Project Report

Ubiquitous and Mobile Computing - 2016/17

Course: MEIC

Campus: Alameda

Group: 2

Name: Gonçalo Alfredo dos Santos Rodrigues **Number**: 78958 **E-mail**: goncalo.s.rodrigues@tecnico.ulisboa.pt

Name: Nuno Cerqueira Afonso **Number**: 79035 **E-mail**: nuno.c.afonso@tecnico.ulisboa.pt

Name: Diogo Miguel Reis Silva **Number**: 79039 **E-mail**: diogo.m.r.silva@tecnico.ulisboa.pt

1. Achievements

All the requirements specified in the project description were fulfilled. The table below shows all the desired features as well as the implementation phase they were developed.

Version	Feature	State
Baseline	Sign up	Fully implemented
	Log in / out	Fully implemented
	List / create / remove locations	Fully implemented
	Post messages	Fully implemented
	Unpost messages	Fully implemented
	Read message	Fully implemented
	Edit user profile	Fully implemented
	Support for different policies	Fully implemented
	Centralized message delivery	Fully implemented
	Decentralized (direct) message delivery	Fully implemented
Advanced	Security	Fully implemented
	Relay routing	Fully implemented

2. Mobile Interface Design

The activity wireframe can be seen in the Appendix 1.

When planning the application's interface, we wanted it to be simple and with the different functionalities well separated. So, we mostly chose activities with a drawer, which present to the user all the different sections in only one place. We also included a floating button on the bottom right corner, allowing the creation of the various entities. This is a way of being consistent, no matter in which section a user is. Finally, we added the specification of the maximum number of mule messages in the profile, because it may be unique to each user.

3. Baseline Architecture

3.1 Data Structures Maintained by Server and Client

The server maintains a database with users, sessions, both Wi-Fi and Global Positioning System (GPS) locations, centralized messages and filters. When the Android application wants to retrieve some of these information, the server has different endpoints for the different operations.

The Android application makes use of shared preferences and of the available database. The first stores session related content, such as the session identifier, login timestamp, message counter and current user's username. The second is in charge of more complex data like the components of the different messages (both centralized as ad-hoc), location information, profile filters and measurements caching.

3.2 Description of Client-Server Protocols

For the baseline solution, we did not have security concerns. So, all the communications between the clients and the server were done by Hypertext Transfer Protocol (HTTP). The messages follow a JavaScript Object Notation (JSON) and server is allocated at locmess.duckdns.org.

Regarding the log in and sign up operations, the client sends an username and the correspondent password. For the first, the server checks if the username is assigned and if the password matches the one expected. For the second, the new account is only created when the username is unique. In a successful case, both will receive a randomly created session identifier and the operation's timestamp. Moreover, the login will also have the profile filters chosen by that user as well as that account's created messages.

For the log out operation, the client contacts the server, sending its current session identifier. Then, that token is taken away from that device and it cannot make other requests, unless it reauthenticates itself.

To have the latest information, the client can make many requests. It can get location information (both simple and detailed), available filters, number of centralized messages available to receive and, if the user touches on the notification, those counted messages. The latter two need the last aggregated positions by the client.

The individual user can contribute to the community by creating both GPS and Wi-Fi identified locations, filters and messages. For locations, the server must receive their specification in only one of the two ways. Filters are added directly to a user's profile or, if a new message contains some that do not exist, they are included in the server's database, but are not associated to any user. When posting messages, the server only accepts them when the given session identifier is associated with their author. The message identifier is sent from the Android application and is composed by (type of message byte || random value || login timestamp || message counter).

Finally, the clients can also remove locations, profile filters and posted messages (unpost message in the achievements table). Removed locations become invisible for all users. Profile filter elimination means that the server removes the connection between that filter and the user represented by the session identifier. However, that filter remains accessible to selection when posting a message. Message removal eliminates that message from the server's database. But, other clients that received it and other user's devices will still have them until they log out.

3.3 Description of Peer-to-Peer (P2P) Protocol for Decentralized Message Delivery

The chosen P2P protocol for this architecture is quite simple. Every time the number of nodes change in the ad-hoc network, the client sends its decentralized messages to everyone he had not yet send. The receiving nodes will just send an acknowledge (ACK) confirming the reception of the message. They are the ones responsible for discarding the message if they cannot receive it, due to filters or location.

After this initial transmission, the receiver can act as a mule. This is done in a similar fashion as the normal message delivery, because the number of bytes that would be saved in the communication are not that much and we can protect the privacy of the users (with respect to their location and filters).

All the messages are sent in a plain text JSON format, given its simplicity and ease of understanding.

4. Advanced Features

4.1 Security

To ensure secure communication between the server and the clients, we replaced the original HTTP protocol for HTTP Secure (HTTPS). The server is authenticated through its self-signed 4096-bit public key

certificate, which was previously distributed through the Android application's installation file. The client has to supply its own username and password, in order to receive an unique session identifier.

Because of the Diffie-Hellman protocol, the exchanged messages will be ciphered with symmetric encryption, making an eavesdrop attack unsuccessful. The Transport Layer Security (TLS) component will add a Message Authentication Code (MAC) that will be used to check message integrity. So, a tempered message will be detected. The Transmission Control Protocol (TCP) will prevent the suppression and addition of packets. The first is defeated by the resending of non-received packets. For the second, the attacker must know the expected sequence number, create a message with a correct MAC and cipher it with the right symmetric key. This is very unlikely to happen.

In order to avoid a key distribution problem, the clients trust that the server cannot be compromised and that it only signs existing ad-hoc messages. The correctness of those messages is assured when the given session identifier is the one from the message author.

The server signs ad-hoc messages when the clients contact it to know specific details of the receiving location. For that, it uses a SHA-256 hash over (message identifier || username || location || start date || end date || message text || filters) and cipher the result with its private key. Then, this digital signature is appended to the corresponding message and verified when it arrives to a new peer.

On the centralized model, the clients are sure that the messages are authenticated, because they come directly from the trusted server. On the ad-hoc model, the receiving peer authenticates the message creator when the message matches the one that was signed by the server.

4.2 Relay Routing

In order to speed up the delivery and reachability of decentralized messages, our algorithm for relay routing is based on the history of each user's location. We assume that users which travelled along some path will travel along the same one in the next few days with high probability. So, whenever a user wants to post a message at a location L, he will propagate this message to nearby nodes that usually pass by that location, hoping that they can reach the final users.

In the implementation, we keep histories of GPS paths (sequence of vectors linking GPS locations) and seen Wi-Fi IDs. As time goes on, the sample size will get larger, leading to reduced disk space, unreliable information (not fresh) and increasing computation time. So, history entries are kept in the database for a period of seven days in each user's device.

For Wi-Fi ID based locations, a user keeps a mule message if he has seen at least one of the Wi-Fi IDs of that location, otherwise he will discard it. For GPS, it is a bit trickier.

To check if a user has recently passed by a location, we start by computing the minimum distance of that location center to all paths. If we denote V as all vectors of all paths, then the distance of a location L to all paths can be written as $d(L) = \min_{\vec{v} \in V} d(L, \vec{v})$. Given a point X and a vector $\overrightarrow{AB}$, $d(X, \overrightarrow{AB})$ is equal to the distance of a perpendicular line connecting X to $\overrightarrow{AB}$, unless X is *before* A or *after* B. In those cases, the distance of X to the vector is the same as either the distance of X to A or X to B, choosing the smallest.

To compute this perpendicular distance, we can use Pythagoras theorem, as showed in Figure 1, and we get that $d(X, \overrightarrow{AB})^2 = \left\|\overrightarrow{AX}\right\|^2 - \left\|proj_{\overrightarrow{AB}}\overrightarrow{AX}\right\|^2 = \left\|\overrightarrow{AX}\right\|^2 - \left\|\frac{\overrightarrow{AX} \cdot \overrightarrow{AB}}{\left\|\overrightarrow{AB}\right\|^2}\overrightarrow{AB}\right\|^2 = \left\|\overrightarrow{AX}\right\|^2 - \left\|\frac{\overrightarrow{AX} \cdot \overrightarrow{AB}}{\left\|\overrightarrow{AB}\right\|}\right\|^2$. If $0 < \overrightarrow{AB} \cdot \overrightarrow{AX} < \left\|\overrightarrow{AB}\right\|^2$, then d is the distance to vector.

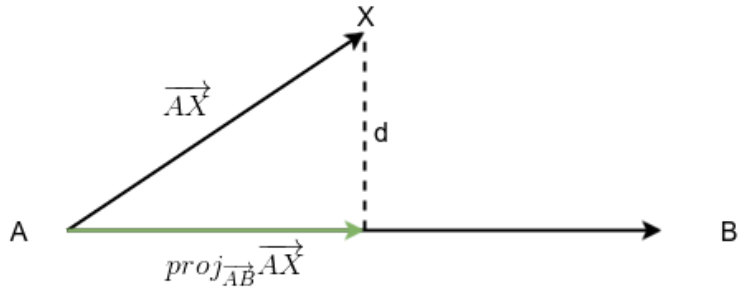

Figure 1- Graphical representation of the projection of $\overrightarrow{AX}$ over $\overrightarrow{AB}$ (in green).

In practice, we get these values incrementally, computing $|\overrightarrow{AB}|$ and $|\overrightarrow{AX}|$ only once for every vector that we store. We always use squared values to avoid doing square roots. This leads to an efficient algorithm that scales linearly with the amount of vectors.

However, the large amount of GPS points can lead to huge CPU and memory waste. To overcome this, we periodically aggregate paths. For each path, we start with the initial vector $\overrightarrow{AB}$ and check if the next points are within a small distance d (in our case ten meters) of a line with the direction of $\overrightarrow{AB}$. We ignore all these points but the last one that is still within this distance, connecting B to that it. After, we will use the vector starting from the last point that was still within distance d and repeat this process until we reach the final vector of the path.

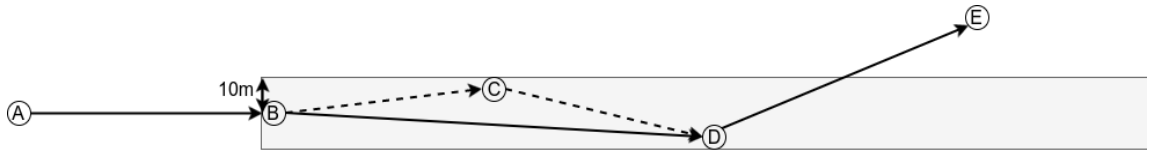

Figure 2 – Example of the point aggregation algorithm. Dashed arrows represent the user's path and filled arrows show the stored vectors.

In Figure 2 there's an example of this algorithm, where A->B->C->D->E form a path. After the aggregation, the path is reduced to A->B->D->E, because C was within a small distance of the line formed by vector $\overrightarrow{AB}$.

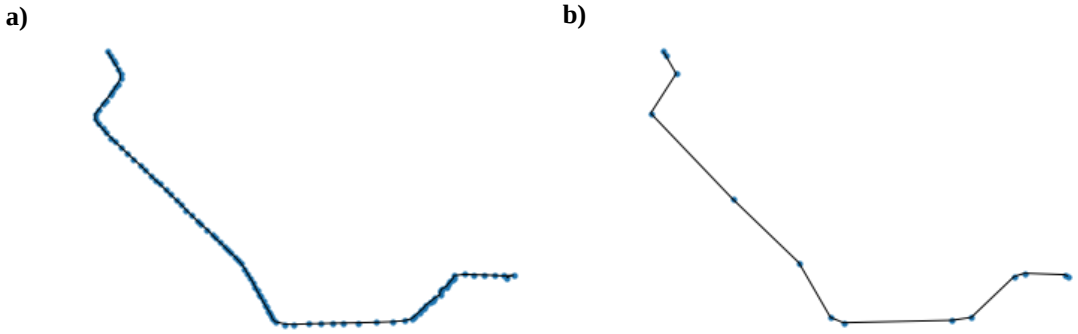

Figure 3 – Real example of a user path. On **a)**, all the recovered points are shown. On **b)**, only the points filtered by the aggregation algorithm are shown.

In Figure 3, we show a real world example of the previous algorithm. Each blue point indicates a GPS location. On **a)**, we have the original path with all the GPS system readings in an area of about 10 km^2. On **b)**, we have the same path after executing the aggregation algorithm, going from 100 points to just 14.

Finally, the user can specify the maximum number of messages to mule in their profile. Every time the user receives a new message and the buffer is full, the application replaces the oldest message by the newer one.

5. Implementation

The server was developed in Python 2.7, making use of the Flask framework. The main reason behind this decision was that it abstracts all the network issues and the code only needs small annotations to create the endpoints. When developing the Advanced Features, we also learnt that the HTTPS configuration was straightforward, needing to only give the certificate and private key as context.

To allow the communication between the server and its database, we used the *python-mysqldb* module. We had to also use the *pycrypto* module, so that the server could sign the ad-hoc messages. On the Android side, we used the *Termite* library to allow P2P communication and the *SearchableSpinner* to select the location for posting a message.

The Android application is mostly composed of activities with a navigation drawer, to allow easy access to all the functionalities. For the location measurements, there is a Periodic Location Service that stores the most recent GPS and Wi-Fi Direct hotspots readings and provides them to other components. Namely, the NewLocationActivity needs it when the user wants current readings for a new location and the WifiDirectService (itself a service) checks it to infer if a new decentralized message should be kept or discarded. Another important component is an alarm that sets off once a day to aggregate the paths discussed in the section above.

The only permanently active threads are a thread to send the location to the server periodically and a thread waiting for Wi-Fi Direct requests from nodes that want to share decentralized messages. The others are created periodically or on demand to, for example, clean the database, aggregate paths, send decentralized messages or to communicate with the server (using Async Tasks).

The mobile device keeps persistent state, either to avoid communicating with the server or to be able to work in the decentralized mode. It maintains SQLite tables, namely for received messages, created messages, muled messages, profile filters, recently walked paths and seen SSIDs. Login information is kept in Shared Preferences. Temporary network information is stored in the NetworkGlobalState class. They are cleared out on logout.

For the relay routing algorithms, we map the (Latitude, Longitude) values to a (x, y) coordinate system, working on the second ones. This allows an Equirectangular projection in which Portugal is the center of the coordinate system. Although being frequently negligible, the error will be bigger for other longitude values, but it can easily be fixed by using different coordinate systems for different geographical regions.

The application aggregates measured locations before sending them to the server. The increased period allows resource sparing without losing accuracy, even though it increases latency.

6. Limitations

Locations are only sent to the server if they differ from the last communicate ones. Due to this optimization, if a new message is sent to a location in which a user is already in, he will only receive it when he moves.

7. Conclusions

This application allows users to send messages to any location using either a centralized or a decentralized approach. For the second, the receivers don't need to communicate with the server or even have a working Internet connection. Our approach tries to maximize the probability of messages reaching the receivers by using the history of each user's location with an efficient use of the available resources.

Furthermore, we protect each user's privacy by encrypting all communication with the server and by not revealing the user's location to other nodes in the decentralized mode. We also authenticate the posted messages, using the help of the central server.

Appendix 1 – Activity Wireframe