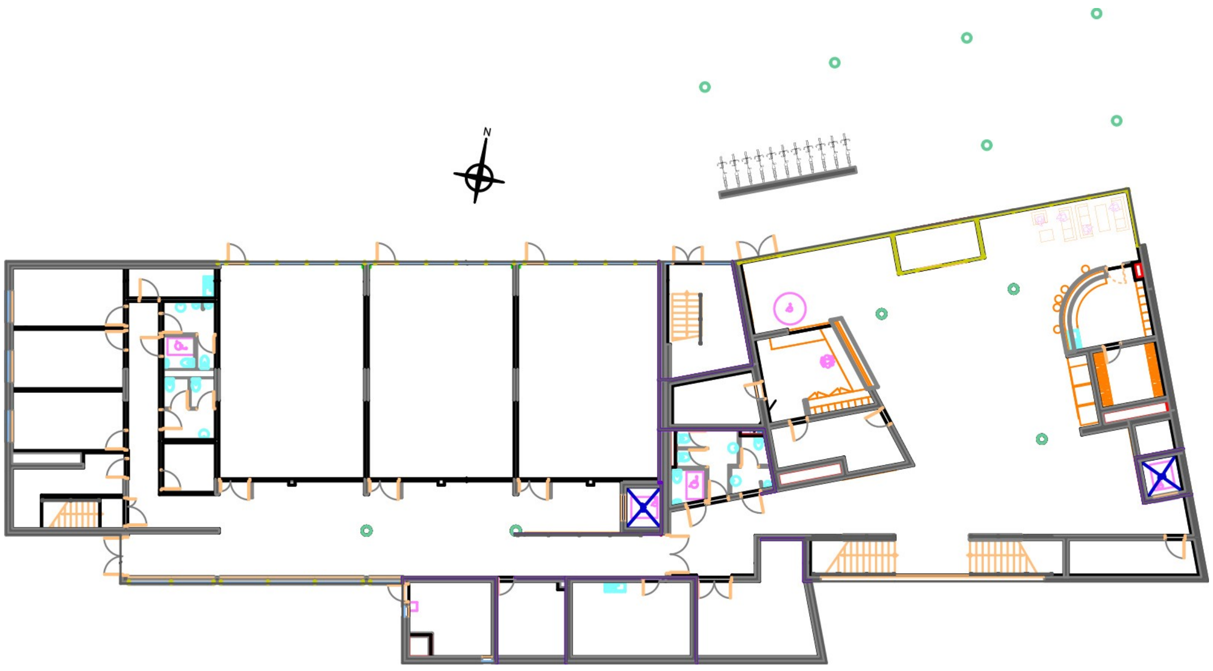
We have decided to use an interpolation polynomial with Lagrange formulas. All functions need to interpolate a polynomial function is in library Math3 of Apache.

To find polynomial function close to the real path, we create a polynomial function with three points (i, i+1, i+2) to create the path between (i, i+1). The end of path is create with only the two last points.

Android application

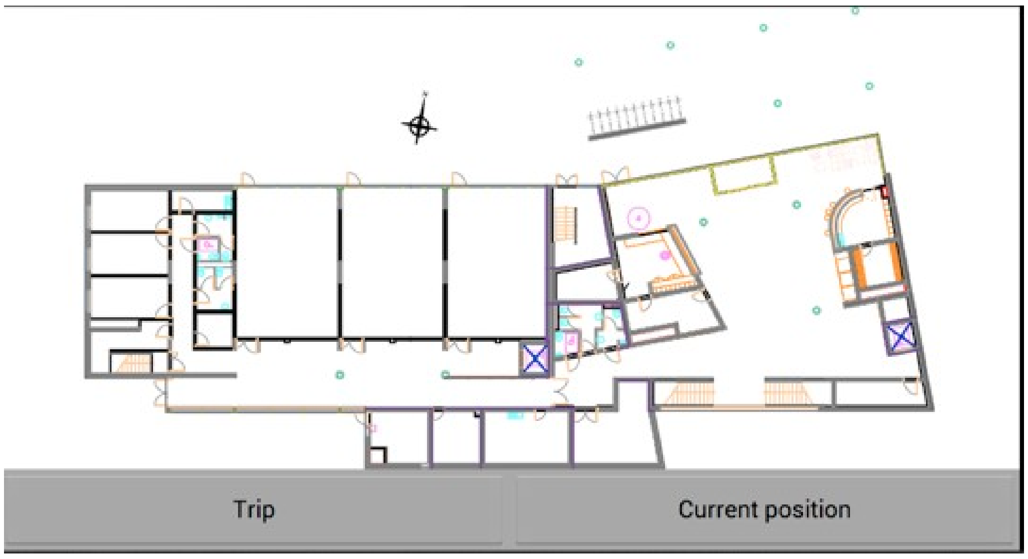
I – Principle

The Android application will request the path from the tracking start to the current time or the current position. For this project we will follow users in building H.



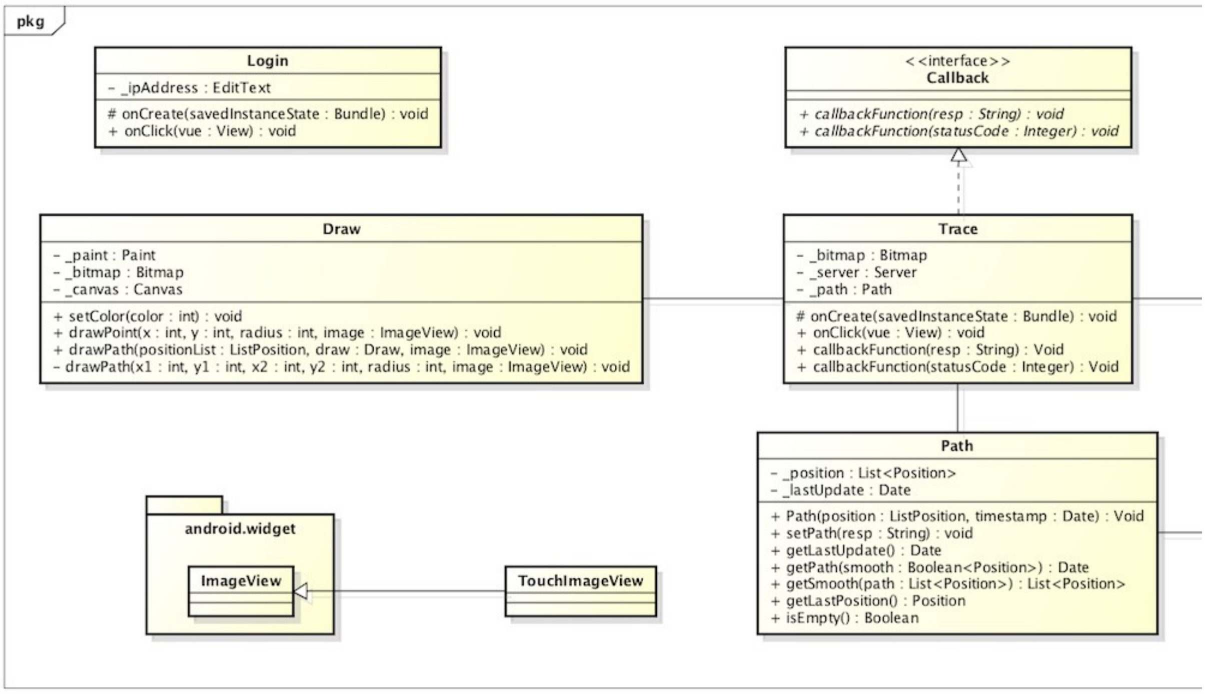
Building H

After the user is connected on the application he have to enter the ip server address, then he will found the map where he may choose two options: “Trip”, or “Current position”. The first one will draw on the map the path taken and the second one will draw on the map the current position of the user.

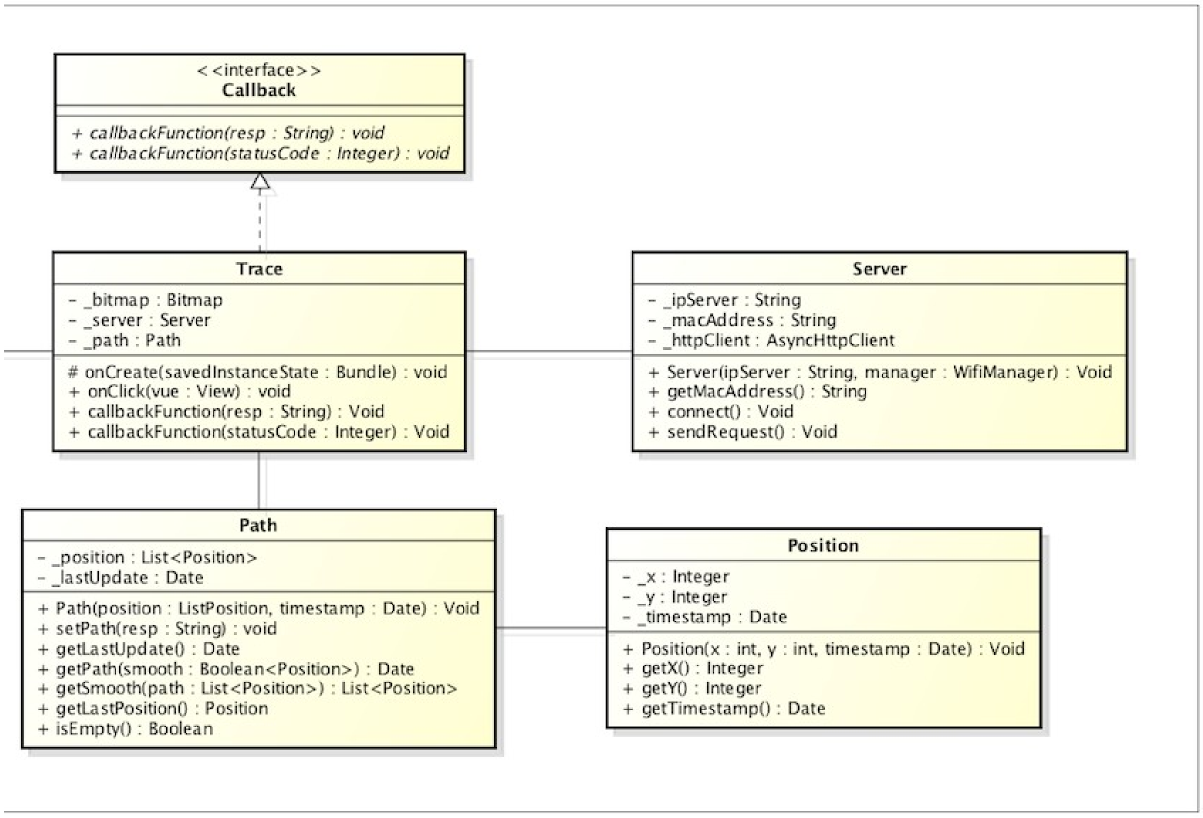


Main activity

II – Class diagram


pkg
Login
- _ipAddress : EditText
# onCreate(savedInstanceState : Bundle) : void
+ onClick(vue : View) : void
<<interface>>
Callback
+ callbackFunction(resp : String) : void
+ callbackFunction(statusCode : Integer) : void
Draw
- _paint : Paint
- _bitmap : Bitmap
- _canvas : Canvas
+ setColor(color : int) : void
+ drawPoint(x : int, y : int, radius : int, image : ImageView) : void
+ drawPath(positionList : ListPosition, draw : Draw, image : ImageView) : void
- drawPath(x1 : int, y1 : int, x2 : int, y2 : int, radius : int, image : ImageView) : void
Trace
- _bitmap : Bitmap
- _server : Server
- _path : Path
# onCreate(savedInstanceState : Bundle) : void
+ onClick(vue : View) : void
+ callbackFunction(resp : String) : Void
+ callbackFunction(statusCode : Integer) : Void
Path
- _position : List<Position>
- _lastUpdate : Date
+ Path(position : ListPosition, timestamp : Date) : Void
+ setPath(resp : String) : void
+ getLastUpdate() : Date
+ getPath(smooth : Boolean<Position>) : Date
+ getSmooth(path : List<Position>) : List<Position>
+ getLastPosition() : Position
+ isEmpty() : Boolean
android.widget
ImageView
TouchImageView

Class diagram - part 1


<<interface>>
Callback
+ callbackFunction(resp : String) : void
+ callbackFunction(statusCode : Integer) : void
Trace
- _bitmap : Bitmap
- _server : Server
- _path : Path
# onCreate(savedInstanceState : Bundle) : void
+ onClick(vue : View) : void
+ callbackFunction(resp : String) : Void
+ callbackFunction(statusCode : Integer) : Void
Server
- _ipServer : String
- _macAddress : String
- _httpClient : AsyncHttpClient
+ Server(ipServer : String, manager : WifiManager) : Void
+ getMacAddress() : String
+ connect() : Void
+ sendRequest() : Void
Path
- _position : List<Position>
- _lastUpdate : Date
+ Path(position : ListPosition, timestamp : Date) : Void
+ setPath(resp : String) : void
+ getLastUpdate() : Date
+ getPath(smooth : Boolean<Position>) : Date
+ getSmooth(path : List<Position>) : List<Position>
+ getLastPosition() : Position
+ isEmpty() : Boolean
Position
- _x : Integer
- _y : Integer
- _timestamp : Date
+ Position(x : int, y : int, timestamp : Date) : Void
+ getX() : Integer
+ getY() : Integer
+ getTimestamp() : Date

Class diagram - part 2

Since the last report the class diagram was updated, we can found four new classes implemented: *Login, Draw, TouchImageView, Trace* and an interface: *Callback*.

III – Development

The *Login* class will appear on the first connection, and require the ip server address. If this one is correctly set the Login class will call the main activity: Trace.

The *Position* class is the location (x,y) in the map at a certain time for this specific device.

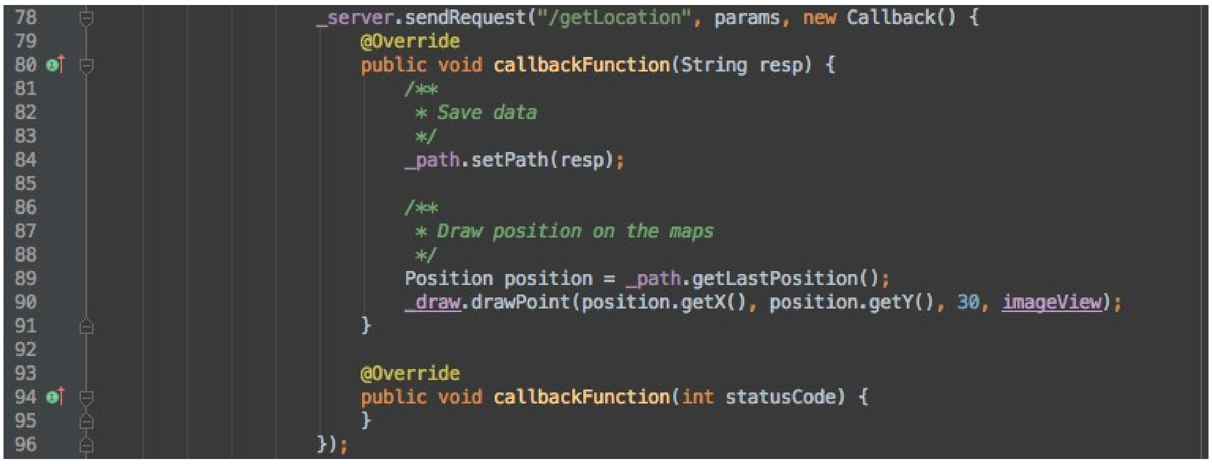
The *Path* class is nothing more than a list of these positions through time with the last update time.

The *Server* class is the link between the application and the server. It will send HTTP request to the server in order to get the tracking of the mobile or the current position. This class have to main function:

- connect(), send a request in order to test the connection with the server
- sendRequest(), send a request to get the current position, the full path or an update of the path. On each request the client will send in parameter his mac address.

The *Draw* class paint on the map the full path or the current point.

The *Trace* class is the main class of the application. Based on the choice of the client, this function will send the right HTTP request. Moreover the HTTP request are asynchronous so after sending the request, the android application will continue his execution. When the server will return his answer, the android application will call the callback function defined in the main function.

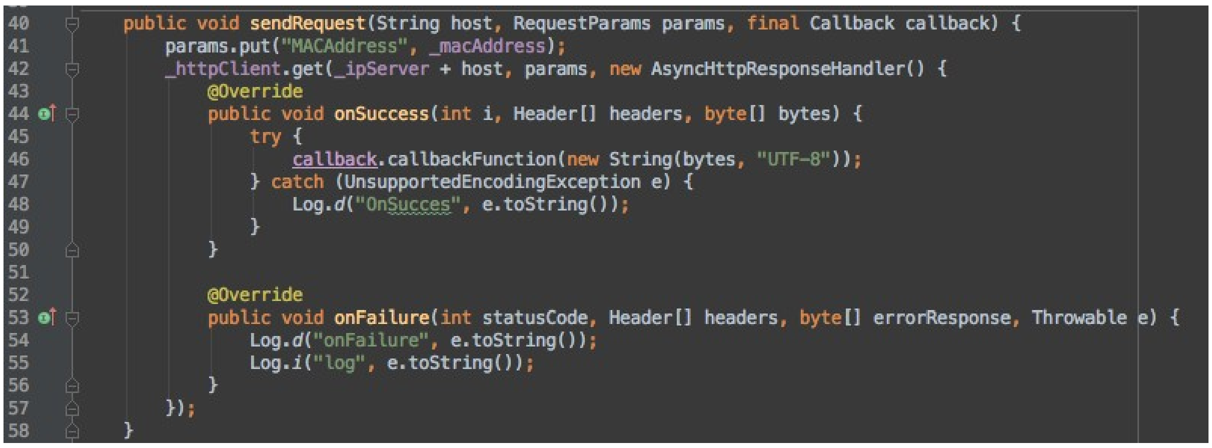


```
_server.sendRequest("/getLocation", params, new Callback() {
    @Override
    public void callbackFunction(String resp) {
        /**
         * Save data
         */
        _path.setPath(resp);

        /**
         * Draw position on the maps
         */
        Position position = _path.getLastPosition();
        _draw.drawPoint(position.getX(), position.getY(), 30, imageView);
    }

    @Override
    public void callbackFunction(int statusCode) {
    }
});
```

Class Trace - sendRequest function with callback function

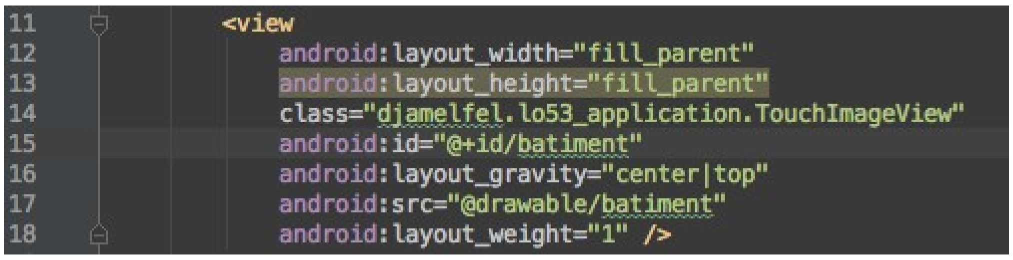


```java
public void sendRequest(String host, RequestParams params, final Callback callback) {
    params.put("MACAddress", _macAddress);
    _httpClient.get(_ipServer + host, params, new AsyncHttpResponseHandler() {
        @Override
        public void onSuccess(int i, Header[] headers, byte[] bytes) {
            try {
                callback.callbackFunction(new String(bytes, "UTF-8"));
            } catch (UnsupportedEncodingException e) {
                Log.d("OnSucces", e.toString());
            }
        }

        @Override
        public void onFailure(int statusCode, Header[] headers, byte[] errorResponse, Throwable e) {
            Log.d("onFailure", e.toString());
            Log.i("log", e.toString());
        }
    });
}
```

Class Server - sendRequest function which will call the callback function when the server will respond

The *TouchImageView* class is implemented in order to enable zoom function on the map, as following.



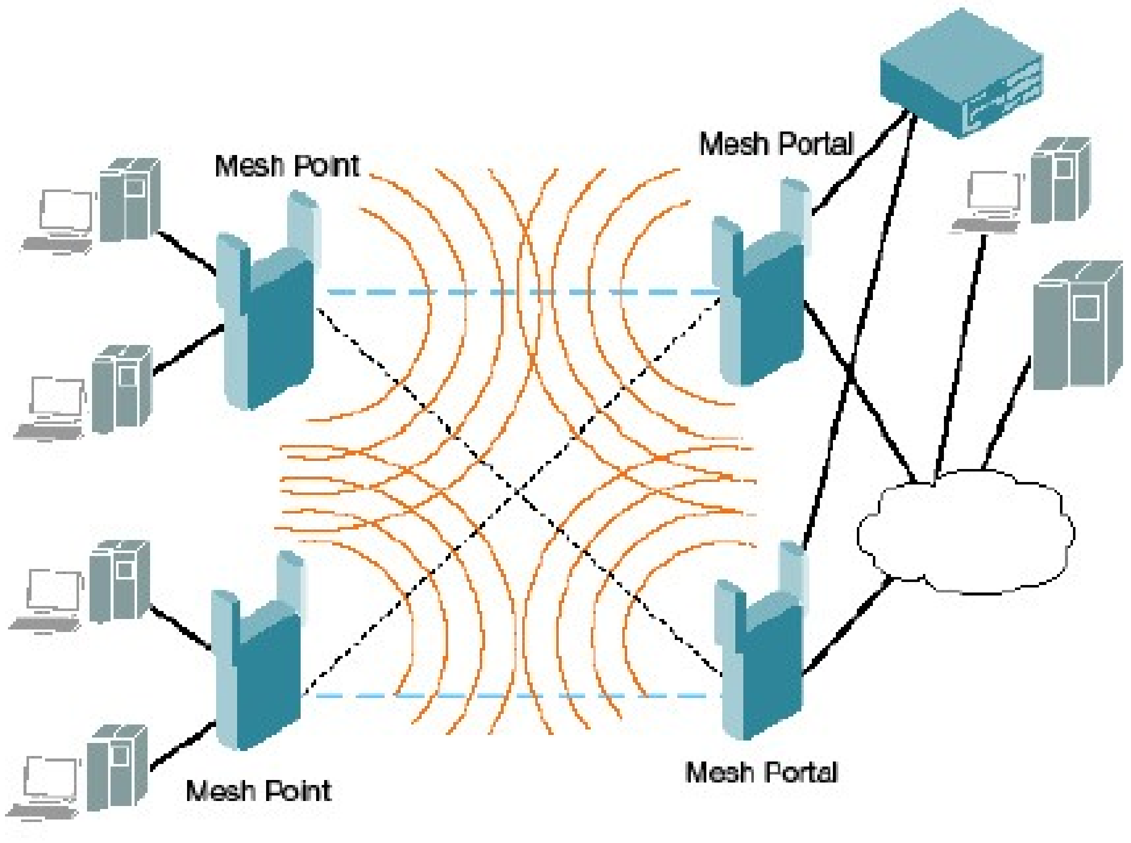
```xml
<view
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    class="djamelfel.lo53_application.TouchImageView"
    android:id="@+id/batiment"
    android:layout_gravity="center|top"
    android:src="@drawable/batiment"
    android:layout_weight="1" />
```

Access points

I – Principle

The access points are the core of the project in two ways. They are the mean used to create a wireless network so devices can connect to the network and to the server, but they also take periodically the RSSI meausrements needed to compute the position of those devices. Since the access points, TP-LINK TL-MR3220 and TL-WDR3600, are embedded platforms with relatively low memory and CPU power they will not compute a single thing but rather send raw data to the server.

The goal of this project is to track a device thanks to WiFi in a medium-size area. Because of that, the access points should be linked together to form a network with the help from mesh protocols thus avoiding all of them to be linked with long network cables. The mesh protocol is managing an access point to access point network. To do that, each access point being a node in the mesh protocol have to emit and receive in at least two frequencies: one which will be used to carry data like a basic access point would do and an other one which will maintain a connection with one or several other mesh nodes. The mesh network will also find automatically the best, the shortest or the most stable path to reach one device and update this path when one is better or if the current path doesn't exist anymore (due to interferences or a broken access point).



There are actually two ways of tracking a device: asking the device the permission to locate it or taking it as granted once the WiFi module on the device is on power.
We chose the second option to demonstrate how easy it is to be tracked without our consent and especially without knowing it.

II – Working of the access point

a) Summary

All access points are reachable in the network and are connected to the main server. They are continuously sniffing WiFi packets and storing the RSSI values of each intercepted packet. When the main server asks the access points what are the RSSI measurements for a certain MAC address, they return the recorded values. When a new device is detected by an access point, the latter will send a message to the server so it can acknowledge a new device to track.

b) Precise definition

The program is divided in three different threads: one for the packet sniffing, one for an HTTP server waiting a request from the main server to send RSSI measurements and an other one for an HTTP client, sending requests to the main server whenever a new device is detected in range of this access point. The main thread will create the thread in charge of the HTTP server, then create the HTTP client and will finally sniff WiFi packets.

1) Packet sniffing

WiFi packet sniffing can be done via the libcap[1] C/C++ library available on all platforms and used by some famous project like Wireshark. To sniff packets, a NIC (Network Interface Controller) can be in two modes: promiscuous and non-promiscuous. The first one will sniff every packet from any device to any device on the mentioned interface (if WiFi, all WiFi packets even from different networks will be caught). The second one will catch packets coming from the interface, to the interface or passing by the interface. Since we want to discretely track any device, the promiscuous mode will be chosen. However, you have to keep in mind that in a crowded environment or in high traffic networks, the device sniffing packets will catch a lot of packets and thus, might need a lot of resources to handle all this data. Moreover, the promiscuous mode can be detected by the device[2].

First opening the device to sniff is required for the library to work:

```
pcap_t *pcap_open_live(char *device, int snaplen, int promisc, int to_ms,
             char *ebuf)
```

It will return a kind of a stream we can listen to. The first argument is the interface we want to listen, then the size of the buffer for the packet sniffing, enabling the promiscuous mode, the read

1 http://www.tcpdump.org/
2 http://en.wikipedia.org/wiki/Promiscuous_mode#Detection

time and finally an error buffer.

Then we will get the first packet in the stream of caught WiFi packets thanks to:

```
u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h)
```

It will return a pointer to the sniffed packet. The first argument is the “stream” gotten with pcap_open_live(), then a pointer to the structure that will store the header data of this particular packet.

Thanks to information contained in the header, we can find the position of the MAC address of the sender and the RSSI value for this packet. After that the functioning is straight-forward, either:

1. the MAC address is already known to the program so we just have to add the current RSSI values to its RSSI histogram,
2. or, the MAC address is unknown to the program, we have to create a new device, insert it in the list of devices, create a new histogram and add the current RSSI value to it. However, we also need to notify the server that a new device has been detected so we send this information to the HTTP client thread.

2) Respond to HTTP requests

The access point has to listen continuously on HTTP port in order to receive HTTP requests from the main server. To avoid reinventing the wheel, we used one of the few HTTP server available as a library for C projects: MicroHTTPd[3].

MicroHTTPd is actually implemented as a daemon so you have to launch it with:

```
struct MHD_Daemon * MHD_start_daemon (unsigned int flags, unsigned short port,
      MHD_AcceptPolicyCallback apc, void *apc_cls, MHD_AccessHandlerCallback dh,
      void *dh_cls, ...)
```

It will return the created daemon so we can stop it whenever we decide to close the program. The first argument is an OR-ed combination of flags, then the port on which the daemon will listen, the callback function to check if a user is allowed to access the server, the arguments for this callback function, the function handling all URIs and the arguments for this function.

Therefore we need to write the function which will handle the URIs. Its skeletton should look like:

```
int *MHD_AccessHandlerCallback (void *cls, struct MHD_Connection * connection,
      const char *url, const char *method, const char *version, const char
      *upload_data, size_t *upload_data_size, void **con_cls)
```

It should return the integer associated to the HTTP status code (404 for not found, 200 for successful). The first argument is the last argument from `MHD_start_daemon`, then the URL requested by the client, the HTTP method used (GET, POST, PUT...), the HTTP version, the uploaded data (for POST requests), the size of the uploaded data and a pointer passed to every call of this function.

3 https://www.gnu.org/software/libmicrohttpd/

Since the main server will send an HTTP request of the following form:

```
http://[ip_access_point]:[PORT]/?mac_addrs=[addr1],[addr2],[...]
```

We have to parse the URI to get the URL parameter after "?mac_addrs=". It is done by calling:

```
const char * MHD_lookup_connection_value (struct MHD_Connection *connection,
enum MHD_ValueKind kind, const char *key)
```

It will return all the data after the string defined by `key` for a GET HTTP request when `kind` is set to `MHD_GET_ARGUMENT_KIND`.

Then we separate each MAC address from each other and look for their respective RSSI histograms. Before doing the latter, we need to delete all outdated values in the RSSI histogram or the results would be distorted. After all the data have been gathered, a JSON answer is created and sent to the main server as a GET HTTP response. This JSON looks like:

```
{"mac_ap":"90:BB:6E:0D:5A:6C",
 "results":[
      {"mac":"6F:A9:0D:03:8C:6C","histogram":
            [],"total_occurences":0},
      {"mac":"FD:AB:AF:A6:32:EB","histogram":
            [{"rssi":2.511886,"occurences":6}],"total_occurences":6}
      ]
}
```

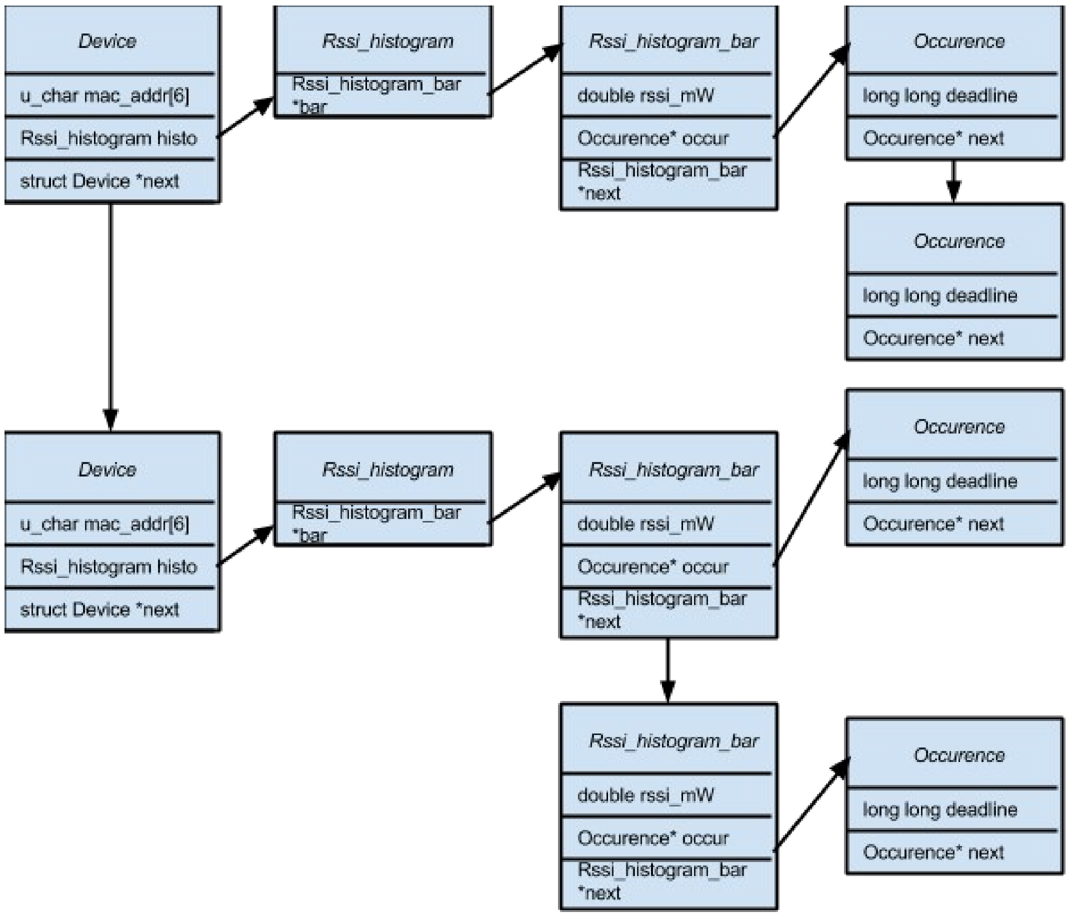
3) *Notify main server of the detection of new devices*

When the thread in charge of sniffing WiFi packets detects a new device, the HTTP client thread needs to send this information to the main server. This can be done using cURL library for C projects: libcurl[4]. The implementation is straight-forward: you initiate a connection with `curl_easy_init()`, set the URL with `curl_easy_setopt(curl,CURLOPT_URL, "http://192.168.2.150/")`, set the POST data with `curl_easy_setopt(curl, CURLOPT_COPYPOSTFIELDS, post)` and execute the request with `curl_easy_perform(curl)`. The POST data is the following: DeviceMACAddress=XX:XX:XX:XX:XX:XX.

4 http://curl.haxx.se/libcurl/

III – Structures and global variables

a) Structures



Device contains its MAC address, its associated RSSI histogram and the next device in the list.

Rssi_histogram contains a pointer on the first bar of the histogram.

Rssi_histogram_bar contains the RSSI value associated to the histogram bar, a pointer on the first recorded occurrence for this RSSI value and a pointer on the next histogram bar.

Occurence contains the time when the recorded occurrence is outdated and a pointer on the next occurrence.

When a new device is detected, it is added to the devices' list. When a new packet is caught for a known device, we add this value to its RSSI histogram. Either it already has a bar for this RSSI value and it just needs to add it at the end of the occurrence list or it needs to add a new bar at the end of the list of bars.

b) Global variables

- **sig_atomic_t got_sigint**

When interrupting the sniffing thread, we can safely close the pcap “stream” but we cannot safely close neither the MicroHTTPd daemon nor the cURL pointer. That's why we need a shared variable every thread will check at each round of their infinite loop. When the sniffing thread will receive an interrupt, it will set this variable to 1, telling the other threads that they have to stop working and safely close everything.

- **Device** device_list**

This is a pointer on the first device of the list of devices. Its access is protected by a semaphore. The sniffing thread puts all the data from the sniffed packets in this list and the HTTP server thread get information from this list and send it to the main server.

- **sem_t synchro**

This is the semaphore used to protect access of the devices' list.

- **int msqid**

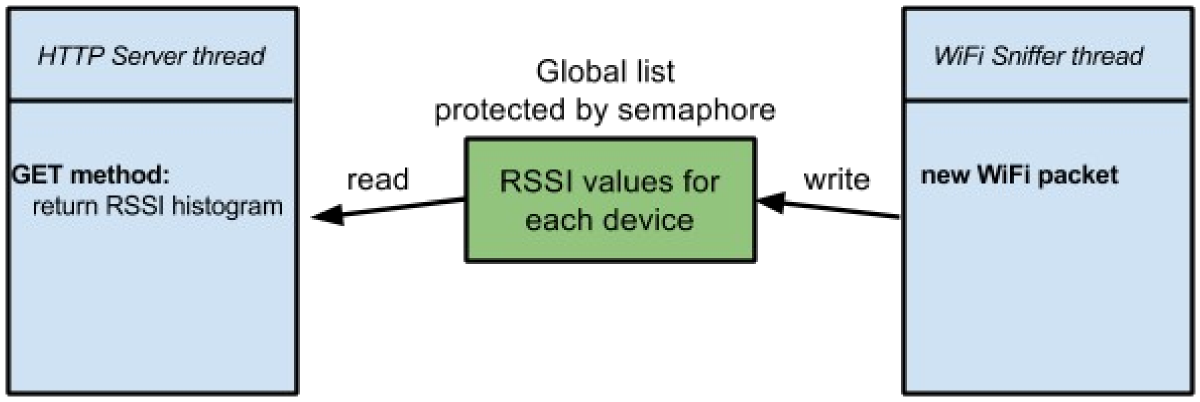
This is the identifier of the message queue used by the sniffing thread to notify the HTTP client thread that a new device has been detected and by the HTTP client thread to get the MAC address of the newly detected device.

IV – Interthread communications

The three threads in the access points need two interthread communications: one for sharing devices' list and their RSSI histogram between the sniffing thread and the HTTP server thread, and one for sharing new devices' MAC address between the sniffing thread and the HTTP client thread.

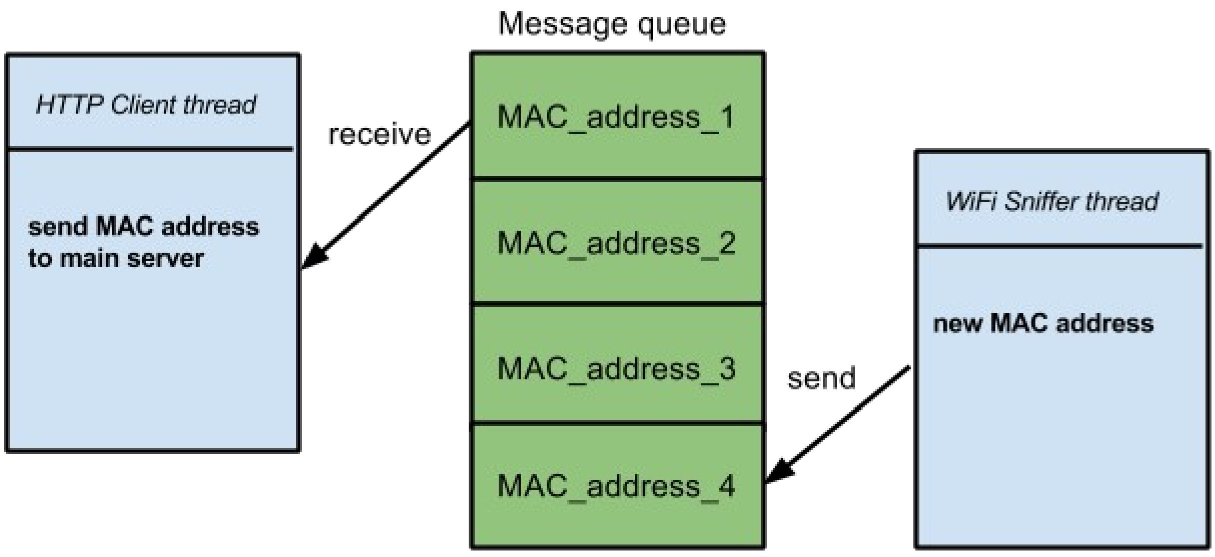
a) Sharing devices' list

Since the HTTP server thread needs to access to devices' list only when the main server is asking for data, which is far less frequently than the sniffing thread and that it needs the exact current RSSI histogram at a certain time, the devices' list is safely shared thanks to a semaphore. The thread in need for this list will look at the value of the semaphore, if it is blocked it waits until the thread using it frees it, if it is free then it blocks the semaphore and frees it when computations are done on the list.



b) Sharing new devices' MAC address

The HTTP client thread needs to send a POST request each time a new device is detected by the sniffing thread. Otherwise it is just waiting for a new device to be detected, doing nothing meanwhile. Message queue is perfect for this situation. When a new device is detected by the sniffing thread, it adds a message to the message queue with the MAC address of this new device. It can add as many messages as it wants. When a new message is put in the message queue, the HTTP client thread will get the first message in the queue and send it to the main server via a POST HTTP request.



Fingerprint map generated by machine learning with an artificial neural network

To be able to locate the user equipment in the building, we need a fingerprint map which contain the RSSI vector at the given point. There is several way to get this map. First there is the empirical way by doing measurement for each location, this would be quite long. Another way to get this resource is to generate it. We can do it using different methods, but for this project we choose to use a machine learning tool : the artificial neural network.

The idea is basically to give to the network the distance between the access point and the user equipment, and also the number of wall crossed by the signal if it was a straight line. From those data the algorithm will learn to return the right RSSI according to the different input which can be given to it.

In order to do it, we used two different library: FANN and GDAL. The FANN library allow me to create, train and use artificial neural network. GDAL is used to read the plan of the building and find how many walls would be crossed by the signal.

I – Parts of the fingerprinting system

There are several parts in this system that we had to implement. In this part I'll describe those part, why we should do it and how the functionality is working.

a) Getting the distance between the access point and the user equipment

The fingerprint is a grid defined by two points (the corners of the rectangle) and the steps between points of this grid. We implemented a function that give the integer dimensions of the fingerprint ans another that convert a point with integer coordinate in a real coordinate point. Form there, the distance is found by a simple Pythagorean theorem application between two real coordinates points.

b) Getting the number of wall crossed by a signal between the access point and the user equipment

For this part, we first had to get the plan of the building in the shape file format. With the GDAL library we could open this file and get all lines of the plan. This library has a lot of features, and one of them is to check if two geometry (the lines here) are intersecting or not . So from this point it become easy to create the line between the access point and the user equipment and then check if each wall is crossing it to count how much walls are crossed.

c) Creating the training file

In order to train the artificial neural network we need to create a training file. This file is formated in a specific way to let the FANN library parse it and use it properly. This file should have as first line, the number of entry followed by the number of input and the number of output. After that, there is one input per line and then one output per line formated as following :

```
number_of_entries number_of_input number_of_output
input
input
output
input
input
output
…
```

In order to have this file we should make some measurement with the access points and with the two previous features we can compute the distance and the number of wall crossed. From there we just need to print the right data at the right place in the file.

d) Train the network to create the model

The FANN library has a lot of features and one of them is to create an artificial neural network and to train it. There is some parameters to choose like the number of input and output, the number of layer, the number of node in the hidden layer, the number of training iteration and the desired precision.

All those parameters were simple to choose except for the number of node in the hidden layer which is chosen by testing according with the documentation of the library. Also the number of training iteration were chosen randomly. But with this function a model file is created and it can be used to create an affective neural network to be applied.

e) Apply the model for a point

To apply the model previously trained, the FANN library is really easy to use. We just have to create an artificial neural network using the file generated during the training and run it with the given input. As this model is used to generate the RSSI received by an access point for an user equipment located at a precise point, we have to apply it as much as we have access points for each point of the fingerprint map.

The algorithm to generate the fingerprint map is separate from all those features and just call them at the required moment. For more detail let's see the source code in the files “fingerprint_functions.c” and “fingerprint_functions.h” .

II - Problem encountered in the development of this part

This part of the project was based on research, we had to find the right libraries, the right data and the right files to use.

Finding libraries were not really difficult, but there were some troubles in their references manual. We had to contact the developers of the GDAL library to be able to find the right documentation. That made us losing a lot of time because at the first troubleshooting we thought that it was a compilation problem or an operating system linked problem.

Another problem was to get the plan of the building in the shapefile format. Actually we had and DWG format from Autocad and we were unable to open it and use it to get data from it. Finally we found a way to convert from a DXF format to the Shapefile format and by digging a little bit more we found that we could convert DWG in DXF with a well known vectorial image editor.

And finally the main problem was that it was quite difficult to do measurement with the access points and without them real testings were impossible. To test the system we will use the Friss formula to see if the neural network can return logical values or not but that will test only the distance part of the artificial neural network because this formula doesn't take walls in count.

Conclusion

We used a lot of tools and libraries we never used before. We also discovered that any packet can be intercepted without us knowing it which is important nowadays with the respect of privacy. We learned how to cross-compile for the given access point.

However, the RSSI values given by the access points are wrongly computed and therefore we couldn't deeply test our project in real conditions. Everything is working apart and together but it lacks practical data to create the fingerprint and locate users on a map. This could be fixed by using another library, used by the reknown aircrack-ng.

Finally, time was missing at the end to fully finish the path smoothing computation. The server is computing the polynomial function associated to the smoothed path but the Android application is actually not using it.