Project Report

Mobile and Ubiquitous Computing - 2019/20

Course: MEIC

Campus: Tagus

Group: 4

Name: João Rafael Soares Number: 87675 E-mail: joao.rafael.pinto.soares@tecnico.ulisboa.pt

Name: José Brás Number: 82069 E-mail: jose.bras@tecnico.ulisboa.pt

Name: Miguel Barros Number: 87691 E-mail: miguel.v.barros@tecnico.ulisboa.pt

1. Features

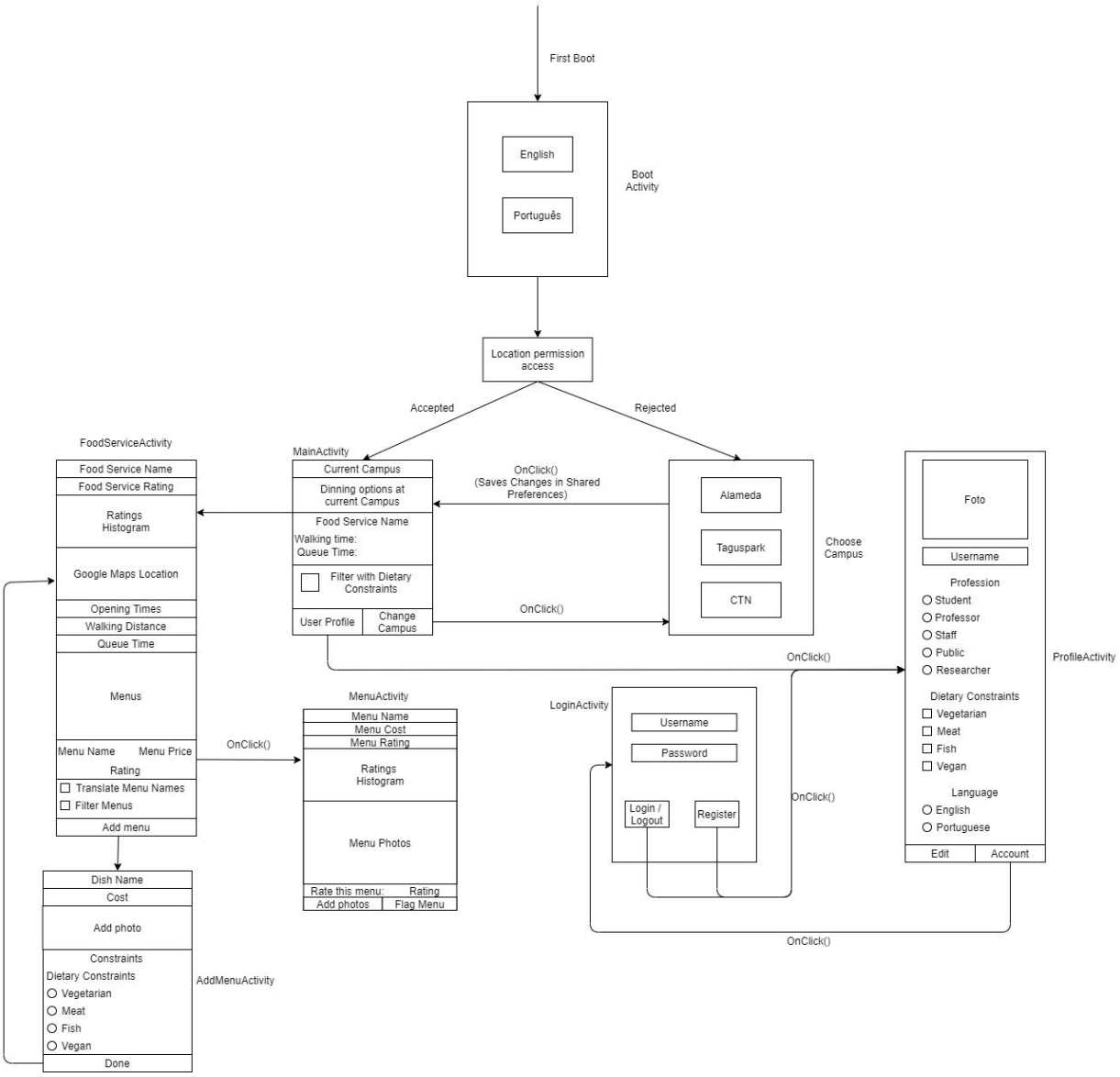
Component	Feature	Fully / Partially / Not implemented?
Mandatory Features	List Food Services by Campus	Fully
	Show Food Service Information and Menu	Fully
	Show Dish Information and Photos	Fully
	User Profiles	Fully
	Dish Information Submission	Fully
	Dish Photo Submission	Fully
	Automatic Campus Selection	Fully
	Food Service Queue Estimation	Fully
	Photo Caching	Fully
	Cache Preloading	Fully
Securing Communications	Encrypt Data in Transit	Fully
	Check Trust in Server	Fully
Meta Moderation	Data Flagging	Fully
	Data Sorting and Filtering	Fully
	User Trustworthiness	Fully
User Ratings	Dish Rating Submission	Fully
	Average Ratings for Dishes / Food Services	Fully
	Rating Breakdown (Histogram)	Fully
User Accounts	Account Creation	Fully
	Login / Logout	Fully
	Account Data Synchronization	Fully
	Guest Access	Fully
Social Sharing	Sharing Food Services	Not Implemented
	Sharing Dishes	Not Implemented
Friend Tracking	Location Sharing Option and Selfie	Partially (Implemented Selfie)
	Show Photos of People at Food Services	Not Implemented
	Sort Photos by Friend Likelihood	Not Implemented
User Prompts	Prompt Upon Queue Entry	Not Implemented
	Prompt Upon Queue Exit	Not Implemented
Localization	Translate Static Content	Fully
	Translate User Submitted Content	Fully
Dietary Constraints	Dish Categories and User Diet Selection	Fully
	Dish Filtering	Fully
	Food Service Filtering	Fully

Optimizations

To improve usability when the network is down, we cache the application state that is maintained by the server, i.e. the food menus at each food service. When the network is down the user still has access to all the information he saw when the network was up.
To improve perceived performance, all communication that is intensive (i.e. parsing food service data from a JSON file) or that involves network communication is not run on the UI thread, to block it from being stuck. To avoid the overhead of creating and destroying threads, a cached thread pool is used, which reuses previous threads.

Since GRPC uses protobuf, the data shared between the server and the device is compressed therefore decreasing the overhead between communications. Since HTTP2 is used as a basis for GRPC this allows for efficient client-server communications.
The application maintains a partial functionality even if it does not have access to the user location. However, whilst working offline, the application is unable to infer the campus of the user and the walking time to each food service.
To improve power consumption, we avoid requesting new location requests to infer the current campus and calculate the walking time to the food services if the previous request is sufficiently fresh.

2. Mobile Interface Design



3. Server Architecture

3.1 Data Structures Maintained by Server and Client

The server maintains all the menus for each service and photos uploaded for each menu. It also stores user account data (username and password hash) and keeps queue information for each service, i.e. how many people are currently at each food service, how many people are ahead of each one and also records how much time it took for each user to exit the food service, which it uses to deduce the current queue time of each service by linear regression.

We simulated a key value store in memory using Maps. Each menu and photo is indexed by a sequential Id defined by the server.

The client caches menus and photos seen, so that if the network is unavailable the application is still usable and useful.

3.2 Description of Client-Server Protocols

For the client server communication we used GRPC, for its simplicity, ease of use and high performance.

We use one way TLS authentication for the server and Username and Password authentication for the client. The server stores the passwords hashed with a 64 bit salt using PBKDF2. After authentication, the user receives a session cookie which he uses to stay authenticated even after closing the application and reopening. Uploading a menu and a Photo requires the user to be logged in, thus the user must send a cookie when requesting it.

For requests related to the service queues (joining and leaving a queue or canceling a previous join), a UUID is used to identify each client. If we were to develop the application further, privacy considerations would have been taken into account since the server currently can keep track of the user's activities by this UUID.

4. Implementation

The server was implemented using Java 11, Gradle 6.3.

For external libraries, we used:

- *GoogleDirectionLibrary*
- *Google translate*
- *Histogram*

The app was implemented in Java 8.We have one service running on our application:

- SimWifiP2pService, used for Termite simulation;

All communication with the server is done trought AsyncTasks in order to be as efficient and parallel as possible, including for:

- Download and upload of Menus;
- Download and upload of Photos;
- Upload of Ratings;
- Registering and Login features;
- Joining and Leaving Queues.

Since a Cached Thread Pool is used to run the Async Tasks, threads can be reused to avoid the overhead of creating threads.

Data shared between multiple activities is either stored in the Application Context (ex: GRPC stubs) or passed between them using intents. The user profile is stored in the Shared Preferences and is the only permanent state the application maintains.

State is maintained on the mobile device by saving data in local files, like profile information and downloaded photos.

5. Running the Prototype

Running the Server:

To run the server, you must first:

1. Go to *FoodIST-Server* folder;
2. Enter the folder *server* and run the command *source googleTranslate.sh* (this sets an environment variable necessary to communicate with the google translate API);
3. Start the server by running the command *gradle startServer*.

Running Termite:

To run Termite, you first:

1. Go to *Termite-Cli* folder;
2. Run the command source *env_setup.sh* (this sets the necessary environment variables that Termite needs).
3. Run the command *./termite.sh.*

We have included some Setup scripts prepared for Termite in the *scripts* folder of *Termite-Cli*, specifically:

- beaconSetup - Sets up all the beacons names;
- one/two/threeClientSetup - Sets up either 1, 2 or 3 clients for testing and connection with the emulators.
- beaconNameTest - Tests the functioning of the queues for each beacon;

The application can be run in an AVD (device model Nexus 5X 2019 with API level 29). This can be done via Android Studio or the command line.

6. Simplifications

In a real world scenario, we would apply the following changes:

- Use a more robust authentication protocol (something like https://en.wikipedia.org/wiki/S/KEY), since our current protocol is very simplified and not the most secure;
- We would implement persistent data on the server side using a database system (for example, DynamoDB) and we would replicate the server for bigger availability;
- We would have certificates for each server IP/DNS, since currently our certificates are for localhost usage only;
- The server data cached on the device is not permanent. We would use something like SQLite available on the android platform to make it so;
- We would create a more elaborate front-end interface and test it on multiple different devices;
- We would use a cloud provider like AWS to run the servers.

7. Bugs

Whenever the user logs in he is booted back to the main activity as if he had just booted the application. This was done because when a user logs in his language can change, however previous screens would still be drawn in the previous language.

In order to fix this we would have to redraw each screen when it comes into the foreground. When this problem was detected the prototype was already in an advanced state thus requiring significant changes in the application's structure. Since some of the text displayed in the interface is stored in intents from previous activities, it was decided that, despite its flaws, it was a reasonable compromise between further implementation of new functionalities and development time. However, this is without a doubt an usability problem that, given more development time, would have fixed. Nevertheless, since our solution can potentially support an arbitrary number of languages, a more general solution would have to be developed where the client translates the screen using the Google Translate API, which at the present time only the server makes use of.

Finally, another bug present in our application is the main activity not being drawn after clicking run on the Android Studio IDE. When this happens, one shall close the application in the AVD and reopen it. Despite this being an Android Studio bug, and not a bug in our mobile application, we decided on including it on our report merely for bug awareness purposes.

8. Conclusions

With this project we were able to work with many different components of a normal Android application, in a practical and down-to-earth project. We implemented techniques related to context awareness and adaptability, energy saving and mobile UI design. It would be interesting to further develop this application and add its functionalities to the tecnico app, which is sorely lacking food services information at the present time.

Venting:

- We found that some of the elements we used from the android API, namely AsyncTasks are deprecated. We aren't sure why they would deprecate such an intuitive feature, but it is now recommended to use Java's Concurrency API.
- It is also now impossible to declare application wide Broadcast Receivers on the android manifest, they must be started from the application context, adding needless boilerplate code to the application.

The timing of the theme of the app was unfortunate given global events but we still found it to be very interesting. We thank professor Luís Pedrosa for his help in answering all the questions on slack and during zoom calls.

Some more variety in extra components would be appreciated. Since many of them rely on one another, students are basically funneled to a subset of choices(for example, ratings and flagging menus require user accounts). Nonetheless the extra functionalities made us recall concepts from previous courses allowing us to better understand how these would be applied in modern-day mobile applications.