Project Report

Ubiquitous and Mobile Computing - 2015/16

Course: MEIC

Campus: Alameda

Group: 16

Name: Sérgio da Silva Mendes Number: 84955 E-mail: smendes_93@hotmail.com

Name: Diogo Manuel Perestrelo Abreu Number: 84952 E-mail: diogo.abreu2012@gmail.com

Name: Carlos Sérgio Figueira Faria Number: 84936 E-mail: carlosfigueira@tecnico.ulisboa.pt

1. Achievements

Version	Feature	Fully / Partially / Not implemented?
Baseline	Send / receive points between mobile devices	Fully
	Send / receive text messages between mobile devices	Fully
	Register user	Fully
	Send new trajectory	Fully
	Show the most recent and past trajectory on map	Fully
	Get user information	Fully
	Get list of stations with bikes available	Fully
	Book bike at specific station	Fully
	Handle bike pick up / drop off events	Fully
	Detect what bike is in use by each user	Fully
Advanced	Security	Fully
	Robustness	Fully

2. Mobile Interface Design

When the user opens the application he is prompted with the login where he can Sign In if he already has an account or Sign Up if doesn't have the account. If the user clicks Sign Up, he will go to another activity where he will enter his personal information that will be used to register him with the server. Here he will also choose a Secret Key that is going to be shared with the server for validating points sent by the user.

When the user Signs in or registers successfully he is taken to the Station screen where he is presented with the closest station to him and where he can easily book a bike. We chose it to be this way since this an application to be used while cycling, it makes sense that the user can easily book a bike and also get that station that is nearest to him. If he wants to chose another station, he can click the tab Others that will list others stations ordered by proximity to the user.

For easy access to other screens, we provide a navigation drawer where the user can click in Friends (explained in more detail in Section 3.5) to exchange points or messages or in History where he can check all the previous trajectories he has completed. In the navigation drawer, user related info is also displayed, like his current points for example.

3. Baseline Architecture

3.1 Data Structures Maintained by Server and Client

We believe that non relational database (MongoDB) would be a good choice to store information regarding **trajectories**, **stations** and **user information**, due to high consistency and partition of MongoDB by default (CAP theorem), and of course, NoSQL data sources, that offer a flexible client API that can save tons of time while implementing. For storing our database we opted to use a PaaS and mLab.

We have three collections: **user**, **station** and **trajectory**. And the properties for each collection are related with a POJO object in the server, explained in section 5.1, related to server implementation.

We use two different data structures in the application, **SharedPreferences** and **SQLite**. In Shared Preferences we choose to store simple data and information that is frequently accessed so don't we have the additional overhead of accessing SQLite. We store user related information: his server id, his first name, his last name, his email, his points and his secret key.

For the SQLite, we choose to store information that can become quite large and would not be feasible to store in Shared Preferences. We have 4 tables: ***chatTable:*** Stores all the messages exchanged with each friend; ***pendingPointsTable:*** Stores all the points that are waiting for validation. When the user comes online we send all the points in this table to the server for validation; ***pendingTrajectory:*** Stores all trajectories done by the user, how many points the user gained per trajectory, each distance and travel time.
This table has a column isPending that indicates if this trajectory is still pending to go to the server or not. When the user goes online, we perform a query on all pending trajectories to send them to the server;
stations: This table stores information about the stations, their location, their bikes available, coordinates;

3.2 Protocol for Bike Pick Up

Before the user can pick up a bike, he needs to make a reservation. If the reservation is made when the user reaches the station, he can go to a bike and start travelling. No distinction is made between bikes, so the decision of which bike to use is made by the user. When a reservation is made, the number of bikes available in the station is decreased. So, a bike pickup is detected when the following conditions are met:

1. The user has made a reservation;
2. The user is in a station (detected by the proximity of a beacon representing the station);
3. The user has reached a bike (detected by the proximity of a beacon representing the station);
4. The user isn’t currently travelling.

The detection is made by a broadcast receiver (***SimWifiP2pBroadcastReceiver***) that detects and processes broadcasts sent by the Termite API.

3.3 Protocol for Trajectory Tracking

When the user picks up a bike, the system triggers an intent that starts the activity ***MapsActivity***. This activity will display the user’s current trajectory, travel time, travel distance and points earned. Points are calculated by doubling the user’s current travel time in meters. Until the user returns to a station with a bike, the system assumes that the user is travelling and so keeps tracking the user’s trajectory.

3.4 Protocol for Bike Drop Off

A bike can only be dropped off if the user has previously gone through the process of reserving and picking up a bike, allowing him to be considered by the system as travelling. If the user is currently travelling, a drop off is equivalent to reaching a station with a bike. When this happens, the user is no longer considered to be travelling by the system, and all the trajectory information is stored. The end of a trajectory is detected when the following conditions have been met:

1. The user is in a station;
2. The user has a bike;
3. The user is currently travelling.

The user can now leave ***MapsActivity*** by returning to the previous activity. This is done so that the user can review his trajectory on the map.

3.5 Protocols for Interaction between users

In ***Friends*** we have two tabs, the first one, displayed by default, shows a list of friends that are currently within the same *Termite* group. If a user clicks in one of the friends displayed in this list, it leads the user to a private chat (***ChatActivity)*** with the friend he clicked. In this activity, it's displayed to the user all the previous messages exchanged with that friend and the user can send a new message to that friend or send points to him.

We also have another tab called ***Conversations*** where it's displayed to the user all the conversations he previous had with his friends. In each entry, it's displayed the name of the friend, the date of the last message exchanged within them and the last message exchanged within them. The user can click in one of the chat entries and this will lead him to the ***ChatActivity*** mentioned in the previous paragraph. Still in this tab, in the bottom right corner, there’s a button the user can click that will allow him to choose to send points to someone in the same *Termite* group (***SendPointsTo)*** or send a message to someone in the same *Termite* group (***SendMessageTo).***

3.5.1 Protocol for point sharing

Regarding point sharing between bikers, points can only be shared if they are within the same *Termite* group otherwise they cannot send points to other bikers. In order to simplify the life of the application user, we have an activity called ***SendPointsTo*** and ***Friends*** that presents a list of nearby bikers (in a *Termite* group with the application user) that shows to whom the user can currently send points. The application also has a chat (mentioned above) where the user has the option to send points to whom he is currently chatting with, however if the person who he is chatting leaves the group, an error message will be displayed informing the user that he can’t send points because the person he wants to send points to, is not in the *Termite* group.

We only allow the user to send points if he has enough points to do so. If he tries to send more points than he has, or a negative amount of points, the application displays a message to the user informing him that the points were not sent and why.When the user successfully sends points, the application immediately updates his current points. When receiving points however, it only updates the points after successfully validating the points with the server (explained in more detail in **4.1).**

3.5.2 Protocols for Message Exchange between Bikers

Messages can only be exchanged between users that are within the same *Termite* group. As with exchanging points, we also help the user to know to whom he can send message to by showing in the activity ***SendMessageTo*** and in ***Friends*** as mentioned above**.** When a user sends a message to a friend, only that friend receives that message and a chat (***ChatActivity)*** will be created between them in each device. The users can then to go to the chat the see all the messages exchanged with that friend.

4. Advanced Features

4.1 Security

For security we provide integrity, freshness and authentication guarantees therefore accomplishing all the security requisites which were, avoid tampering, replay attacks and not allow a user to resend received points on behalf of another user. In the Appendix, in figure 1 we illustrate the exchange of messages to properly validate points.

To achieve these three security properties, when a user sends points to another user, he doesn’t just send the points, he also sends a unique timestamp, his identifier, the receiver name, and the HMAC of all that info, in other words he sends: **<SenderID, ReceiverName, Timestamp, Points, HMAC(SenderID, ReceiverName, Timestamp, Points)>**.

The HMAC is calculated on the concatenation of all those four elements mentioned previously using **HmacSHA1**.The Secret Key chosen by the user will be used as the key to create the HMAC. When the user registers, the timestamp is also initialized and is incremented each time the user sends points therefore being unique for each points sent.

As for the validation, when the user receives, it doesn’t immediately update his points locally, he only updates after the server validated the points. Therefore, when the user receives points he forwards what he received to the server for validation. To perform the validation, we perform several checks:

1. Check if sender and receiver exist in the server database;
2. We then retrieve the Secret Key of the sender from the database and use it to compute the HMAC on the tuple of data received <SenderID, ReceiverName, Timestamp, Points> and then we compare with the HMAC also received by the server. If both HMACs match, then we proceed to the next step;
3. We then perform the final validation; we check the timestamp. We only accept points one time per timestamp, the server has an entry containing all the points sent by a specific user and with what

timestamp. In other words, when we validate the timestamp, we check if the server previously saw that timestamp, if it saw then it doesn't pass the validation, if it's the first time the server see this timestamp for this user, then it passes the validation.

After 3. succeeds, an acknowledgment is sent to the receiver of the points acknowledging him that the points were successfully validated and he updates his points locally. In the server side, we updated the timestamps received by the sender of the points and we update the points of both the sender and the receiver.

4.2 Robustness

As mentioned earlier, the application is in charge of keeping the scores updated. Since the application has this responsibility, the points are always consistent, the user can never send points that he doesn't have, for example.

Let's imagine the case that A has 50 points and the server knows that A has 50 points. A sends 50 points to B and then cycles and gains 30 points and sends those 30 points to C. If B and C go online before A, they will update the server and the server will have -30 for A. This is not a problem so we accept that A will have -30 in the server and why is it not a problem, because like mentioned, the application always has the amount of points the user really has at the moment, so in this case A will have 0 points. The application doesn't rely on the server for consistency of the points. When later A goes online, the server will change the amounts of points A has to 0 or anything else if A received or cycled.

So that we can understand why the server has -30 for A, we keep a transaction record for each user. Taking the case mentioned above, when B goes online and updates the server, the server besides updating the points of A and B, it also updates the transactions records for A. It adds an entry to the transaction record of A with {B_ID, 50}. Then C goes online and we again update the transaction record of A which now will have two entries {{B_ID,50},{C_ID,30}}. Thanks to this, we now know why A has -30 points.

5. Implementation

5.1 Server

For server implementation we chose the Spring Framework, since we wanted to build a RESTful Web Service to serve our clients (Android app), exposing endpoints that do want we want. We choose this approach because expose our resources (data source) in a controlled environment via HTTP protocol, the endpoints translate CRUD operations in our MongoDB collections (user, trajectory and stations), we specify every endpoint in README of the server, for example:

- POST http://localhost:8080/users - create a new user, and return HTTP status code 201 with the user created.
- POST /users/{userId}/validatePointsReceived - validate the received tuple and HMAC sended by the user who receive the points ({userId})
- POST /users/login - receive the user information from the client and validate the password if correspond with the one stored return status code 200, 406 if not and 404 if user cannot be found in database.
- GET /stations/{stationId}/bookingBike - book a bike in a specific station {stationId}
- GET /stations/{stationId}/bikeReturned - return a bike in a specific station {stationId}

Our server architecture follows the MVC model, the information is exposed in the endpoints in JSON format. The Model is specified via POJO objects, and controller via RestController files/classes for each Model (See Fig. 3 of appendix).

5.1.1 POJO objects - To represent a resource/Model in our architecture we follow a basic standard, Java POJO object, that specify all the properties and methods for each resource (in this project: User, Trajectory, Station are the main ones).

5.1.2 Rest Controller - The controller is responsible to handle the request for a specific resource/Model, and is here that we apply the **logic of our project** to deliver to the client the treated data that he wants.

5.2 Android components

5.2.1 NetworkDetector - The ***NetworkDetector*** is a broadcast receiver that is in charge of detecting changes in connection to the internet. This is done mainly to ensure that any pending information is sent to the server as soon as a connection is established.
5.2.2 MapsActivity - The ***MapsActivity*** contains a Google map and information about the trajectory of the user. It is started when the system detects that a user has started travelling.
5.2.3 Global Class - For sharing state between Activities we use **Global Class.** We share state for communication purposes like **SimWifiP2pManager** and **Channel.** We have a HashMap that contains all the friends nearby and their respective IP calculated by a function also belonging in this class. The Global Class also contains information about the user's location, namely the latitude and longitude. These are updated at first in the ***Login*** activity, to allow for calculations of the user's distance to every station.
5.2.4 Interfaces - When a user switches from tab to tab, or to pass information from an Activity to a fragment, for these two purposes, we use interfaces since intents cannot be used. In ***TabConversations*** we have an example of such interface called ***Callbacks*** which is used to exchange information between the ***TabConversations*** and ***TabFriendsList*** and the activity ***Friends.***
5.2.5 Client API - The client (Android app) needs to have capabilities to communicate with the server using the server API (endpoints), for this we use a 3rd party library, Retrofit.

5.3 Threading

We use Async Tasks for operations that can be performed in the background and generally long time operations and that don't need user interaction, therefore allowing a better user experience as the user doesn't have to wait until these operations conclude in order to keep using the application.

We use an Async Task called ***IncommingCommTask*** that is started when the user opens the application. This task is in charge of waiting for communications (messages or points) from other users forever (until the application is closed). When a message or points arrive, it treats each one accordingly, it differentiates both by a special character being, *":"* identifies a messages whereas *";"* identifies that points were received. When a message is received we display a Toast informing the user a message was received and we add it to the SQLite so its persistently stored. When points are received, we also display a Toast to the user to inform him that he received points but they are still waiting to be validated before the user can use them. We then check if the user is online or not, if he is we send the points to the server for validation, if the user is not online, we add the points to SQLite so that when the user comes online again he can send these points for validation in the server. Finally, when the points are properly validated, a Toast pops up informing the user that X points were validated and he can now use them.

SendCommTask is the Async Task that we use to send messages or points to other users. We have two SendCommTask defined, one for sending points and another sending messages. For sending points, we first increment the timestamp, then we create the tuple for sending the points consisting of <SenderID, ReceiverName, timestamp, points> and then we hash this tuple and we send both the tuple and hash. For sending the message is simply retrieving what the user typed and sending to the friend he wants to send to.

5.4 Socket communication

Two types of sockets are used for communication between users, ***SimWifiP2pSocketServer*** and ***SimWifiP2pSocket***. When the user opens the application, *SimWifiP2pSocketServer* is immediately opened so that the user can start receiving messages or points.

We use an Async Task called *Incomming Task*(like explained in 5.3) that has a loop that runs forever until the application is closed, in that loop SimWifiP2pSocketServer keeps waiting for connections to it, when it accepts a connection, a SimWifiP2pSocket gets the value accepted by the SocketServer and then we treat whatever was accepted by the SocketServer.

We initialize a SimWifiP2pSocket using the friend IP and port (defined in the SocketServer of our friend) when we want to communicate with a friend.

5.5 Server communication

The communication between server and client is done via HTTP protocol.

5.6 GPS Events

The detection of a user's location is made by a LocationListener object, which contains a OnLocationChanged method with the location of the user as a parameter. Using a LocationManager, we define the frequency of the location updates based on a minimum time or a minimum distance between the current and last location update. These objects can be found in the ***Login*** and ***Station*** activities, where a variable in the global class is updated with the user's last known location, and the ***MapsActivity,*** where the user's location is crucial to trace his trajectory on the map.

5.7 Persistent state maintained in the application

The persistent state maintained by the application was already explained in 3.1.2 where we use Sharedpreferences to store user related info and SQLite for larger information like, messages exchanged with other users, trajectories done by the user.

5.8 Optimizations

5.8.1 GPS Events - The updates to the user's location don't have to be made in real time, and this behaviour can be defined in the LocationManager object, more specifically in the parameters ***minTime*** and ***minDistance***. Setting a ***minTime*** above zero means that the location will be updated every ***minTime*** seconds. The same happens with the minDistance, with the user's location being updated every ***minDistance*** meters.
5.8.2 Getting nearby friends - We only update the friends that are nearby when it actually matters. For example, if the user goes to History, there's no point in updating the friends that are nearby since the user is not going to do anything related with friends. Therefore, we only update friends that are nearby when the user switches to **Friends** activity.

6. Limitations

User's Location updates - Currently, if there is no known last location of the user's device, the system will assume that both the longitude and latitude of said location are zero. Since the stations used in tests are in Lisbon, the distance from the last known location to the stations will be completely wrong. To circumvent this we provide the system with a location update in the login screen, allowing the system to update the user's last known location.

7. Conclusions

This project provided us with an opportunity to work with and better understand development in mobile and ubiquitous environments, more specifically Android applications. It also allowed for a better understanding of how to make use of the various tools that Google makes available, such as the Google Maps API. We successfully implemented all the requisites for this project and even applied some optimizations. We are sorry for exceeding the limit page by one, we tried to cut as much as we can but all the information seems relevant so we couldn't cut any more.

8. Appendix

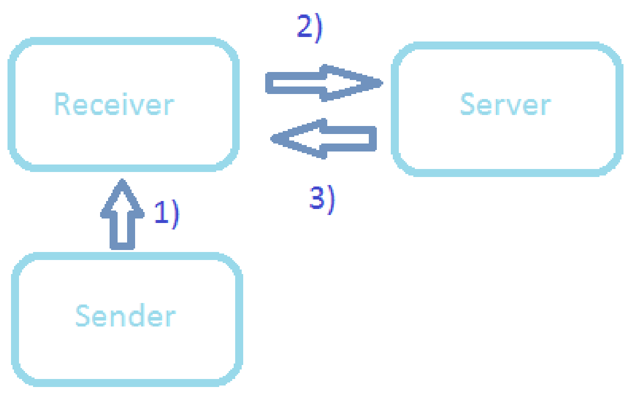

Fig. 1. Dataflow of point validation

1) Sender sends <SenderID, ReceiverName,Timestamp, Points, HMAC(SenderID, ReceiverName, Timestamp, Points)>

2) Receiver sends <SenderID, ReceiverName,Timestamp, Points, HMAC(SenderID, ReceiverName, Timestamp, Points)>

3) Server send ACK(Not validated) or ACK(validated)

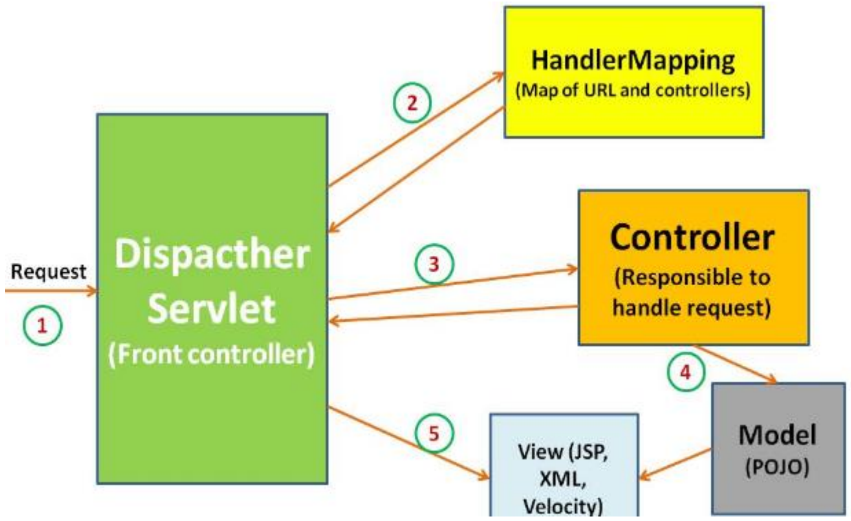
HandlerMapping
(Map of URL and controllers)
2
Request
1
Dispacther Servlet
(Front controller)
3
Controller
(Responsible to handle request)
4
Model
(POJO)
5
View (JSP, XML, Velocity)

Fig. 2 - MVC