P2Photo - Project Report

Ubiquitous and Mobile Computing - 2018/19
MEIC-A
Group 11 - Alameda

Leonardo Epifânio	83496	leonardo.epifanio@tecnico.ulisboa.pt
Pedro Lopes	83540	pedro.daniel.l@tecnico.ulisboa.pt
Pedro Salgueiro	83542	pedro.f.salgueiro@tecnico.ulisboa.pt

1 Achievements

The target platform for P2Photo is Android version >= 4.0.3 (API level 15+). All the features were fully implemented with exception of groups of more than 2 people in Wifi-Direct as it was impossible to us develop and test due to the impossibility of running more than 2 Android Studio emulators in our personal computers and in the RNL computers. However, it's fully implemented for working with groups of 2.

Version	**Feature**	**Fully / Partially / Not implemented?**
Cloud Mode	Sign up	Fully
	Log in / out	Fully
	Create albums	Fully
	Find users	Fully
	Add photos to albums	Fully
	Add users to albums	Fully
	List user's albums	Fully
	View album	Fully
Wireless Mode	Sign up	Fully
	Log in / out	Fully
	Create albums	Fully
	Find users	Fully
	Add photos to albums	Fully
	Add users to albums	Fully
	List user's albums	Fully
	View album	Fully
Advanced	Security	Fully
	Availability	Fully

1.1 Optimizations

The login and sign up are made using **Google Sign-In**, which is a secure authentication system that reduces the burden of login for the users, by enabling them to sign in with their Google account, the same account they already use with Google Drive. This increases usability because the user no longer needs to remember or type credentials to login/sign up into the application. If the user is not yet signed up, the application automatically registers the user, without the user having to perform an extra step. It's also in this phase that needed authorizations are asked to the user regarding the Google Drive access.

Apart from that, there are a set of small optimizations made to improve the usability of the app, mostly regarding the availability protocol that will be explained in section 5, where, firstly, the user is presented with the information that's in cache. More precisely, the users' albums, images' thumbnails and users present in the system (with a limit of 100 users).

The application only requests the fully sized image when the user clicks in the image. This reduces the latency to show the images present in the album because it

only shows the thumbnails stored in cache. In case it doesn't exist in cache, it will request the thumbnails.

2 Mobile Interface Design

The mobile interface implements the activities and functionalities stated in the project specification. The wireframe can be found in the appendix (Figure 1).

3 Cloud-backed Architecture

The Cloud Storage version uses Google Drive as the cloud service provider. For that, we used the Google's REST API both in server and the Android app.

3.1 Data Structures Maintained by Server and Client

The server maintains all the information about the albums and users, for that, it stores persistently a list with the albums and the users. Then, in the users structure, it has a map with the IDs from the albums the user is part of. All the information is stored in JSON files.

In the client side, the user saves these lists in the shared properties, using the JSON format, with an additional string listing all the cached images (thumbnails).

3.2 Description of Client-Server Protocols

The communications are made using HTTPS. The server has a set of fixed endpoints which are available for the client to make requests. Two of them are are responsible for authentication (login and sign up), and the rest are for requesting data such as listing albums, users and for changing the state, for example, adding albums and adding users to albums. All requests, with the exception of the login and sign up, must have present, in the HTTP header, an authentication bearer token that is obtained during the login phase. The token is a JWT JSON web token and is valid for 24h.

When the user enters the activity containing the list of albums, it first shows what it has in cache and, asynchronously, requests the list of albums. The same happens for the images in an album.

When a client signs up, the server receives the Google's credentials and uses them to access the Google Drive and create a folder called "P2Photo". This folder will be used to accommodate all the albums and user's photos.

When a client creates an album, the server creates a folder (with name equal to the album's ID), inside the P2Photo folder which was created at sign up time, it creates a JSON file in it (representing the catalog), and saves this file's ID in its structures.

When a client adds a user to the album, the server shares the album's folder (in Google Drive) and shares it with the new user. If the album has more participants, this will happen to each of them. After that, the server creates a new folder (whose

name is the album's ID) in the newly added user's cloud storage, and shares it with the other participants present in the album.

When a client adds a new photo, the client itself uploads the photo to the corresponding album's folder and updates its metadata.

3.3 Other Relevant Design Features

As explained in the previous sub-section, the requests are made using an authentication token that has a limited span of time. When this span of time ends, the token is no longer valid and the server responds with an *unauthorized* exception. The application reacts to this exception by automatically performing the login, and this sequence is hidden from the user.

4 Wireless P2P Architecture

The wireless P2P mode is selected in the initial screen, by clicking the bottom-most button. This activates a flag which is used to decide the tasks to perform for each of the protocols (cloud and P2P). Much of the functionality is similar to the cloud mode. The most relevant differences are described bellow.

4.1 Data Structures Maintained by the Mobile Nodes

Since the catalogs are saved in each device, they must be transmitted through sockets, using some kind of structure to indentify the album and user they belong to. For this end, we created a structure named **Catalog** with the attributes *targetVirtIp* (reprensenting the IP of the P2P node that sent the message), *userName*, *albumName* and *catalogLineList* (the lines from the catalog file). Each node's process maintains a list of the **Catalog** objects it has received from the peers. The current network peers are also saved in a list, which is updated as the broadcast receiver gets events regarding network modifications.

4.2 Description of Messaging Protocols between Mobile Nodes

When an album is created, its metadata is saved in the cloud and the catalog is created in the owner's device. When the application queries the list of albums for the current user, the user creates the catalogs that are missing for his albums. Each time the network is modified (peers enter or leave the network group), the catalogs from each node are broadcast to the group peers. This broadcast also occurs when the user adds a photo to an album. When a peer receives a catalog, it saves on the list of **Catalogs**, and when the user enters an album photo's menu, it goes through the list, gets the **Catalogs** that correspond to the album and asks the respective peers for the photos, which then send them back. In this mode, the server will not access the user's cloud drive, using different endpoints for the client's requests.

5 Advanced Features

5.1 Security

All the communication between the client and server are secured using HTTP over TLS (as known as HTTPS). This mitigates man in the middle attacks, replay attacks, tampering or spoofing messages.

The Google Drive give us for free the security regarding the privacy of the albums' photos. The scope of user's permissions for the application are different regarding the server and the application iself. The server as more restricted permissions handling the Google Drive, being only able to create files and manage files' metadata such as users allowed to read/modify, not able to download any files from the Drive.

However, for future work, the application could be extended for different cloud storage providers like One Drive or Dropbox. And those new providers might not have this functionality that Google gives to us. To mitigate this problem, each user, when it signs up creates a RSA key pair where the private key is saved persistently in the mobile device and the public key is stored in a public repository. The public repository is only used for testing purposes, in a real world application, the better solution would be to use a Certificate Authority to save and sign the public keys. Having the keys, the user, ciphers the catalog's metadata file using the private key. The other users deciphers using the public key that can be retrieved from the public repository.

5.2 Availability

The availability protocol consists in caching all the main information in the mobile device, hoping that it will use it in the close future. It uses a simple cache protocol of the most recently used, where it will delete the files that the user doesn't use for a long time.

Cache is a precious resource, and we try to maximize the use of it. For that, we don't save the fully sized images as they are heavy, we save the thumbnails generated by Google which are small, compressed images. As an example an image that has a size of 153Kb, ends up with a 20Kb thumbnail. u Cache guarantees that when there is no internet connection, the application still functions properly with no warning dialogs saying there is no connection, improving the user experience.

6 Implementation

The server is implemented with Typescript and runs in Node.js. Server side uses many external libraries such as Expressjs, Passport, UUID and the Google Authentication Libraries / Google Apis used to communicate with the Google Drive. Client side, uses Gson to parse JSON, some Apache libraries such as CommonIO, Google Apis to communicate with Google Drive and OkHttp to communicate with the server.

There are a total of 6 activities. These activities communicate through intents and the interface lists (*RecyclerViews*) are implemented using *adapters*.

All the requests are using asynchronous tasks, every time a user enters or refreshes one of the following activities: *AlbumMenuActivity*, *ViewAlbumActivity* and *ViewPhotoActivity*, a task is started that tries to get the information required for that activity. In case of failure, it fails silently. In the P2P mode, another task is also running, from the point the user logs in, providing a *ServerSocket* to receive incoming requests. In this mode all communication between devices is made through Java sockets.

The application caches state, mostly the albums, users and images.

7 Limitations

While using Wifi-Direct, a user should be added to a network group while the user's app is running, since the catalogs are broadcast when the network group changes its state. These groups also need to be set manually by the users.

8 Conclusion

This project familiarized us with android development and two approaches to communication and storage in ubiquitous computing. We managed to implement both solutions and all of the features, but this proved to be a challenge. The problem was that we had to implement two versions/architectures of the program, being the P2P architecture the most problematic, chiefly due to the use of the Wifi-Direct API and its emulator, Termite. Several of the hours dedicated to this project were spent with problems regarding the setup of these tools and usage of the P2P testing. The Android Studio emulator is very resource heavy, to a point where we weren't even able to test the application with 3 emulators. Even 2 emulators proved impossible to run on two of the group members' personal computers, which impairs the testing of the app on those computers, plus, installing more than one AVD was not possible in the RNL lab computers.

In our opinion, it would've been a better project if either we used a feature, in the second architecture, that had more support from the android community, or, even better, if we only had to work on the cloud architecture, but with more functionalities to implement, as, for example, the removal of photos, which raises the problem of consistency, characteristic of the ubiquitous computing world.

Appendix

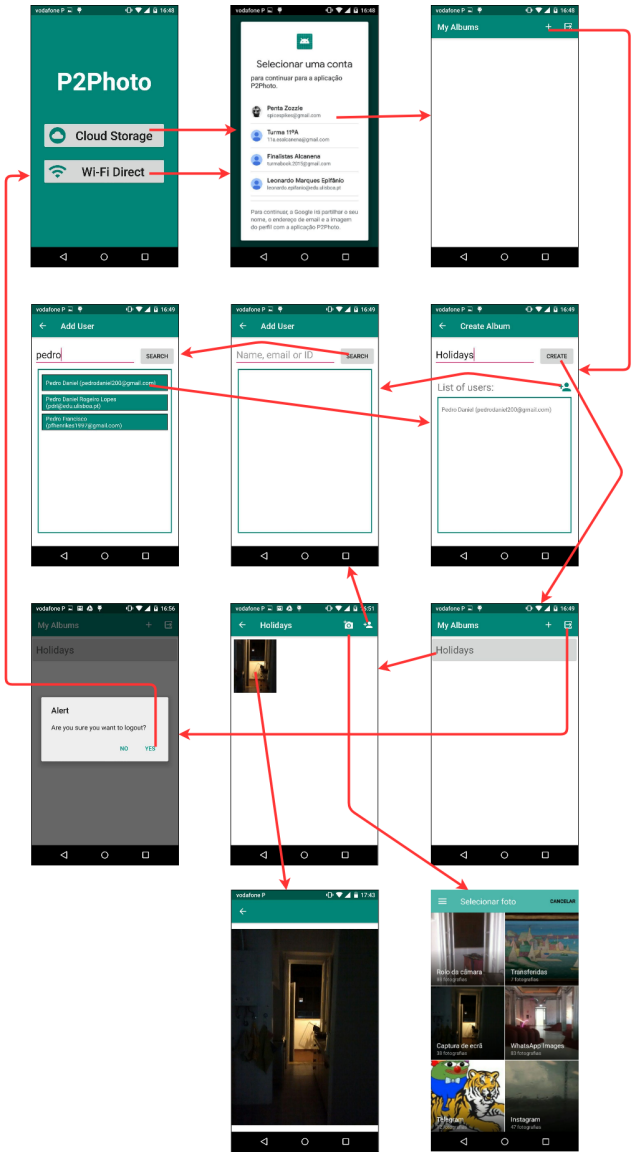

Figure 1: Application wireframe.