Project Report

## Ubiquitous and Mobile Computing - 2016/17

Course: MEIC

Campus: Alameda

Group: 12

Name: Luís Filipe Abreu dos Santos Number: 77900 E-mail: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Name: Catarina Alexandra Pereira Cepeda Number: 77966 E-mail: catarinacepeda@tecnico.ulisboa.pt

Name: Paulo Jorge Almeida dos Anjos Number: 87822 E-mail: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*(PAGE LIMIT: 5 pages – excluding the cover)*

## 1. Achievements

*The following table describes which features stated in the project specification were implemented.*

|  |  |  |
| --- | --- | --- |
| **Version** | **Feature** | **Fully / Partially / Not implemented** |
| 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 | Not implemented |
| Relay routing | Not implemented |

## 2. Mobile Interface Design

*Draw the activity wireframe of your program. Explain it succinctly. Describe any specific features of the program behavior that have not been explicitly stated in the project specification. The wireframe diagram can be added to the report in a separate appendix not counting for the 5-page limit.*

## 3. Baseline Architecture

**3.1 Data Structures Maintained by Server and Client**

The server stores and manages user accounts, locations, keys, *key-pairs* associated with each user, and messages posted in locations by the users. Each client maintains data related to each user that has logged in on the device – namely, the users’ *key-pairs* and their posted, received and read messages –, as well as a complete list of locations and keys pertaining to *key-pairs* of all registered users. In both the server and the clients, this data is persistent and stored in databases. Additionally, the client makes use of a key-value store provided by the framework, to keep simple information such as the device’s current geographical coordinates and the list of SSIDs of detected WiFi signals – henceforth referred to as *coordinates* – which can be shared by several components.

**3.2 Description of Client-Server Protocols**

Initially the client must sign up or log in with the credentials provided by the user; the server will generate a session token in response, and send it back to the client, which stores it. The client may now issue requests for listing, creating or deleting *key-pairs*, locations and messages, which must be accompanied by the session token. These requests contain the data required to fulfill them, such as the information entered by the user, in the case of creation; or IDs defined by the server after creation, in the case of deletion.

With regards to the centralized message delivery, the protocol is as follows:

Periodically, the client will send a request to the server, containing the device’s current coordinates. The server will then retrieve all the *active* messages (whose visibility time window has not expired) posted in locations that match the coordinates sent by the client. For each of these messages, the server will compare its posting policy (whitelist and blacklist), as specified by the message’s author, with the *key-pairs* defined in the user profile associated with the request, and the messages that are selected are then sent back to the client.

The criteria used for filtering the messages are: if the whitelist is not empty, the client will not receive the message if the user profile does not contain any of the *key-pairs* in the whitelist. In the case where the user profile contains at least one *key-pair* in the message’s whitelist, or the whitelist is empty, the client will receive the message if the user profile does not contain any *key-pair* that is present in the message’s blacklist.

Upon receiving the messages, the client will merge them with the already stored messages. Messages that are stored locally but are not in the list of received messages are deleted from the database; messages that are in both the list of received messages and the database are kept; the remaining messages are new messages, and thus they are stored in the local database. (This approach is also used when merging a list of keys, *key-pairs*, or locations retrieved from the server.)

**3.3 Description of P2P Protocol for Decentralized Message Delivery**

The protocol designed for decentralized message delivery is described below:

Periodically, the client will scan the database for all the user’s posted, active P2P messages and, for each one, compare its location with the device’s current coordinates, retrieving those that match the device’s current location. For each message, the client will then create a *match request* associating each message’s unique P2P ID, generated in that moment, with the respective whitelist and blacklist. The unique P2P ID is a hashed value generated based on the user’s username and the local ID of the message. At this point, the *match request* is sent to every peer detected by the device.

When a peer, who is also a client, receives a *match request*, it will match each message’s posting policy against the *key-pairs* defined in the user profile, based on the same criteria described in the previous subsection. Additionally, to avoid storing the same message multiple times, the peer will also scan the database for received messages with the same P2P ID of a message present in the *match request*, and in case of match, will skip the filtering of the posting policy for that message. After this process, the peer will create a *match response* associating each message’s unique P2P ID with a Boolean value, representing whether the peer expects to receive that message.

Upon receiving the *match response*, the client will create a *message request* containing the information of every message which the peer declared to expect to receive. When the peer receives a *message request*, it will store all the messages in its local database. Finally the peer will send an *acknowledgment* to the client, terminating the communication. (P2P messages that do not match the device’s current coordinates are routinely deleted from the database.)

**3.4 Other Relevant Design Features**

The application and database have been designed with the intent of fully supporting multiple user accounts in a single device. We did not implement the components of the User Interface that would allow for a *fast-switch* between user accounts – presently, if the user wants to switch to another account, it will have to log out from the current account, granted that all data will still be available when they log back in – however, we thought that it was important to note.

## 4. Advanced Features

**4.1 Security**

*Describe the protocols that were designed to achieve the security goals of your project.*

We did not design nor implement any security protocols for our solution.

**4.2 Relay Routing**

*TODO: Describe the protocols for relay routing.*

## 5. Implementation

The server was implemented in Python, using Version 3 of the Django REST Framework. In order to determine the distance between two geographical coordinates, the server uses a geocoding library named GeoPy.

The client was implemented in Android, supporting API ≥ 15. It uses the Google Play Services location APIs for requesting and retrieving the device’s current location, as well as determining the distance between two geographical coordinates; and the Termite API for emulating virtual WiFi Direct networks.

The client application consists of a set of activities, services, and broadcast receivers (the most important being those that respond to regular alarm triggers), as well as a content provider and the required helper classes. User authentication is handled by means of an *AuthenticatorActivity* and a *SignUpActivity*, and then controlled by an authenticator service, as recommended in the Android guidelines. The authenticator service is used by the system’s *AccountManager* to simplify the account creation and management. The session token obtained after a successful login is stored in the *AccountManager*, which associates it with the account.

Other activities implemented provide the user with an interface for issuing requests to list, create or delete data – the activities implemented refer to the user’s inbox (with tabs for available and opened messages), posted messages (with tabs for active and archived messages), locations list and user profile, as well as activities to create locations, open messages and create messages.

The application has two alarm receivers that respond to periodic triggers.

One is responsible for initiating an intent service that, given the current coordinates, fetches the messages posted in that location (*FetchLocationMessagesService*); it also controls the lifetime of the service that receives updates for changes in geographical coordinates or detected WiFi signals (*LocationUpdateService*) – this service binds to Google’s location provider and to Termite’s service.

The other alarm is responsible for controlling the lifetime of services associated with the decentralized message delivery. Three services were created in order to implement the P2P protocol described in subsection 3.3. *P2pMessageScannerService* is in charge of retrieving the posted messages for the current location from the local storage; it also receives updates to changes in peer group membership (via Termite). This service will spawn new threads to handle the communication with each peer using the service described next, and following the procedures described in the protocol. *P2pMessageSenderService* is an intent service which receives the IP address and port number of the peer, and the data to send – *match* and *message* requests. It opens a client socket to the peer and after receiving a response, sends it back to the *P2pMessageScannerService*, as defined in the protocol. *P2pMessageReceiverService* has a server socket, and it will accept connection requests. Upon receiving a request, the service proceeds according to the protocol. To avoid conflicts in the database, this service can only answer to one P2P request at once, and thus it does not spawn any extra threads.

All the services described above run on a separate thread. Apart from the services, everything else runs on the UI thread, except: (1) when the class’s default definition already assures a separate thread will be created to run instances, as is the case with the *AbstractThreadedSyncAdapter* and *CursorLoader*, (2) the user authentication runs on a separate thread since it communicates with the server and that could potentially lead to Application Not Responding errors, and (3) when creating a message, a separate thread is spawned to handle the necessary verifications before posting the message – creating a separate thread was not strictly necessary, but we thought it would be safer to do it in order to avoid Application Not Responding errors.

Persistent state is kept in a database, and global state is kept in the key-value store available in Android, the *SharedPreferences*. Communication between components running of the same device is handled through broadcast receivers, and in some cases *ResultReceivers*, which can be seen as a simplification of broadcast receivers.

Regarding network communications and sockets: communication with the server is done through the class *HttpURLConnection*, and the session token, a JWT token, is added to the request’s Authentication header; with the exception of the requests issued by *FetchLocationMessagesService*, all server requests issued by the client are handled by a *SyncAdapter*, which provides automatic queueing and scheduling of requests; communication between peer devices, in the decentralized delivery mode, is done through sockets using the Termite API. In both cases the data is sent and received in plaintext, without any security protocols in place.

As mentioned before, the mobile device maintains data related to the each user that logs in – the *key-pairs* associated with the user’s profile, as well as their posted, received and read messages. It also keeps information about the locations and keys created by any registered user; these are updated occasionally, currently at an interval set to 30 minutes. This data is stored in a SQLite database, and manipulated using helper classes such as the content provider class *LocMessProvider*. While the data directly connected to a user is kept alongside an identifier (a hashed value of the user’s username) to allow for multiple accounts in the same device, data such as the list of locations or keys need not have that identifier and are shared by any user account.

We implemented some optimizations that we thought would improve battery life. Every time the alarm for fetching location messages is triggered, it reads both the current coordinates and the previous coordinates from the key-value store (the previous coordinates are stored by the alarm upon finish its work, with the value of the current coordinates) and compares them: if the WiFi SSIDs have not changed and the distance between the geographical coordinates is smaller than a defined tolerance distance (currently 10 meters), then the application presumes that the device’s location has not changed. When this happens, the application reschedules the alarm, doubling its current *repeat interval*, yet this interval is always kept within a defined bound (currently no more than 5 minutes). If the application determines that the device’s location has changed, then the alarm is rescheduled, resetting the *repeat interval* to the default value (currently 1 minute). With this approach, we take advantage of locality to prevent the battery from draining too fast – that is, if the user has not changed location since the last time the alarm was triggered, then it is likely that they will not change location in the near future. Furthermore, we also implemented another measure to prevent the fast draining of battery: when the alarms are triggered, the services which depend on network connections or which would generate results to be sent through the network will be stopped if the device does not have an Internet connection. These services will be restarted the next time the alarms are triggered, if at that time the device has an Internet connection.

## 6. Limitations

Given that it was not possible to test the application with mobile devices, the reference values defined are rough estimations of possible values that could be used in a real-life scenario, and should be fine-tuned once a proper evaluation of the application was conducted. For instance, the tolerance distance of 10 meters, for determining whether the current location has changed, would need to be dynamically adapted according to the accuracy of the geographical coordinates retrieved from Google’s location provider; this is not possible in the current test environment, since the coordinates retrieved are those entered into the Android emulator. Similarly, the default and maximum intervals for the alarms defined should also be subject to proper evaluation, so as to provide the best compromise between the application requirements and battery life.

## 7. Conclusions

Although we implemented only the baseline features presented in the project specification, we are quite pleased with our application, as we feel it succeeds in achieving the goal of providing location-aware messaging functionalities. In the future, we would like to improve our application by implementing the advanced features concerning security and relay routing. We would also consider a redesign of the user interface, while paying special attention to Google’s guidelines for accessibility in Android applications.

Finally, regarding the practical component of the course: we think that the class’s plan was adequate, and that the provided material proved to be of great help in understanding the Android basics. However, we feel that the Genymotion connector mentioned in the last lab guide as an alternative to AVD could have been introduced in the earlier lab classes, as one of our group members could not run AVD on their laptop due to its high resource consumption and that limited their ability to test the application.

## 8. Appendix