Wiride - An application for a carpooling system

Rui André Pinto dos Santos

Thesis to obtain the master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Alberto Manuel Ramos da Cunha

Examination Committee

Chairperson: Prof. Miguel Nuno Dias Alves Pupo Correia

Supervisor: Prof. Alberto Manuel Ramos da Cunha

Member of the Committee: Prof. Renato Jorge Caleira Nunes

October 2014

Acknowledgements

First, I would like to thank my advisor, for accepting me as his student and my thesis, for his input and guidance throughout the last year.

Being this thesis a small portion of a big project i would like to thank the Wiride team and everyone involved in any way that made all this journey possible.

Thanks to my colleagues that took part in discussions about this subject and showed interest in contributing with inputs. A special thanks to Miguel Wahnon Monteiro and Pedro Oliveira that were part of the technical team of Wiride, collaborated with me to build this system and were also present in most of my academic path.

Abstract

People in general sometimes have a hard time conciliating their schedules because of the way they move from one location to another. As students, we think there should exist more suitable transportation solutions to places where transportation networks are short.This thesis proposes a platform to help improve one's mobility through carpooling, a way for car drivers to share their private vehicle with more people with the prime goal of splitting and reducing costs.

Carpooling may be one of the best solutions for when there is no other mean of transportation to a specific location but naturally it is not the only one. Information systems take more and more part of everyone's lives, different services for carpooling with different features begin to compete with existing transportation solutions. Some people start to prefer using new carpooling services over the traditional services represented by taxi services[1].

Wiride aims to promote carpooling by targeting small niches making it easier for people to adhere and use this system. By targeting niches such as universities or companies people will more likely join the service since its users are primarily other people form the same environment.

To put the carpooling system in place, we designed and developed an application server and a mobile application for users to access the carpooling service through their smartphones, additionally the application involves some features that are critical to the service. To test the application from both technical and business point of view we launched, a standalone application for a summer festival with the goal of promoting both carpooling and the application.

Keywords: Wiride, carpooling service, mobile application, Android

Resumo

As pessoas no geral por vezes teem dificuldades a conciliar os seus horários pela maneira como se deslocam de um local para o outro. No entanto todos pensamos que deveriam existir soluções adequadas para as zonas onde as redes de transporte são deficitárias. Este tese propõe uma plataforma para melhorar a mobilidade de uma pessoa atravé do carpooling, que é uma maneira de condutores partilharem os seus veículos privados com mais pessoas com o objetivo de partilhar e reduzir custos.

Carpooling talvez seja uma das melhores soluções para quando não existe outro modo de transporte numa zona específica mas naturalmente que não é a única. Os sistemas de informação têm um peso cada vez maior na vida das pessoas e fazem cada vez mais, parte da vida das pessoas. Atualmente, os sistemas de carpooling começam a concorrer com outros serviços existentes precisamente por usarem os sistemas de informação e aplicações para dispositivos pessoais. Alguns utilizadores começam a dar preferência aos novos serviços de carpooling (como o Uber) relativamente aos serviços tradicionais, representados pelo táxi[1].

O objetivo do Wiride é promover a partilha de boleias segmentando a escolha do públoco alvo sendo assim mais fácil que as pessoas nestes segmentos adiram a este conceito porque se encontram todas no mesmo meio.

Para meter este sistema em funcionamento, desenhámos e implementámos uma aplicação móvel para que os utilizadores possam procurar por boleias facilmente nos seus smartphones. Para testar este conceito e a aplicação em si foi lançado um teste piloto, num festival de verão em que o objetivo era precisamente promover a partilha de boleias e a própria aplicação.

Palavras-chave: Wiride, carpooling, aplicação móvel, Android

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis is about a mobile carpooling application to help people go from one place to another in their daily routine, reducing costs and traveling time. The idea at first was to focus on students but there was no reason to close the system in that way so the concept quickly became to open the application to everyone but promote on niches such as students from one university, employees of a company or other organized committees. Its work derives from a real project but it is only a part of it, the technological part of it. Ultimately it is expected to implement an application with all of the components that serve for the purpose of carpooling. In this chapter and to better understand the document we will briefly explain the evolution of the project and then we will specify what was our role in the whole process. Before that, we will discuss some of the issues that people have in moving from one place to another in their daily routine and how a solution like the proposed one aims to reduce that impact. The last section of this chapter will show the document's plan.

1.1 Carpooling Impact - Hypothesis

This work tries to address one common need, the need to get from one point to another, the need of mobility. Many areas have poor transportation networks, poor meaning that either they have a very low supply, having no public transportation at all in a certain time window, or the available options are too expensive. This means that ultimately people will waste more time traveling or pay a bit more to a more viable solution.

One way to address the mobility issue is to carpool, to join different people in the same vehicle and to get to their destination, splitting their costs all in all in a more convenient way. Only today is possible to make a technologically feasible solution to this problem because only today there

is such availability from a users perspective with smartphones. Solutions in the past have failed because they arrived too soon and had a completely new platform, required a new behavior and users sometimes are not susceptible to change, especially when they are big changes. Working solutions nowadays originally came from facebook groups, a platform where users already were a part of and knew and therefore were not subject to much change.

This work proposes a technological response to facilitate and promote carpooling among interested parties contributing at the same time to a better ecological and health environment and promoting social engagement. The goal is to develop and implement a carpooling platform where users can find lifts to their routes. Around lifts additional modules have to be implemented to be able to provide a sustainable and maintainable service, some of the features include messaging between users, smart locations to easier get location matches, notifications to make the user aware of his lifts status, etc. These features will be described in detail in chapter 3. Note that the proposed solution is not trying to replace the current transportation networks nor it is 'the solution to our problems', it is merely serving as a complementary way to achieve the same goal reducing average wait times and costs. It is intended to develop an online platform for drivers to share their private vehicles and for passenger to search for available seats. The implemented platform consists of a secure application server with all the components to persist the data over the time as well as an Android application to allow users to use and access it anywhere they go through their smartphones. It is intended to use state of the art technology and the best software writing practices.

1.2 Project Evolution

Wiride, previously named iRide[1], is an entrepreneurial project idealized by a group of students. It all started when one of those students found the necessity of a transportation mean where he did not have to wait so much time, take so much time and on top of that, pay a such a fee that does not even begin to compensate for all that wasted time. The author joined the project team when the idea was still being conceived, the current and initial team lacked the technical skills to implement the idea and the author joined the team to not only fulfill this need but also to consult on whether some aspects of the idea were technically possible or not. The basic idea was to develop a carpooling mobile application, something it did not exist in Portugal at that time, something it is very, very recent in Portugal at this time. Later on, there was the need to strengthen the project team with more developers, the more the idea was being established

[1] http://www.dinheirovivo.pt/faz/interior.aspx?content_id=3873347

the more we realized it. So, on February 2013, 4 students from Instituto Superior Técnico[1] and 2 students from Universidade Católica Portuguesa[2] started the development of an idea and an application to help students from across the country meet other students with the same needs to help each other reduce valuable time they waste in the public transport networks. The team of students comprised 2 developers, 1 designer and 3 marketing/management/communication. By this time we were seeking validation of the idea from different programs that could provide us valuable inputs and directions from where to go from then. Wiride project was admitted at Taguspark Business Incubator and was given a place to (co)work in Taguspark, where it spent most of the time designing and developing the application. In March 2013 Wiride signed up for LisbonBigApps, a contest aimed to develop applications that could facilitate aspects of the city life in Lisbon in one of three areas, transportation, social responsibility and turism. This contest lasted for and when the winners were announced, Wiride was one of them. At this time, there was no working system or application but Wiride won several partnerships from sponsors of the contest to help along the way. One very important was Vodafone, which proposed to make launch a Wiride pilot in their festival 'Paredes de Coura' in August. Naturally, Wiride accepted it and started immediately working on the application for the festival and later was the pilot to test both the concept and the system/application. After the festival, the team broke apart for several reasons but mainly because the team was being mismanaged, members were constantly disagreeing with each other and had a different vision for the future of the project.

1.3 Focus of this work

The application development was basically split into two functions, the mobile application (client side) and the server side. All the platform design and implementation was done by two members of the team. Having some experience in mobile this work is focused on the Android application development but the responsibilities of the author also included:

- Overall system design and architecture;
- Technology decisions;
- Mobile application.

These decisions had the aid of the other team member thaw was also a developer. He was the last team member to join the project and he was implementing the server side. Before we

[1] http://tecnico.ulisboa.pt/
[2] http://www.ucp.pt/

started to develop this solution, together we decided how we were going to build it in terms of protocols, models, frameworks to be used, languages, etc. Some of the decisions exposed in further chapters were result of a combined effort to find the best solutions for the problem in a software design point of view. In terms of design, the author was also very present in design decisions because having mobile experience does not encompass only the implementation side of it, it also comprises the whole experience in using and developing for a device with limited computational power and a rather small screen compared to the traditional desktop or laptop. The author did not do the designs himself but he did the implementation on the Android framework and he contributed through valuable inputs concerning element positioning in the interface and overall user experience.

1.4 Plan

The rest of the document is organized as follows. Chapter 2 presents the related concepts to carpooling and the state of the art when it comes to other carpooling mobile applications. Chapter 3 details the functional architecture of the system and its main features. Chapter 4 details the technological architecture, the components of the system, languages and frameworks that were used and an overview of the interactions between client, server and third-party applications. Chapter 5 details the whole implementation of the application, from the models to the protocols, the smaller systems inside the application as well as the interface. Chapter 6 shows the evaluation process, concisely, the pilot beta test. Chapter 7 presents the main conclusions, discusses issues that should have been done or solved differently and possible directions for future work.

Chapter 2

Concepts and Related Work

This chapter will present the main concepts of carpooling, car-sharing and the differences between one another. The main challenges of this practice will be discussed as well some alternatives to workaround those challenges. Finally we will present Uber[1] and Lyft[2], which are the world leaders in this matter. There are some other examples of working emerging solutions to carpool but Uber and Lyft are the most successful by far.

2.1 Main concepts

Carpooling, also known as ride-sharing or lift-sharing, implies sharing a vehicle and it gathers in the same vehicle, users that want to do the same trip and are willing to share their vehicles and costs. This method makes use of private vehicles and splits the travel and maintenance costs among all its passengers. Its goal is to increase vehicle sharing among several users and consequently reduce the number of vehicles that only has one person in it. Some groups of people search and adopt this solution in their daily life, usually it is a closed group from a specific workplace, or a group of students from a university campus or even a mix of the two. Several groups of people use carpooling to make the not so common trip to another city every now and then. It is very easy to find people to carpool with when going, for example, from Lisbon to Porto and vice-versa in the social networks or other carpooling websites. Carpooling brings many advantages, some of which:

- Reduce traffic congestion, particularly in rush hour;

[1] https://www.uber.com/
[2] https://www.lyft.com/

- Avoid parking problems, less vehicles overall means more free parking spots;
- Energy consumption and pollution reduction;
- Reduce travel costs;
- Alternative transportation mode to areas with low transportation services.

There are only a few inconveniences on this type of service: (1) drivers do not have full independence because, the more people inside the same car, the more schedule constraints need to be taken into account, (2) there are many security issues since people do not know each other. The latter is the biggest barrier to entry and the one that people show more concern about, let us take a close look at the challenges:

- **Carpooling with strangers** increases the concerns over security and has been an obstacle sharing a vehicle with strangers although the risk of crime is small. One common used approach to address this issue is using a reputation system that flags problematic users and allow responsible users to build up their trust. Reputation systems like this increase the value of the website/application because it offers some sort of assurance from the other users. Other approach is slowly introducing the system in new niches and/or use the power of social networks to connect people. Most successful applications of carpooling run background checks and require criminal records as part of the sign up process to ensure a minimum security level in this matter.
- **Flexibility** may also be a concern because it can be very hard to accommodate en route stops or change what already is a very rigid pattern. Location and schedule limitation and travel flexibility are the top two reasons people do not carpool[2]. While it is not a barrier when long-distance travelling, the shorter the travel distance is, the less likely drivers are up to detour their own route to catch someone on their way.
- **Availability** is another concern. Carpooling needs a lot of people to get the system working, chances of getting a match for a certain (or similar) trip are extremely low and need a considerable amount of active users. This is something the proposed solution tries to workaround by deploying the system by niches. It is more likely to have matches with people from the same geographic region, school or workplace than with the same amount of people scattered all around.
- **Legal constraints** may apply because essentially, ride sharing lets anyone be a taxi driver and these types of service are getting more and more regulated as they become more common.

A similar concept to carpooling is carsharing, which is sold as an alternative, cheap way to urban mobility. The concept here consists in renting a car, by a single person or a group of people, for a certain amount of time (usually a period multiple of an hour), paid by the hour and the kilometres travelled. Carsharing offers guarantee in parking in the initial and final location (location where is rented) and every other expense is covered (insurance, the car itself, gas and maintenance), it is like having a car without the cost of initially obtaining it. Carsharing is a middle option between having a private vehicle or not having a vehicle giving a moderate mobility and moderate costs.

Since one of the goals is to reduce traffic, emissions and more flexibility for its users, many of the carsharing services use electric cars to reduce carbon emissions and allow to the initial and final pickup points to be in different locations. Carsharing offers advantages to everyone, the user for instance gets to choose the appropriate cars for purpose of the journey (carsharing providers have different categories for renting), reduces the time of travel comparing to other public transport modes, gets access to dedicated lanes and parking spaces. To the community, there is less carbon emissions, less congestion and ultimately lowers the probability of car accidents because it reduces the volume of traffic. Also this service as well as carpooling encourages and stimulates social engagement, which might be particularly important if the people sharing the same vehicle have the same workplace.

Successful examples of implementation of this service both national and international are Mob Carsharing[1], Mobility[2], France Autopartage[3] and many others.

Depending on the annual kilometers one person does with their car, carsharing may even save quite some money[3] on car expenses considering that using this service also covers maintenance, cleaning tasks as well as repair costs. It is a very good option for households that cannot afford to own a car.

These are two distinct concepts that can be combined to reduce costs. People can use a carsharing service for their own purpose and carpooling may be an extra to reduce costs.

2.2 Challenges

As mentioned above, carpooling may have some legal constraints. In fact, in several cities across the world, companies providing these services were banned but ultimately they won the legal battle and were able to operate in those cities. There are a lot of legal issues by providing a carpooling service and they may vary from city to city, country to country because depending on

[1] http://mobcarsharing.pt/pt/default.aspx
[2] http://www.mobility.ch/de/pub/
[3] http://citiz.coop/

where you are different regulations may apply.

The problem is that with this type of service, someone can spend the whole time making a living out of carpooling, basically exercising a taxi service and as a taxi service it must be submitted to taxi regulations and that does not happen and it is not regulated. Here is the problem, when the service starts to be regulated it can destroy the whole model, can no longer make sense to run such system. Additionally there may be also some issues when dealing with accidents in terms of insurance and who/what is covered. These legal challenges only become a problem when the service starts to grow, only when becomes a threat to competitors, such as taxi services, which in some cities have tremendous power.

2.3 Related Work

Currently there are a lot of carpooling/carsharing solutions being developed, some by major established companies others being developed by startups. Uber and Lyft are both startups and are both a ridesharing service but their approach to the concept is very different from one another. Simply put, Uber is a premium service, more corporate with professional drivers that exclusively drive black cars. Lyft on the other hand has more of a community atmosphere where drivers, unlike Uber, are ordinary people with their daily routines, a community driver.

2.3.1 Uber

Without a doubt, Uber is the most successful provider in this business. It is also the most valuable startup in the world at the moment, valued at 18 billion dollars as of June of 2014. Uber is a ridesharing service that connects passengers to drivers for hire, its success is due to the low fees provided. Such fees became very competitive with the traditional taxi services as they claim to have prices of 40% in average lower than these. Uber's added value is in the concept of ride on demand and once it is requested, the user can track his driver's location on his smartphone until he arrives. Uber recently launched[1] a new feature, that it is similar to the carpooling concept we are exploring. Uberpool allows to passengers find other passengers on the way to the same destinations as yours and split the costs accordingly. The differentiation is in the luxury service they provide at relatively low cost, sometimes it may be the same price as a taxi service but that is what you pay for getting a cab service that is very reliable, punctual and comfortable, characteristics that may be hard to find in the traditional taxi services.

[1] http://blog.uber.com/uberPOOLSF

2.3.2 Lyft

Lyft is the biggest competitor of Uber but has a slightly different approach, it promotes ridesharing as a fun and social experience. Just like Uber, payments are made through the application and they get a cut. Again, like Uber, they also have a 'track your driver' features that lets you know exactly when to expect your ride. Just as Uber recently launched the Uberpool, Lyft recently launched Lyft Line, which is also to share rides with other passengers with shared routes. Both startups claim that this carpooling method can save 40% on Uber[1] and 60% on Lyft[2] comparing to the traditional version of ridesharing on Uber and Lyft respectively.

2.3.3 Other carpooling services

The services discussed next are not as big as Uber and Lyft but have they have the advantage of having their own territory, european territory which is an area where these two mentioned services do not operate as heavy as in the United States.

- **Sidecar**[3] is very, very similar to Lyft, they have their own background checks and document verification. The reason there is space for both Lyft and Sidecar is that it is a big market and they are both good tools to get around public transportation services.
- **BlaBlaCar**[4] - It's an international service and recently arrived to Portugal. To increase the security and trust has a rating system, users rate each other and can write reviews, it is possible to read the classification of the other users before the lift takes place. It also has an experience level: users acquire experience by the number of reviews, lifts taken/given, completion of profile (sits on the concept of the more you say about yourself the more people will trust you) and for how long you've been registered on the site. BlaBlaCar has a moderation from real administrators that analyze profiles, photos and comments from the users to guarantee a certain level of quality of service.
- **boleia.net**[5] - It's a portuguese platform, like BlaBlaCar has a rating/review system, users need to setup a profile with a minimum amount of information and also has the ability to connect to facebook and google plus.
- **Amovens**[6] - Pretty similar to BlaBlaCar, its a spanish startup that is trying to expand to Portugal. Unlike BlaBlaCar, already has an application for smartphones (Android and iOS) and

[1] http://blog.uber.com/uberpool
[2] https://www.lyft.com/line
[3] http://www.side.cr/
[4] http://www.blablacar.com/
[5] http://www.boleia.net/
[6] http://pt.amovens.com/pt/

offers a voice system to get in contact with the other users as soon as possible. **Premade Groups** - Social networks allow people to find common ground and connect them. Facebook for instances has countless 'groups' where the only goal is for people to give, search and share their seats from and to many different cities. In these groups theres almost no moderation, people use their facebook profiles to do so.

Chapter 3

Functional Architecture

The system is built in a modular way to easily add more features, we believe that at the moment it aggregates the absolute minimum functionality to have a good experience. Some of the features were thought prior to the development but never came to be implemented so it is expected to be missing some of the features that are present in actual systems and were discussed in the previous chapter. Some of these features that are missing are discussed in Chapter 7.

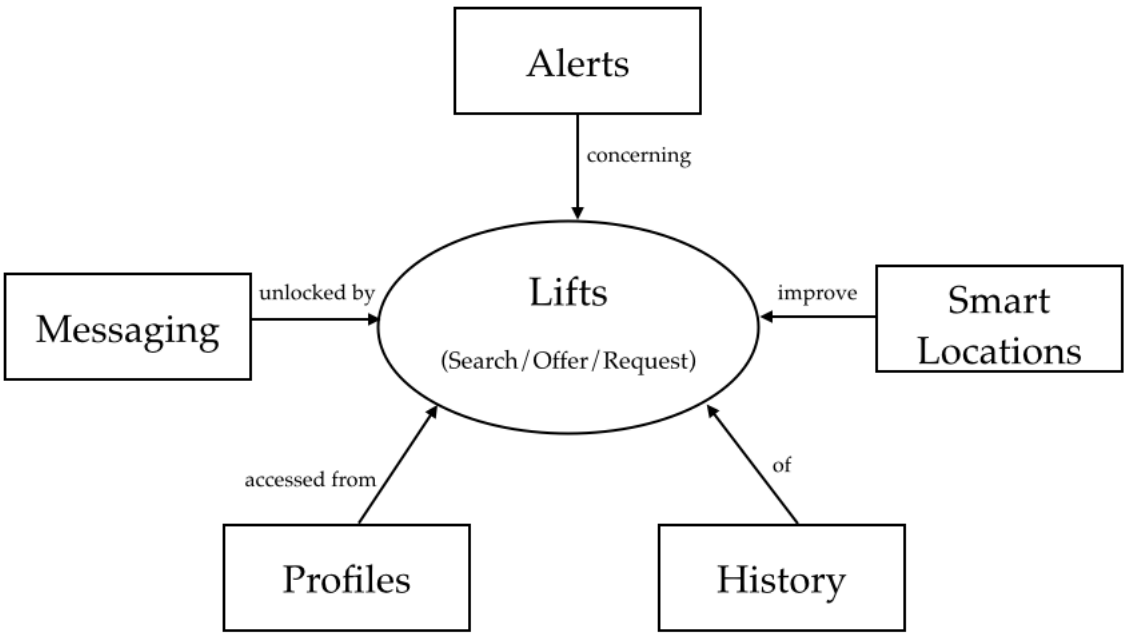

Figure 3.1: Main features of the Wiride application

The system follows a centric architecture like shown in Figure 3.1, lifts being the core feature, the

remaining being features that focus and depend on this one and are designed to make a better and maintainable experience. The implemented features are presented next.

Since Wiride follows a client-server architecture approach, makes sense to separate the logic of both sides. The server is where all data is persisted and main business logic operations are made, meaning that the clients will store a version of the data and that at some point in time, that data in the client may be outdated and like so, needs refreshing, either manual or automatically. The server serves as an interface for the clients to access, interact and trigger (through an API) events for other clients. Let us analyze each functional block and see what side is responsible for what part of the logic.

3.1 Lifts

This is the core of the application, the most basic function to have any carpooling system up and running. Users can make their cars available to carry passengers with the same course or with a section of that same course, let us say they are offering a lift. Figure 3.2 shows the simple process of creating a lift.

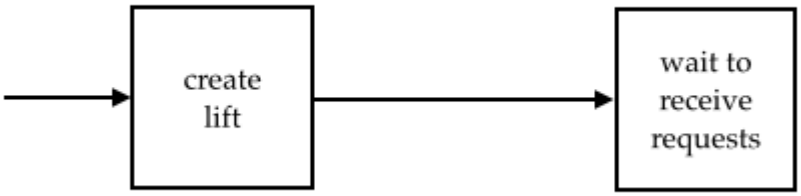

Figure 3.2: Create lift process

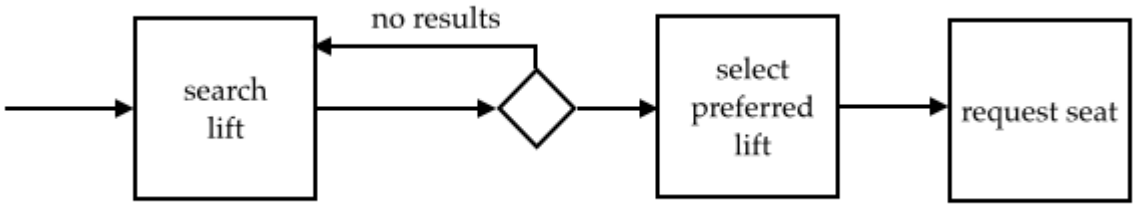

Figure 3.3: Search lift process

Figure 3.3 shows the process of searching for a lift. Once a lift is available for a pre-determined number of passengers, who is defined by the owner, it can be searched for every user if the right

start time is given. When a user searches for a lift, it must provide a time for when he wants to leave his origin point, the system only gives the results that either start at or after that time specified in the same day, or that start a predefined number of minutes before the starting time of the lift. After the search has been made one of two things will happen, the system will present a set of results or it does not present any, because there are not any matching the inserted criteria. In the first case the user can consult the best option and if it is the case, request a seat. In the second case, the user can either make another search with another time or location or simply there is not a suitable lift for him. After the seat has been requested, it is up to the owner to either accept or deny the request but before doing so he can check the profile of the requesting user.

Any user can create a lift and when it does, becomes searchable by other users. Before the lift is created, the client side of the application needs to do the geocoding request to the Google APIs to have the required parameters for both departure and arrival locations. After lifts have been created, users can search and select any of their preference to request a seat as long as there are any results. Requesting a seat will trigger an alert to the driver whom will either accept or reject the request.

Figure 3.4 shows the whole process between two users A and B of creating a lift, searching a lift, requesting a seat and either get accepted or denied in the lift. The whole process is asynchronous as the figure suggests meaning there is no active wait in between of any interactions.

All this communication between client and server uses the JSON[1] format. The fact that mobile clients are involved, working with weaker processors, 3G/4G networks (limited bandwidth) are factors that must be considered in the language of choice. JSON offers the most advantages for data-interchange between client and server over the most common formats (XML, YAML, etc):

- Less verbose;
- Lighter to process;
- Both language and multi-platform compatibility;
- Most APIs that the system needs to communicate already are JSON format enabled or it can be requested to be in JSON format.

3.2 Profile registration

System's users have their own profiles, where they store a description and have important information. The available information is the description provided by the user, name, photo to be

[1] http://json.org/

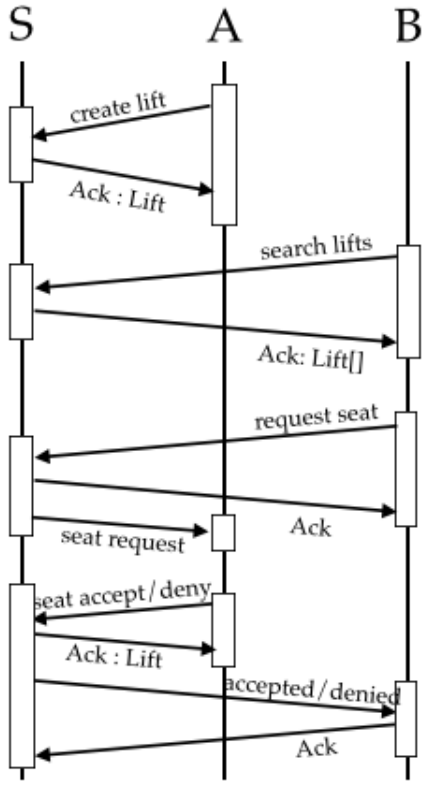

Figure 3.4: Possible process of a Lift

recognized, workplace or education institution the user is inserted in and vehicle information if it is the case of a driver. At the same time these profiles, depending on what info is available, give to other users some validation and trust when it comes to share a trip. Only the most basic information is available for now, additional data fields can be inserted in future work (Chapter 7) to cope with security issues required by carpooling systems. These profiles cannot be searched, they can only be accessed through the lifts, meaning they are semi-public, only users you are currently sharing a lift with can view your profile.

The client is responsible to receive the input from the user and make an HTTP request to the server. The server then proceeds to update his database and reports back to the client (which is passively waiting) with the right HTTP status code.

3.3 Messaging

After the users have shown the interest to share a trip among each other, most of the times they need to talk to each other to sort out the details, either by the use of the messaging system

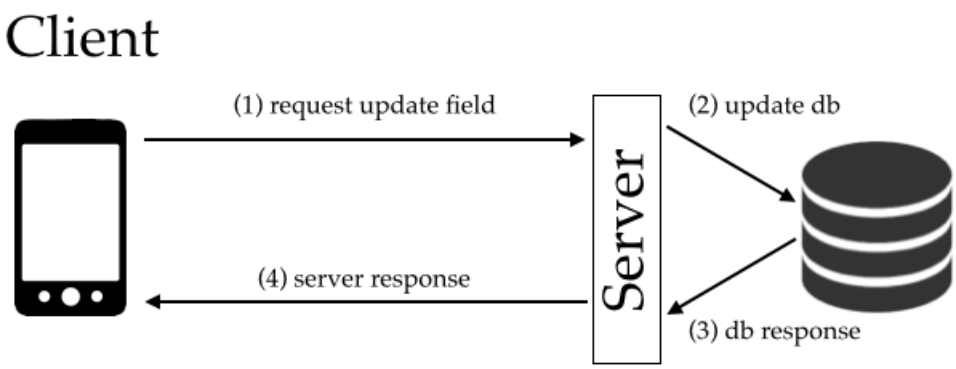

Figure 3.5: Profile update process

the application offers or using their cellphone numbers or any other mean of communication, like email or facebook, but for that they need to have a way to change their cellphone numbers without being explicitly available to any user in the application so it is up to the user to share their personal contact information through the application. Two users, to be able to use the messaging feature, need to share an interaction. When a user requests a seat in a car, both users are automatically eligible to message each other, we call it, they are in each other's circle. In this case, if the lift owner decides to deny the request, they are removed from each other's circle, otherwise, they are removed from the circles only when the lift terminates.

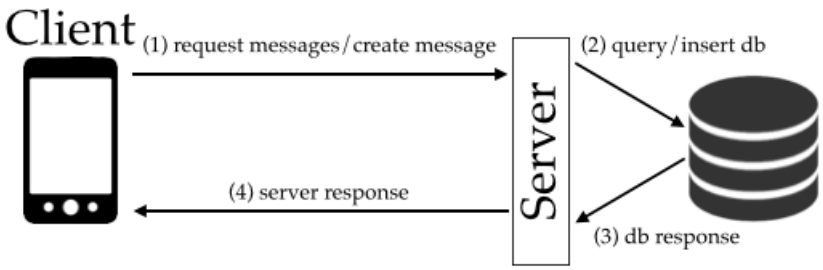

Figure 3.6: Process of creating/requesting a message

Messages can either be created or queried, you cannot change the value of it. Like so, users can either request the messages present on the server or request to create new ones and once it is created, the server returns the message object so the client can update its internal database. So all the server does it storing and returning message objects, it is up to the client to read those messages and check whether they are new and should notify the user or the other way around. In the next chapter we will show a little tweak we did to improve both performance on the client, the server and to reduce the network overhead.

As shown in Table 3.1 some interactions trigger events that may add one user to other's circle.

Table 3.1: Events/Actions that trigger a circle add/remove event

Actions/Event	Circle Add	Circle Remove
Accept a user in a lift	X	
User joins a lift	X	
Kick a user from a lift		X
Cancel a lift		X
Leave a lift		X
Lift ends		X

3.4 Smart Locations

In order to reduce mistakes the system must be prepared to receive different names for the same location, to give the same identification key to different values because different users can input different names for the same location. For example, 'Instituto Superior Técnico' and 'Avenida Rovisco Pais 1' point to the exact same location, it is different values for the same place and the system must understand those values as the same location to match any users using these two different types of nomenclature, this is done through geographic coordinates and it will be explained in Chapter 5. We leave these requests to be made on the client side instead of the server because it is independent of any data that the server might be holding so there is no point in overloading the server with this task. After the requests have been made the client uses this information to complete the request to create a new lift on the server.

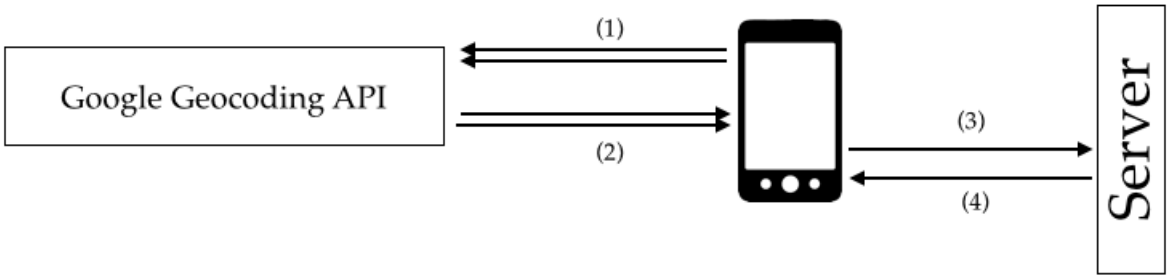

Figure 3.7: Geocoding/Reverse geocoding process

3.5 User Alerts

When important actions occur, users get a notification warning for those, for example, that their status in the lift has been changed, either because his request to join the lift has been accepted,

or denied, or the lift itself has been cancelled. This is important because, smartphone users do not have time to check all their applications for updates, instead, the applications must notify the users if such updates exist or else they will just go unnoticed.

In this functional block there is actually no communication for the purpose of sending an alert. Instead, the client periodically asks the server if there is any update on his user. Once there is, it proceeds to update his database and pop an alert if necessary. At this point, every update reflects in an alert to the user but this does not have to be this way. If other minor updates were to be included in this updating service (lets call it that way) they were to be used only to maintain consistency between client/server and not to alert the user of some event, for example, an avatar change for a person in the user's circle or an update to a lift where other person joined it. The user does not need to know exactly when that happens, but it is good experience to be already reflected on the interface without having to update manually.

3.6 Lift History

Lift history is available mostly for consultation of previous runs, it is a history of lifts where the user has been either a passenger or driver, in the latter case, as long as more users participated in the lift otherwise it is not stored as a lift. The use of this is for the user to be able to check the details of previous runs like passengers or the time it took place.

Similar to the examples above, at the time of login the client requests the history of his lifts. But as long as the user is logged in into the application the client alone manages his history. Whenever a new lift takes place, which the user is in it, it adds itself to the history and whenever it is cancelled, removes it from the history.

Chapter 4

Technological Architecture

The developed system uses classic client-server architecture and as such can be naturally broken down into two pieces:

1. Client - compound of all the smartphones that use the developed application;
2. Server - one single application server accessible through a REST API[4][5] which is a common architecture to build web services.

In this model, clients make requests to the services provided by the server, clients are responsible for initiating contact with the servers and keep requesting any updates - an alternate method is proposed later in Chapter 7 where some of the responsibility is moved to the server to reduce network/computing operations in both client and server, this method makes use of the push notification service provided by the operating system provider.

4.1 Client

The client in the designed system is the application running on a smartphone. Smartphones are a tool today that everyone take to everywhere because in present days they are an indispensable part of people's lives. There are different types of smartphones that are relevant when it comes to designing applications and they can be sorted into their operating systems.

This is important because, decisions have to be made when designing to different operating systems. For this work, the application was developed for the Android operating system, using the Android SDK (Software Development Kit) standard language, Java. This means that at this

Period	Android	iOS	Windows Phone	BlackBerry OS	Others
Q2 2014	84.7%	11.7%	2.5%	0.5%	0.7%
Q2 2013	79.6%	13.0%	3.4%	2.8%	1.2%
Q2 2012	69.3%	16.6%	3.1%	4.9%	6.1%
Q2 2011	36.1%	18.3%	1.2%	13.6%	30.8%

Figure 4.8: Smartphone operating systems market share

point, the application is not cross-platform, does not run on anything else besides smartphones with the Android operating system.

This application is limited to the Android operating system so it is not available to everyone but Figure 4.8[1] shows a clear trend to adopt smartphones running the Android operating system and every other operating system has a declining market share. This is a global market share and needs a careful reading, Android reaches such market share in developing countries because their devices are the cheapest when comparing to iOS for example. If we compare the market share in the US alone, which is the country where most carpooling services launched and operate for quite some time now theres a significant difference between the global and the US national market share, 85% to 62% on Android and 12% to 33% on iOS respectively[2]. This analysis shows that depending on which country you are developing your application Android is always a good option to deliver first and test the market but to reach global scale it also must be developed for other operating systems, iOS particularly, in the US for example, one third of smartphone users it simply is not negligible.

One option considered during the initial phase of the development was to develop the client side using a specific framework. Frameworks have the advantage to accelerate the development process because they have pre-defined easy-to-use libraries that later, can compile for more than one platform at the same time, reducing development costs because you only need to develop the application once, instead of one for every operating system. Using these frameworks also have a downside, which is, you are bound to the framework updates, if the operating system has an update, you need to wait for the framework to update in order to have access to that new feature. One other disadvantage is that you are using the same design and same principles across all platform and different platform may require different user experiences and/or different implementations. Also, it comes with a cost of performance, naturally the application will perform

[1] http://www.idc.com/prodserv/smartphone-os-market-share.jsp
[2] US smartphone OS market share - http://bgr.com/2014/07/01/android-market-share-2014/

better if it is written in its native language and compiled with the appropriate compiler.

4.2 Server

As for the server side, it was built with a simple LAMP[6] stack, which is a software stack that stands for Linux, Apache, MySQL and PHP and combined can build powerful, reliable and stable applications. The benefits of using a LAMP stack include: it is easy to install and set the environment, it is easy to deploy since PHP is a standard module of Apache, it is portable, most host and cloud providers support PHP/MySQL on Linux or have the setup available right away and one of the biggest benefits is that it can be developed on a personal computer and deployed onto the web.

1. **Linux -** It is the operating system that ensures the rest of the components run efficiently. It is very light (which is important when running on public cloud servers), cheap (also important) and the biggest features might the compatibility that it offers cross-platform.

2. **Apache -** This is the web server, it is one very important component of the stack since every action will go through the web server and it is the first possible bottleneck of the communication flow, the number of requests it can serve might be determined by the efficiency of the web server serving them. Also, apache offers great stability since it is the most common web server.

3. **MySQL -** Is a lightweight relational database, it is used to persist the data about all the models (described next section), these models are defined as resources.

4. **PHP -** It is the glue that holds the database and the web server and allows handling and manipulating the content on the database. It is in this layer that all the 'business logic' is written and defined. Like the client side, there are several frameworks that ease the development of applications, in this case, server-side applications. Unlike the frameworks on mobile where the concept is one framework to deploy on all operating systems, the frameworks to write server-side applications are more specific to the language and are usually written in that same language so there are no compatibility issues and when it comes to performance, that framework probably is better written just because there are more people behind it thinking of ways to improve the framework and the performance of it. The goal is precisely to let developers focus more on the application logic rather than the little details that can make the development phase very painful.

According to DB-Engines[1] databases in Figure 4.9 are the most popular. The ranking is calculated through a set of metrics[2] such has general interest in the system and frequency of technical discussions about the system. Out of the top three that have similar ranking scores, MySQL is by far the cheapest. MySQL is free and runs on any operating system without any hassle, both MSSQL and Oracle are licensed and proprietary. MySQL is also capable of handling a high usage and hold several records without compromising performance.

223 systems in ranking, September 2014

Rank	Last Month	DBMS	Database Model	Score	Changes
1.	1.	Oracle	Relational DBMS	1466.91	-3.95
2.	2.	MySQL	Relational DBMS	1297.14	+15.92
3.	3.	Microsoft SQL Server	Relational DBMS	1208.87	-33.62
4.	4.	PostgreSQL	Relational DBMS	255.79	+5.94
5.	5.	MongoDB	Document store	240.98	+3.63

Figure 4.9: Database rankings

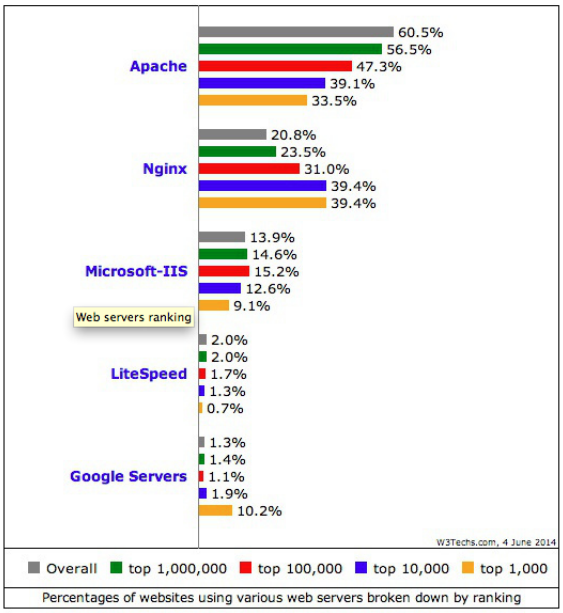

Figure 4.10: Webserver percentage usage rankings

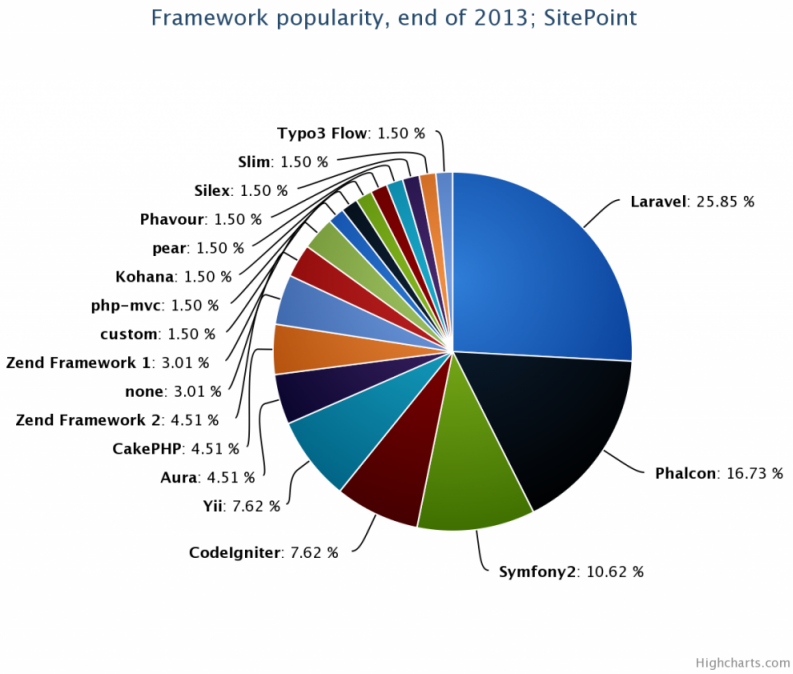

Figure 4.11: PHP framework usage rankings

Figure 4.10 shows the most used web servers, apache is a clear winner overall but on the top 10.000 most ranked websites by searches Nginx takes the lead. Nginx is becoming more popular because it performs better than Apache, it has a lot less features than apache making it more light and faster. Apache comes bundled with the traditional LAMP software stack and that may

[1] http://db-engines.com/en/ranking
[2] http://db-engines.com/en/ranking_definition

be one of the reasons it is so popular, another one is because it is a pioneer in web servers.

Sitepoint[1] did a survey concerning PHP frameworks to PHP developers and the results as shown in Figure 4.11 show that Laravel, the chosen PHP framework for this thesis, is the most used PHP framework on the web. In this survey, all results that had people with only experience with this framework or had zero proof of proficiency were discarded, so the results are not influenced in that way. Laravel is a PHP written framework that makes the implementation less difficult because it has so many handy features, these were the reason it was chosen:

- **Uses the Model View Controller paradigm.** Since the client-side will be built also with this paradigm, makes it easier for the client and server to talk to each other.
- **Supports secure connections.** Security in communications is a concern in all applications, but this one in particular is also an issue because it has information of the people's locations, where they are traveling to, who they traveling with and other private information.
- **Database abstraction.** One good thing about this framework is that to create the database tables, you only need to define the models and its properties and the framework will migrate those properties to a proper table in a MySQL database with some additional fields like 'created_at' and 'updated_at', which are timestamps concerning the object that come in handy. Also, to access and manipulate those tables you modify the model object and those changes will be persisted in the database making this database layer very abstract.
- **Routing system.** This routing system allows creating HTTP endpoints very easily. After declaring the route you only need to invoke the proper controller with the parameters and you have your client communicating with the server almost effortlessly.

These bullet points are important because it lets developers focus on developing the application itself instead of concerning with other factors that most of the times are a time sink.

[1] http://www.sitepoint.com/best-php-frameworks-2014/

Chapter 5

Implementation

Taking a look at the architecture and after having some more in-depth of how the system is designed in terms of communications, models and database let us see how some of these blocks are really implemented in such a way that make all the interactions work.

Android programming is very much event based meaning that actions done by the application are triggered by an event. For example, launching a new screen is triggered by taping a button, further more, creating a lift on the application triggers an HTTP request that will create the lift on the server and this lift can subsequently be searched by others users. When a seat is requested by any user it is triggered a new notification to be sent to the owner. Most of the actions done by an user, trigger an event that will create a notification on the server to another user.

For this chapter a brief introduction to Android will be given, showing the main components offered by the Android SDK to understand how they fit into the developed solution. Next we will be show how we approached the problem with the software solution explaining how the components in the previous chapter were implemented. Then, we will take a look at the design implemented as well as the importance of such thing in mobile applications. Last, we will briefly explain the environment the application was developed, mostly during and after the beta-test (explained in Chapter 6) was taking place.

5.1 Android - Main building blocks

As explained on Chapter 4, the framework used to develop the client side was the Android SDK. Android SDK has four base components, used to build applications, each of which has its special use and are used for different things:

1. **Activities -** They represent a single screen containing the user interface. Only one can be active at the time.

2. **Services -** Is a component that runs in the background. Usually used for long-running operations such as downloads and other long network operations or even play some background music. This component does not provide an user interface and can perform these operations while other applications are active, it can perform these operations after the application has been closed.

3. **Content Providers -** Is a component that manages data. There are ways in the Android framework to store data, using for example the file system, an SQLite database or via the SharedPreferences, which is a way Android offers to save key value pairs. None of these methodologies is able to share the data with other applications. With Content Providers you can do just that through a set of APIs.

4. **Broadcast Receivers -** Is a component that catches an announcement sent from the system or other application (this case is very, very rare). For example, when the device connects to the Internet, the system broadcasts that event and if your applications has registered to receive that broadcast it gets notified whenever that event occurs.

All these components must be properly declared in the AndroidManifest.xml file. Each application must have this manifest. This file has all the relevant properties of the application, every component above described must be declared in the manifest with a property that points to the class that represents the component (Activity, Service, Content Provider or Broadcast Receiver).

Besides all these components that are the main building blocks the Android operating system also offers a set of APIs to access the device's features like the GPS, the WiFi, Camera and so on. One other thing that it is important in the Android SDK is the way to create interfaces, it is possible to create the interface in separate XML files and then bind elements in these files to the Activitiy component and once they are bound it is possible to manipulate its values and elements on the interface.

5.2 Wiride Approach

At Wiride we used Model View Controller software architecture in both client and server side, to facilitate object communication between the two sides. This type of architecture fits perfectly into the Android operating system because of the way Android separates the interface design - through the XML files - and the business logic through the .java files. Figure 5.12 shows the

box the application is and its components inside. Like explained before, Android has four main building blocks that make possible to build Android applications, in the Figure 5.12 those four components are outlined in different colors:

- **Blue** indicates the 'Activities' component, they manipulate the elements shown in the user interface. These elements are defined and arranged in the XML files below the Activity layer and are called 'Views'.
- **Green** indicates the 'Broadcast Receivers', they are responsible to run and optimize the service, which is a crucial part of the application.
- **Red** indicates the 'Service' component, the 'NotifyService' is responsible to poll the serve for any updates and if any exist, store in the database so the activities can reflect those updates in the user interface.
- **Purple** should be the 'Content Provider' but as stated above this application does not use any content providers but instead uses a SQLite database to persist the objects throughout the application lifetime. It will be explained later in detail.

All these components have access to each other either through the database in terms of resources and via the application scope. All Android building blocks have a 'Context' type variable which is a reference to the application context. The database is not a main building block so the application itself has a reference to this database, this way all the components can access it when it is needed.

After the basic functions of the carpooling system were defined what we did before start coding were the following steps. Naturally along the development phase some models had to be changed, some protocols had to be redefined.

5.2.1 Defining the models

There are actually few models to work with in this application. Let us take a look at them:

- **User -** Represents the users of the application, both passengers and drivers. It has an incremental unique id, first, last and a username provided by the person creating the account as well as a description. The description is just something the user can input so people can recognize them. Additionally, the user has an 'avatar_url' property that is an URL pointing to the user's avatar. These avatars are stored in the filesystem in both server and smartphones.

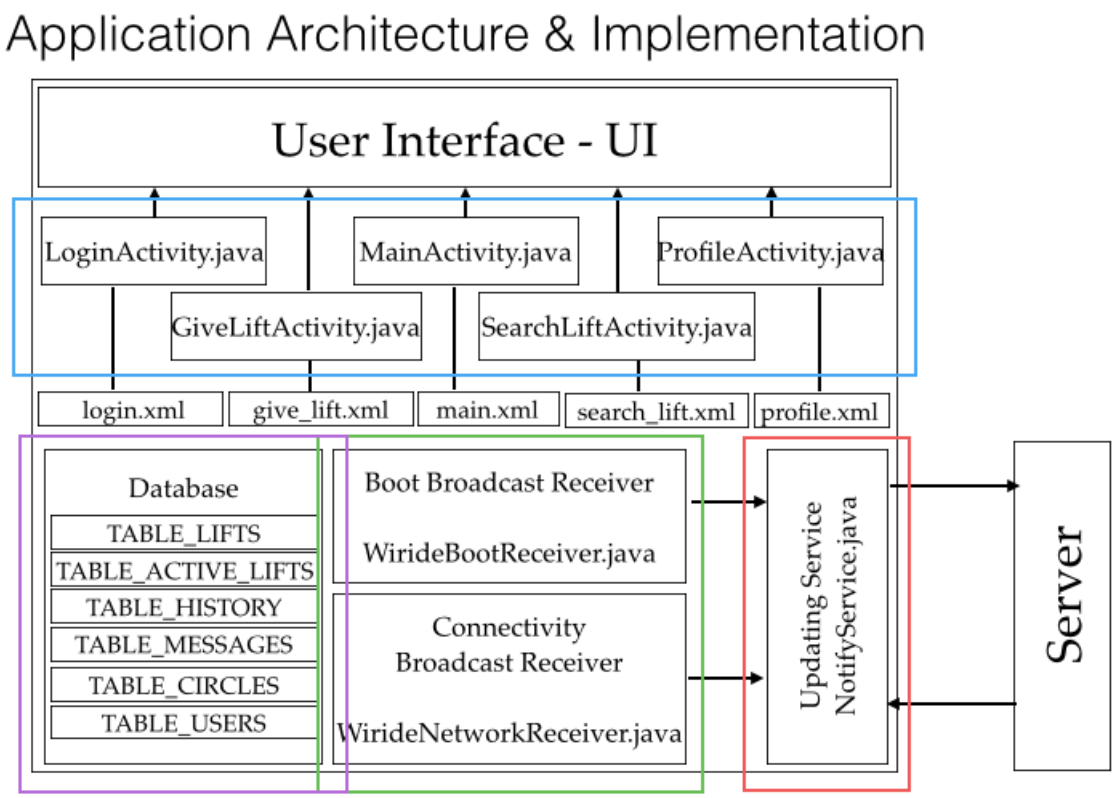

Figure 5.12: Wiride application building blocks

- **Lift -** Represents every lift that is created throughout the application. Let us take a look at each property:
  - **liftId:integer -** Like the user id, it is the lift's own unique incremental id.
  - **driverId:long -** The id associated with the driver of this lift.
  - **seats:integer -** The number of maximum seats that the driver designated for the lift.
  - **seatsTaken:integer -** Seats already taken by the passengers. Used together with seats to check whether there are any seats available to request or not.
  - **date:String -** Date for when the lift is taking place. It is a string in the 'Y-m-d H:i:s' format.
  - **departure/arrival:String -** User input string for departure/arrival locations respectively.
  - **passengers:String -** Passengers ids concatenated with semi-columns.
  - **user_status:integer -** The status of the user making the request before the lift.
- **Message -** Just like 'User' and 'Lift', 'Message' has its own unique id. 'from_user_id' and 'to_user_id' properties are the ids of the sender and recipient of the message respectively

```
public static final int USER_STATUS_NOTHING = 0;
public static final int USER_STATUS_PENDING = 1;
public static final int USER_STATUS_ACCEPTED = 2;
```

Figure 5.13: Possible user status

and ‘message’ property is the value of the Message, what the user has input.

```
8  public class User {
9
10     long id;
11     String firstname;
12     String lastname;
13     String username;
14     String avatar_url;
15     String description;
16     Context context;
17
18 }
```

Figure 5.14: User model

```
8  public class Lift {
9
10     int liftId;
11     long driverId;
12     int seats;
13     int seatsTaken;
14     String date;
15     String departure;
16     String arrival;
17     String passengers;
18     int user_status;
19     Context context;
20
21 }
```

Figure 5.15: Lift model

```
4
5  public class Message {
6
7      long id;
8      long from_user_id;
9      long to_user_id;
10     String message;
11     String created_at;
12     String updated_at;
13     Context context;
14
15 }
```

Figure 5.16: Message model

All the models have a ‘Context’ type variable, which serves for the purpose of connecting the model to the Database. Context is a class that contains information about the application environment, it allows access to application-specific resources, such has database because the database was instantiated with this same context as a parameter.

Later in the development process the ‘Storable’ interface was implemented to abstract as much as possible this layer to the models. The interface Storable declares the create, update, destroy and load which are the four basic functions of objects used in relational database applications - CRUD, create, read, update and delete. With the context variable, the object models can access the database and update their properties themselves. The update operation has two methods, one of each data type just because updating one type in the database is different from the other.

And that is it, there are some other model classes that are just auxiliary to represent some data or objects such as ‘Coordinates’ that not only has the geographic coordinates, but also the formatted address.

5.2.2 Defining the communication protocols

Looking at the architecture in the previous chapters it is clear the kind of protocols the application follows. The application is a web-based application in the way that the server is available through

```
public class Coordenadas {

    String latitude;
    String longitude;
    String formated_address;

}
```

Figure 5.17: Coordinates model

```
public interface Storable {

    public void create();
    public void update(String key,String value);
    public void update(String key,int value);
    public void destroy();
    public void load();

}
```

Figure 5.18: Interface implement by the models

HTTP requests but it has a wrapper around it what we can call it 'Android application', the application itself. Being a web-based application the protocols are based in the HTTP requests and HTTP codes, for simplicity we used four of the HTTP codes to help with communications between clients and server.

- **200 OK -** The request has been fulfilled. This code is used when a request is made. After reading it, the client either reads and computes the output as a JSON format message or just serves as an acknowledgement as the server received the message. For example when a user searches for existing lifts the server returns the HTTP 200 OK code as well as an array of lifts to be parsed by the client side. On the other hand, when drivers accept a seat request in their lift, the server outputs an HTTP 200 OK code but with an empty output, for the sole purpose of acknowledging that the request was received and processed.

- **201 Created -** The request has been fulfilled and resulted in a new resource being created. The server acknowledged the request and either a lift or a message was created because users can only create these two data types.

- **400 Bad Request -** This code is used for when the server due to malformed syntax could not understand the request. For simplicity, at Wiride we also use this code for when other error occurs on the server because either way the request needs to be re-written (in terms of parameters) and re-sent.

- **401 Unauthorized -** The request requires authentication or the provided authentication failed. Wiride uses an Open Authentication (OAuth) system. This means that receiving this code either the 'access token' is missing or it is no longer valid and a new one should be. To request a new access token the refresh token should be used so whenever this occurs, automatically without the user noticing it. This part was not implemented, instead whenever the applications reads the HTTP 401 Unauthorized it forces the logout on the application so the user logs in and forces a new access token generation.

5.2.3 Defining the (REST) API

A REST (Representational State Transfer) is an architecture that when applied to the web services, which is the case, makes it easier (as well as more readable) to make CRUD operations on remote resources with the use of the HTTP methods (GET, PUT, POST and DELETE). For simplicity, in Wiride only GET and POST was used and the API is not fully REST, as in the syntax is not quite as it should, only because by the time the API was being designed and implemented we have not mastered the REST APIs concept. Following the REST architecture, we will show how the resources were accessible on Wiride. Note that these paths are always preceded with a base URL that points to where the application server is being served. In this work's case the base URL was the same URL of the application[1].

To better understand the endpoints and how they work lets first take a look at the resources available:

- **Lifts** - It is possible to create, list, request to join, leave and cancel.
- **Account** - This resource holds accessibility to operations that are bound to the user and his own created resources like profile information or the lifts he created.
- **Messages** - Users can create and request messages to/from others users as long as they are in the same circles.

Users could also be considered as a resource, but they cannot be specifically requested. Information about the users that is shareable is bound to the lift as long as there are passengers/drivers or to the circles if any.

Below we will present a list with the main endpoints accessible through web services that enable the clients to request updates on the resources as well as a brief description of the operation. Note that I had some implementation issues so, like mentioned above, this API does not provide a pure REST experience. In Chapter 7 we will make a comparison of how some endpoints are implemented and how they should have been implemented.

A few notes on the parameters and endpoints:

- _id_ means it should be replaced by the resource id;
- Coordinates have the format 'lat long', for example '38.7273676 -9.366555';
- Every request made through the API needs to include the access token required by the Open Authentication system.

[1] http://www.wiride.com

Table 5.2: Endpoints for the lift resource

Lifts			
Endpoint	**HTTP Method**	**Parameters**	**Return**
"/lifts/add" creates a new lift	POST	time:String seats:int departure_coords:String departure:String arrival_coords:String arrival:String	Lift
"/lifts/nearby" search for lifts	GET	time:String departure_coords:String arrival_coords:String	Lift[]
"/lifts/join/_id_" request to join lift	POST		[]
"/lifts/leave/_id_" request to leave lift	POST		[]
"/lifts/cancel/_id_" cancels this lift	POST		[]

Through Laravel's routing system all these endpoints (or paths if you will) can be written in one file and then redirect to the appropriate controller.

5.2.4 Defining the database

As it was mentioned above, of all the main building blocks of Android, the proposed Android application only does not use Content Providers. Content Providers do not provide the necessary features the application needs, and makes the content shareable between other applications, which is something that does not make sense in this application.

To store and manage data we fell back on the three different options to store data on an Android application, each of which is specific to each type of data:

1. **SharedPreferences -** stores data in key-value pairs. It is used to store the some application specific variables such as 'logged' that indicates if the user is logged in the application or not. It is used to easily persist data throughout the application. The user details are also stored in these preferences and are variables like, id, username, first and last name, access token, etc. Basically, at least the same properties defined in the 'User' model.

2. **Internal/External Storage -** To store private/public data on the device memory/external

Table 5.3: Endpoints for the account resource

Account			
Endpoint	**HTTP Method**	**Parameters**	**Return**
"/account/allow" accepts a seat request	POST	user_id:long lift_id:long	[]
"/account/deny/" denies a seat request	POST	user_id:long lift_id:long	[]
"/account/update_profile/" updates the user profile	POST	description:String car_type:String car_fuel_type:String drivers_license:String car_displacement:String	[]

Table 5.4: Endpoints for the message resource

Messages			
Endpoint	**HTTP Method**	**Parameters**	**Return**
"/account/chat-users" get the circle users	GET		User[]
"/account/messages" request all the messages	GET		Message[]
"/account/send_message" creates a new message	POST	message:String to_user_id:long	Message

storage (smartphone's SD card). Internal storage is used to store the picture of the users and their cars. When the application creates a user object, it proceeds to issue an HTTP request for the provided avatar URL, defined in the 'User' model, and save the image on the device.

3. **SQLite Databases -** The Android framework provides SQLite as a way to store structured data in a private database. Wiride uses this storage mechanism to store all the information about lifts, users, circles, messages, notifications and alerts. Each of these models has its own database table and Android provides a very simple way to display this tables in the interface if needed. To simplify the lifts management Wiride divides lifts in three different categories and as such, each category has its own table, this is so it is easier to update the lifts in the implementation and efficiency point of view. We will briefly describe what is each table and how the table is updated/managed:

- **TABLE_HISTORY -** This table has a compilation of all the lifts that are past. Updates to this table is simply creating a new row, no need for updating every row since there will be no changes because the lift has been finished.
- **TABLE_ACTIVE_LIFTS -** This table contains all the lifts the user is currently in, whether he is a driver or a passenger. Updates are constantly being made on active lifts, requiring updates on some rows very often.
- **TABLE_LIFTS -** This table contains all the lifts that result of a search. This category does not need as much information as other lifts because the user is not part of them. There are no updates to the rows of this table, when the search yields results they are stored in this table, when another search is made, another set of results is displayed and the old ones are deleted because they do not matter, so the most operations being made is truncate and insert.

```
static final String AVA_DIR = "/Android/data/com.wiride/avatars/";
```

Figure 5.19: Base path for the avatars stored in private memory

Optimizing database access is important because these kinds of operations, operations that involve disk reads/writes, block the main User Interface (UI) thread that is responsible for showing the interface in the screen. These operations are usually run on a background thread to avoid UI blocking but the user still has to wait for these operations to finish before he sees them on the interface. Lifts were separated in three tables because of the way they are differently managed, it is possible to make changes on one table without worrying to mess the other table. The first table only receives inserts, the second, either has new insertions, deletion or updates and the last one only inserts and deletes. In the evaluation chapter we describe a method to better handle the case of the second table that either needs updates, insertions or deletes, the other two tables are a no-brainer when it comes to handling the common operations.

5.3 Updating Service

Every twenty seconds the application will ask the server for any new notifications. This is done via the 'NotifyService' class that extends from the Android Service, which is one of the main building blocks. This service is always running, even when the application is forcedly closed. A different implementation for this notification update is proposed in the future work chapter. The service will always request the same endpoint for updates. Most of the times there are not any updates

and the output is null, the service goes back to sleep. When there is one (ore more) updates the services does the parsing and acts accordingly. In order to identify the new update a code has been defined for each type of update. This also serves so the application knows what additional fields to fetch from the incoming JSON message if it is the case.

```
public static final int A_ALLOWS_B = 1;
public static final int A_DENIES_B = 2;
public static final int A_ASKS_B_TO_JOIN = 3;
public static final int A_LEAVES_B_LIFT = 4;
public static final int A_CANCELLED = 5;
public static final int A_KICKS_B = 6;
public static final int A_FINISHES = 7;
public static final int A_MESSAGES_B = 13;
```

Figure 5.20: Notification types

All notifications are parsed the same way except for the type 'A_MESSAGES_B'. The rest of the notifications are just an update to a lift, the application just needs to update on the database and create an alert depending on the type. When the user taps the notification he will know what was the update and it will be already reflected on the interface. The 'A_MESSAGES_B' notification is different because it has more information like the message model that needs to be inserted in the database. For the server to know which notifications he should return to the prompting user, every notification has an unique id. When the applications requests for an update via the service, it provides the 'last_id' parameter, which is the last id that the application received and has stored on the database. This way if the server only returns notifications with higher ids than the provided one, and since the ids are incremental, we can guarantee no duplicate notifications alerting the user.

5.3.1 Running the service

This service is very important to synchronize data between server and clients. That is why it must be always running, in order to do this, a new instance of this service is launched, if is not already running, whenever the applications runs. Also the application subscribed to the 'BOOT_COMPLETED' broadcast sent by the Android operating system whenever the device finishes booting. This broadcast receiver (WirideBootReceiver.java in the Figure 5.12) is registered and the only thing it does is run a new instance of the NotifyService to ensure it is running even before the user launches the application for the first time after a boot.

5.3.2 Optimizing the service

Smartphones today have various issues to consider, one of which is battery life. It is good practice to stop your application from doing unnecessary work. As it was mentioned before this service is always active, every twenty seconds it wakes up, does some work and goes back to sleep. Since what the services do is invoking a remote web service, there is no point in running the service if the device is not connected to the Internet, there is no point in keeping the service live at all if the device is not connected to the Internet. To prevent this from happening the application subscribed to another broadcast sent by the Android operating system. Every time the connectivity of the device changes, Android alerts whatever applications have subscribed to this broadcast. Through the 'WirideNetworkReceiver' the Wiride application knows when the devices is connected or disconnected to the Internet. This way, whenever the application catches a broadcast indicating the device has disconnected from the Internet, the applications shuts down the service, saving battery. Similarly, when the application receives a broadcast indicating otherwise, it starts the service. This way, the service is only running when it specifically has to keep the data synchronized.

5.4 Login

As mentioned above, it is at the login that the user authenticates himself has being a determined person, and it is at the login that the user authenticates before the server to have access to its resources. To achieve this the application implements the rather recent and well-known Open Authentication[1], which is a authorization protocol to enable users to obtain access to the web services provided by one other server. To understand the communication flow we need to understand the different components involved.

- **Authentication Server -** Responsible for validating the credentials and issue tokens to the application server.
- **Application Server -** Responsible for providing the web services that access the applications resources.
- **Resources Server -** Where the data is stored, can be an abstraction for a MySQL database, which is the case.
- **Application -** The entity trying to access the resources, users for the most part.

[1] http://www.openauthentication.org/

Technically the three server components can be assembled in just one machine, which is the case and still provide the same strong and reliable authentication method already with future support to enable third party applications to access the Wiride resources.

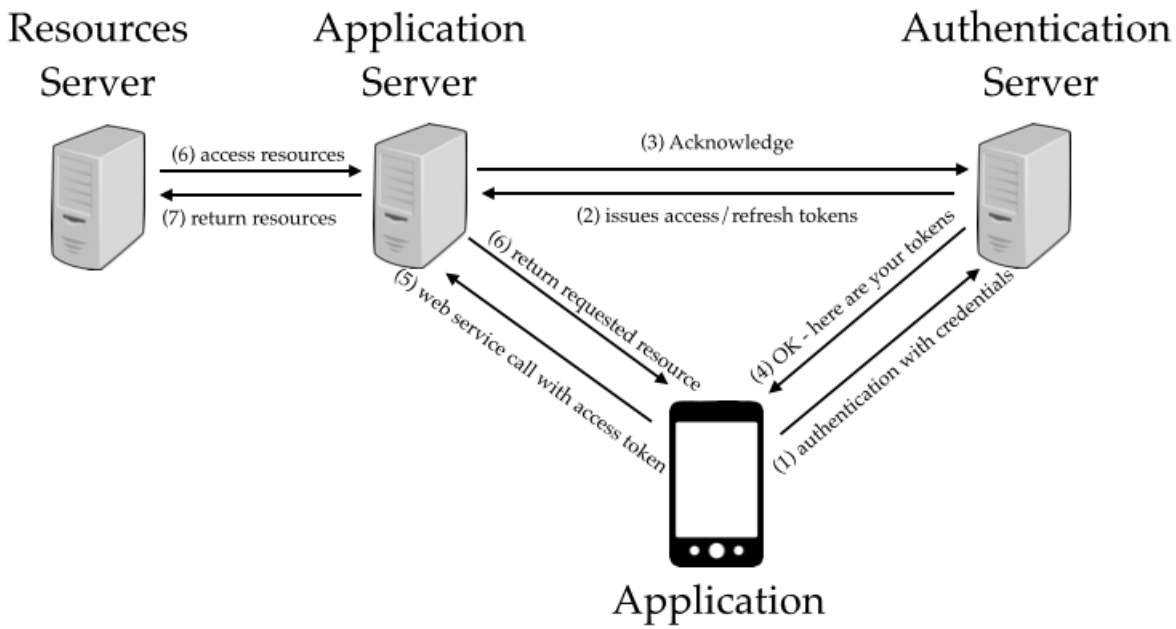

Figure 5.21: Open authentication protocol

First, the application needs to be registered in the authentication server to be able to request tokens that will be issued to the application server. This is done by hand in the server and the application itself ships with the secret needed to request these tokens. After being registered, the application authenticates the user with the credentials they provide in the authentication server that returns a set of tokens and proceeds to send these tokens to the application server so they can validate further requests himself. Theoretically this is so authentication server handles with authentications alone and takes the load off the application server, also, connections with the authentication are more secure since they store users passwords while the application server only stores a token that may be refreshed whenever both the application or the user sees fit. The next time the applications makes a request to the application server, it handles and accesses the resource server with the same tokens and returns it back to the application server, which handles the data as the application logic requires and returns it back to the client.

Refreshing the access token periodically is also part of the OAuth protocol. This should happen whenever a predefined timestamp for the access token expires or it is used a predefined number of times. When this happens, the application should use its refresh token to generate either a new set of tokens or just a new access token on the authentication server.

The Wiride authentication protocol has two little tweaks:

- Refreshing the tokens is not implemented. The only way to refresh the set of tokens is request a logout and log back in.
- The implementation of the resource, application and authentication server are all the same machine meaning that some requests between servers are collapsed and dealt within the same server.

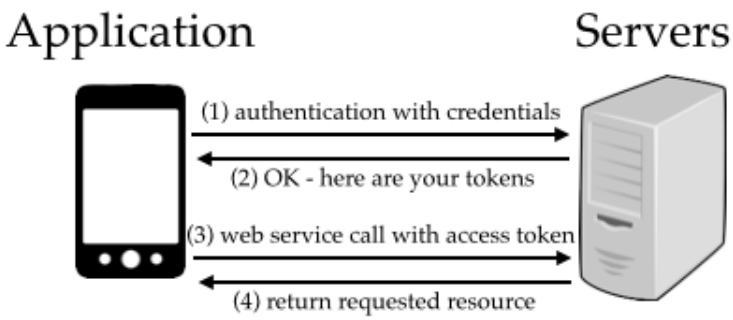

Figure 5.22: Wiride implementation of the open authentication protocol

5.5 Messaging

Users can communicate with each other within the application itself. To protect user's privacy they cannot choose who to message, instead, they can only message other users that have common lifts, we call this to be in the same circle. Both the server and the application will automatically add users to each other's circle whenever a certain event or action that make sense for these two users to communicate occurs.

New messages updates are received in one of two ways:

- **Request for new messages -** Whenever the user enters the messaging interface, the application will request all the messages from the server.
- **Updating service -** As we pointed previously, a new message triggers a new notification to be sent to the application whenever it polls for new updates.

In a first iteration of the messaging system, message updates were only available through the request of all the messages. Once all messages were fetched from the server, the application

would compare which ones were new and which ones were not and alert the user accordingly. It is easy to see that after a few messages the network overhead and the message processing was too big because the server was returning all the messages at all times.

The workaround to this problem was to whenever the application requests the messages, it provides a parameter 'last_id' that corresponds to the last message id that the application holds on Its database. This way, the server knows what messages the user holds and only returns messages that have a higher id than the provided one. This works because the message model has an unique incremental id.

With this change, the application can trust all the incoming messages are new ones, which makes the processing lighter. Also, requesting for new messages does not return all the messages, only new ones, which is good but most of the times the number of messages returned is zero and this reduces a lot data transfer. Additionally, with this simple feature it was possible to add the message updates to the updating service, which was impractical before since the service asks for updates every twenty seconds making the overhead too big to be viable.

5.6 Alerts

Every twenty seconds the NotifyService will poll the server for new notifications. These notifications will make changes on the database and therefore on the interface and they can sometimes trigger an alert. These alerts are created through the Android API for notifications, they are displayed in the top left corner of the device and they show small details about the lift if they are expanded.

Assuming an user A has offered a lift and user B is a possible passenger, there will be an alert each time:

1. User A accepts user B - B gets a notification;
2. User A denies user B - B gets a notification;
3. User B requests a seat in A's car - A gets a notification;
4. User B leaves A's lift - B gets a notification;
5. User A cancels his lift - Version of A leaving his own lift, passengers of A's lift get a notification, in this case, B;
6. User A kicks B - B gets a notification;

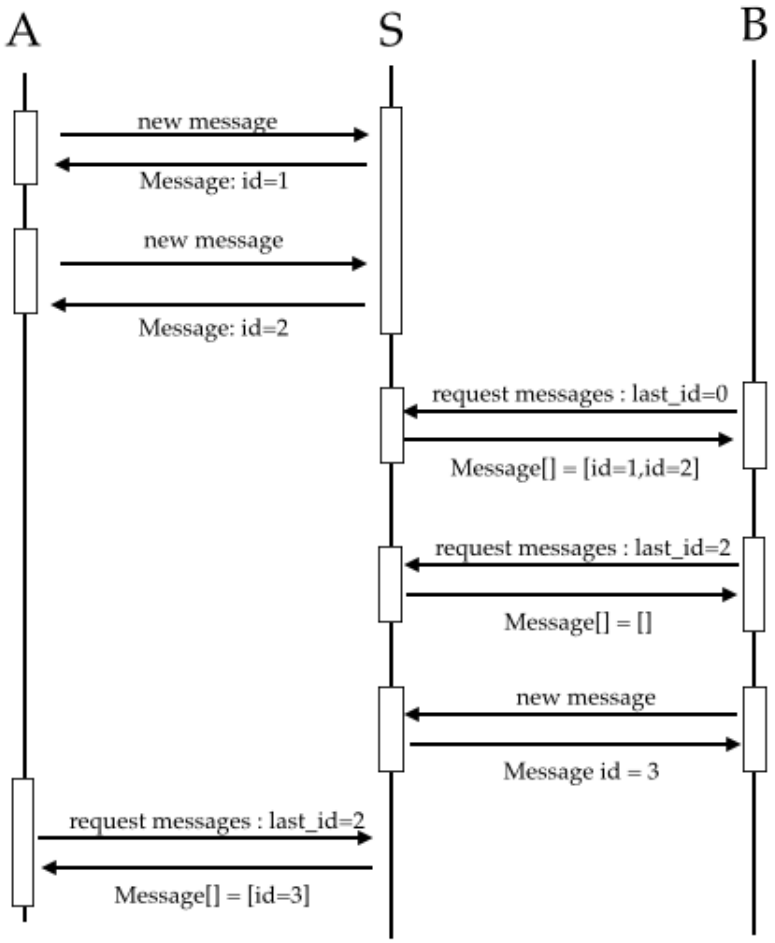

Figure 5.23: Message exchange and synchronization example

7. User A messages B - A gets a notification.

Every notification defined in Figure 5.20 triggers an alert except for notification type 7 'A_FINISHES', which is triggered when a lift finishes. This notification will only affect the application database, marking the recently finished lift as history.

5.7 Smart Locations

Google provides a public API to translate some input text into geographic coordinates.

```
private String GEOendpoint = "http://maps.googleapis.com/maps/api/geocode/json";
```

Figure 5.24: Google provided URL to make geocoding/reverse-geocoding requests

With the base URL, we are free to add some parameters to make the request. The parameters can either be an address or the latitude and longitude but the results will be almost the same since both components are included in the output of the request. Among other things, the main elements used in the applications are the 'formatted_address' parameter, which is the base address of the specified location with all the components and the geometry, which are the geographic coordinates for the specified address. One mandatory parameter to be added to the request is the 'sensor', which indicates whether the request was made by a device with a location sensor or not, this is just so Google can better manage and improve their information and requests.


```
{
  "results": [
    {
      "address_components": [ … ], // 8 items
      "formatted_address": "Instituto Superior Técnico, Avenida Rovisco Pais 1, 1049-001 Lisbon,
      "geometry": {
        "location": {
          "lat": 38.7368192,
          "lng": -9.138705
        },
        "location_type": "APPROXIMATE",
        "viewport": { … } // 2 items
      },
      "types": [ … ] // 3 items
    }
  ],
  "status": "OK"
}
```


Figure 5.25: Geocoding example


```
{
  "results": [
    {
      "address_components": [ … ], // 7 items
      "formatted_address": "Avenida Rovisco Pais 1, 1000 Lisbon, Portugal",
      "geometry": {
        "location": {
          "lat": 38.736762,
          "lng": -9.139358999999999
        },
        "location_type": "ROOFTOP",
        "viewport": { … } // 2 items
      },
      "types": [ … ] // 1 item
```


Figure 5.26: Reverse-geocoding example

One optional parameter that was added to narrow and obtain better results was the region. Since Wiride was being developed to initially work with Portuguese students the 'components=country:pt' parameter was added to show results only in Portugal since sometimes we were observing that

some addresses were often resulting in areas outside Portugal and that was not desirable at this point. One workaround for the future or to work internationally is to include this same parameter but instead of hardcoding the country, reading it from the smartphone's location. The output has two basic elements, the results and a status. The results are what was explained above. The status is metadata about the request and it is the first component the application reads in the output, there are five status codes but the application only reads two, the other ones are already considered before making the request, for example, making sure the address input box is properly filled. The two codes are the following:

- 'OK' - it indicates that there were no errors, at least one geocode was returned back the application.
- 'ZERO_RESULTS' - also indicates that there were no errors but also no results were found, that can either be because there is no geocode for that place (may happen in remote places) or the user has input a location with a typo.

When reading an 'OK' status, the application parses the output as an array of results, when reading an 'ZERO_RESULTS' status the application would indicate that no results were found and to proceed with another search or check if the input was correct. At this point, the results are inserted in the 'TABLE_LIFTS' and displayed in the screen as a list where the user can scroll for details.

5.8 Lifts

Looking at the architecture of the lifts shown in the previous chapter the implementation is pretty straightforward. When a user creates a lift, it is immediately available on the server to be searched and request a seat.

When an user searches for lifts, the application uses the coordinates to calculate the proximity of the available lifts, this is why every lift that is created needs to have this pair of coordinates, so it is possible to compare the search requests in terms of proximity. The formula in Figure 5.27 is used to calculate this proximity[1]. The calculated distance is the result of a straight line between those two points so naturally the travelled distance between those two points will be slightly bigger.

Every time a search request is sent, the server iterates over all the lifts on the database and calculates the distance between the two to see if it a candidate to be returned. Obviously, when

[1] http://www.movable-type.co.uk/scripts/latlong.html

```
$dLat = deg2rad($latB - $latA);
$dLon = deg2rad($lonB - $lonA);

$latA = deg2rad($latA);
$latB = deg2rad($latB);

$a = sin($dLat/2) * sin($dLat/2) +
     sin($dLon/2) * sin($dLon/2) * cos($latA) * cos($latB);

$c = 2 * atan2(sqrt($a), sqrt(1-$a));

$d = $R * $c;
```

Figure 5.27: Distance calculation through geographic coordinates

scaling this is a real problem, for every search made, the server will iterate for every new lift that it is created, this is a $O(n^2)$ complexity problem so this search needs to be optimized in the future.

Lifts do not need to be created only with addresses. The UI provides a button, when the user is prompted to input an address, which they can toggle to use their current location. When this happens, instead of geocoding the input address, the application will query the device for his current geographic coordinates and a reverse-geocoding request is made instead filling the address with the result from the request. Also, when a geocoding request is made, the output may be a result or a set of results, if more than one addressed is returned the user is prompted with a selectable list of the different options they need to tap to conclude the process of registering a new lift.

5.9 Interface Design

Generally there are different types of design, in Android, there are different ways to implement such designs, one type may be heavy, other may be lighter. Since this application is being developed to run on any Android device and because there is so much fragmentation in Android in terms of phone specifications and screen sizes [7], the application must run smoothly in any of these devices and the implemented design must fit in all available screen sizes.

In the first design iteration, the concern was to fit all the functionality into the application maintaining the user experience, without concerning if the application design was pretty. Here are some examples of the design and implemented user interface.

Taking a look at all screenshots we can see the type of elements, boxes, shades, gradients, custom buttons, custom elements like the action bar at the top, the checkbox and the date selector. All these custom elements are heavy to the application because naturally they have to be loaded and kept in memory. At the far left we have the main screen with the two main options of the

Figure 5.28: Main screen old design

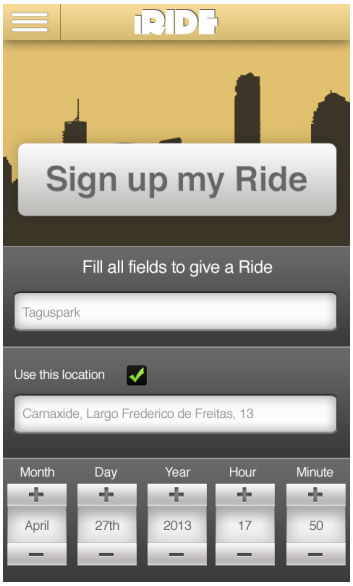

Figure 5.29: Give lift screen old design

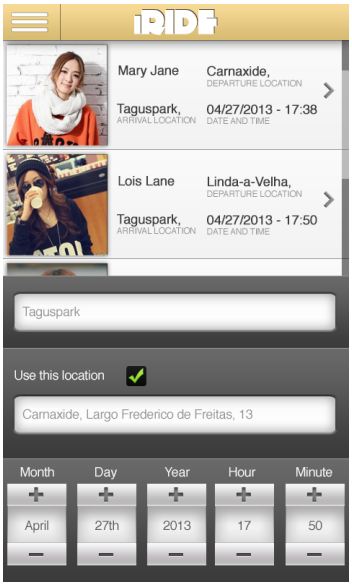

Figure 5.30: Search lift old design

application, offer and search a lift. When selecting the first option 'give a ride' the user will be redirected to the second screen where he inputs the arrival and departure location and the date. After the details have been input the user taps the 'Sign up my Ride' button and the application will do the geocoding like shown above. There is this option for 'Use this location' that will automatically check the user's location through the GPS system in the smartphone so he does not have to input it if they are already at the specified location. On the other hand, if the user selects the 'take a ride' option he will be redirected to a similar screen where he also needs to input an arrival and departure location as well as the date. After they submit their search request they either gets no results (third screen) or a set of results is displayed (fourth screen) where they can select to further request a seat after the user has checked the lift details.

In this last figure we have the same screens but with a different design. This design is more modern, much more light, prettier and the experience is either the same or better. What happened here is that the design moved to a flat approach, relying more on colors and flat shapes, gradients and shades were removed, the presence of more colorful buttons and pictures makes the use of the application a much more enjoyable experience, which is important because, like every application, one of its goals is to make users want to return to the application and that is achievable through good experiences.

Not only the design has changed but also how the components are present in the interface. In Figures 5.34 and 5.35 we can see the exact same profile screen for the same person but with different elements distribution. We decided to make use of the smartphones features to have

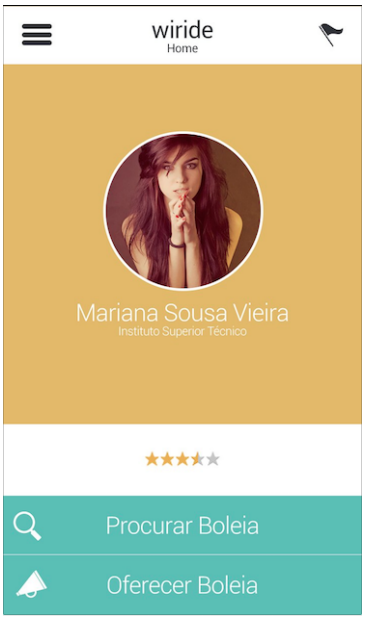

Figure 5.31: Main screen new design

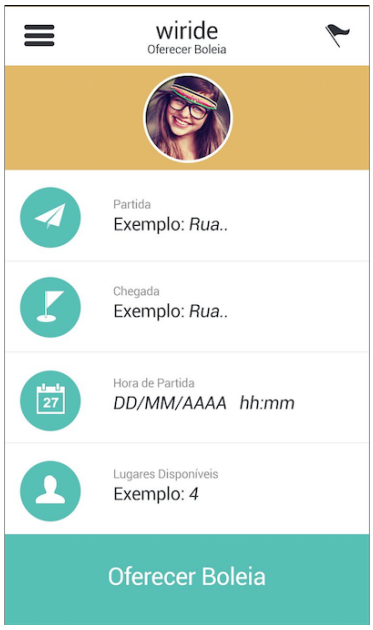

Figure 5.32: Give lift screen new design

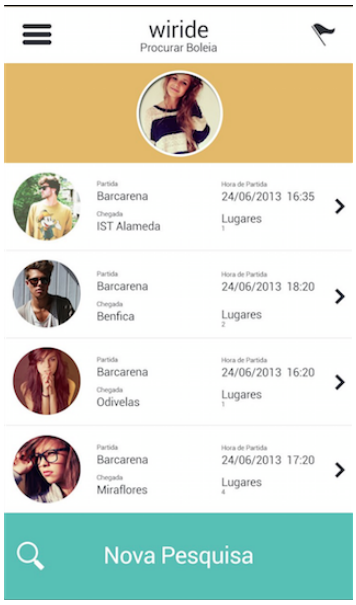

Figure 5.33: Search lift new design

more elements on the interface without making more condensed interface. In this example, stats of how many lifts the user has taken and given, as well as reputation and rating were added to the profile screen. Also, the whole screen is scrollable, this results in a less confusing user interface and it is also easier to read since there is clearly less verbose and the font size is slightly bigger. Specifically in this screen the most important elements were put on top and the less important, or the ones that will be less consulted were put at the bottom.

Making the screen scrollable is the feature that the smartphone has available to allow to make more space to the interface, this is rather important, specially on the Android operating system because there are so many different screen sizes and densities that it is impossible to predict on how the interface will exactly look on all available devices, the interface must be responsive in that way.

Figure 5.34: Old profile screen design

Figure 5.35: New profile screen design

Chapter 6

Evaluation

As explained before, the application took some months of development. Testing was always being done in house, by ourselves, there was no real testing from a user's perspective or any other person besides the Wiride team itself.

The first real test was at the summer festival Paredes de Coura[1], where it was possible to test all components of the system because the target were real users with real needs using the application.

The festival was promoted by Vodafone and as such, the application was a little rebranded in terms of colors to match Vodafone's colors. Also, the original version of the application had different needs compared to what was really needed in a festival, to have the application ready for the festival, the following modifications were made:

- Include Vodafone's colors in the application, red for the most part. Since Vodafone was to promote this application on their social networks, Wiride had to make a few changes to the design to match Vodafone's products, to have Vodafone branding.
- Simplify the login system, through facebook only. In the spirit of the festival and advised by responsible person of the mobile applications department from Vodafone, users want a fast experience a fast reward. Most of the users already use facebook as a way to create accounts on different applications, since this was a standalone application for the festival only, there was no need to have a complex login system to create accounts and users having to memorize their passwords for a one time event when they could do it through their facebook accounts which, lets face it, rare are the festival goers that do not have a facebook account. Figure 6.36 shows the difference between the relatively complex login

[1] http://www.paredesdecoura.com/

system and the simplified version for the festival.

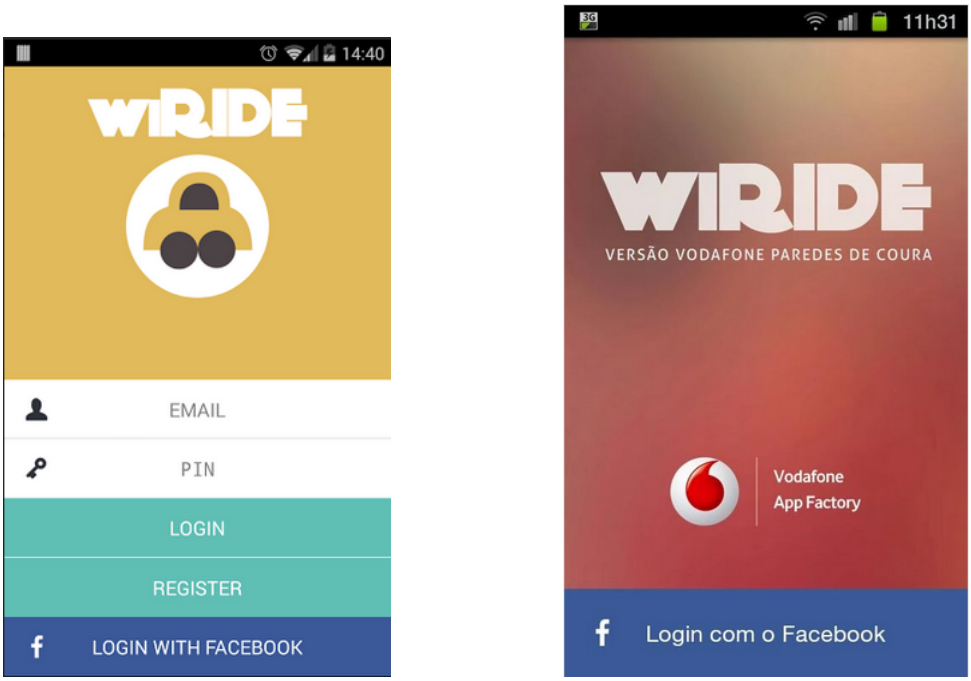

Figure 6.36: Login comparison between the original and the test version

- Reduce the amount of information available in the profile. Initially the application was designed to have information about the user, car information like the year, car displacement, fuel and type of car. Again, this was too much information to be filling for a one-time event, this information was discarded and the interface was simplified as a consequence.
- Change the experience when creating a lift and showing the active lifts. While testing with users and other people outside the team we noticed that when creating a lift the interface was too confusing, too much verbose, no clear idea where to tap to edit details and successfully create a new lift. The interface was slightly changed to outline buttons where the user can tap to edit each of the lift's details. Additionally in the original version the active lifts screen was a list with both types of active lifts, being either a driver or a passenger, with no clear distinction of which one is which. The approach to the festival version was to split them in a tab system. These two changes were not directly involved with the festival itself but rather with the usability tests made to users outside the team prior to the festival. Since these specific changes were a improve to the original experience, they were later merged back to the original version.
- Add a messaging system. By the time we started developing the application for the festival, the original version did not have the messaging system implemented yet. Although initially

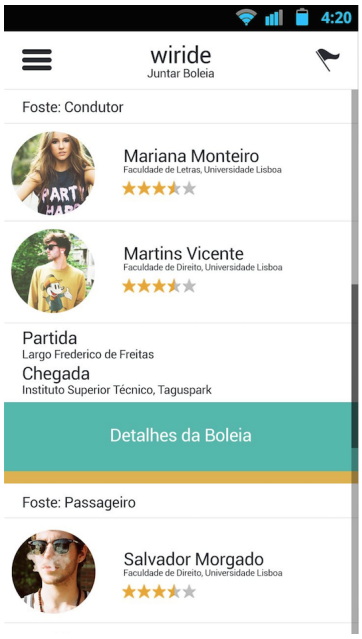

Figure 6.37: Original active lifts screen

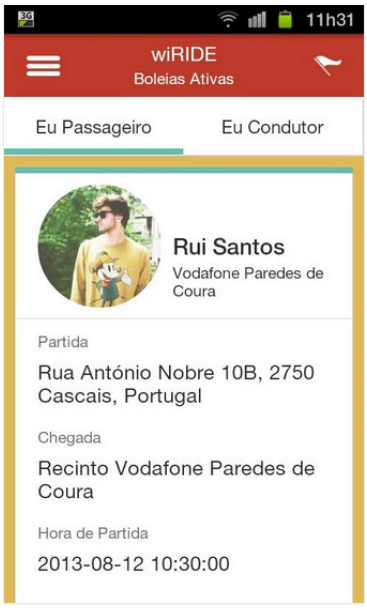

Figure 6.38: Improved active lifts screen

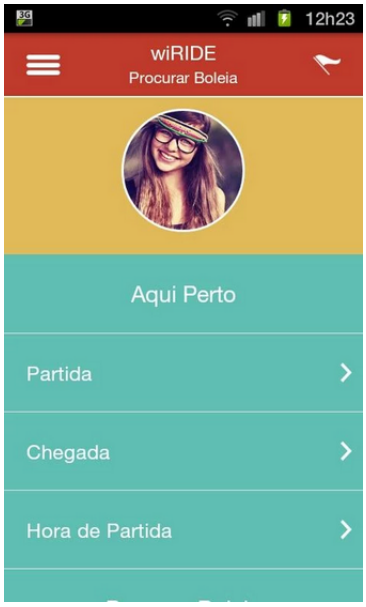

Figure 6.39: Improved search lift screen

the idea was to include the cellphone number in the profile details, that was not viable and so there was no way of communicating with the driver/passengers, the in-app messaging system was crucial in any of the versions.

- Predefined locations. One feature that was added was a predefined set of places in Paredes de Coura, instead of inserting an address in the arrival/departure locations, this version of the application offered the possibility to simply select one of predefined locations that were interest points located or near the festival venue. This feature was used on all lifts, they either started or finished in one of the selected locations. Also when this happened there was no need to geocode the input address since they were already present in the application.

All these features were forked from the original application, a server was replicated and the client side application was replicated as well. After the application was finished, some features were merged back into the original version, such as the messaging system and some changes in the general experience like detailed above. Figure 6.37 shows the original active lifts screen, we can see no clear distinction between the two types of lifts, being a driver or a passenger. On the other hand, figure 6.38 shows a clear distinction achieved through tabs between both types.

Considering the application was ready just days before the festival, which is not enough to get attention from users and Vodafone made one post on three different social networks, we do

not think the application had the proper attention to launch and run a proper pilot test with a satisfactory set of users. Nonetheless it was a good test to see the system working as a whole and it was more focused on the technological point of view, whether the application was crashing or if the users were having some sort of technical issues when using the application.

Table 6.5: Metrics obtained in the pilot test

Number of users	93
Number of lifts created	19
Number of users that actually joined a lift	7
Number of messages exchanged	39
Number of search lift requests	277

The statistics presented above are for the period from 8 of August to 22 of the same month. The application was announced and available in the Google Play store (market for android applications) on the 8th and the festival took place between 13th-17th. It is relevant to show usage before and after the event because in festivals, it is common to arrive some days early and/or leave some days late. Of all the 19 created lifts, 1 was departing from Oporto in the north, 1 from Algarve in the south, 16 from Lisbon in the centre region and 1 was actually departing from Paredes de Coura to Lisbon dated to the day after the festival's last day meaning it was someone with the interest of getting a lift to get back home rather than go to the festival.

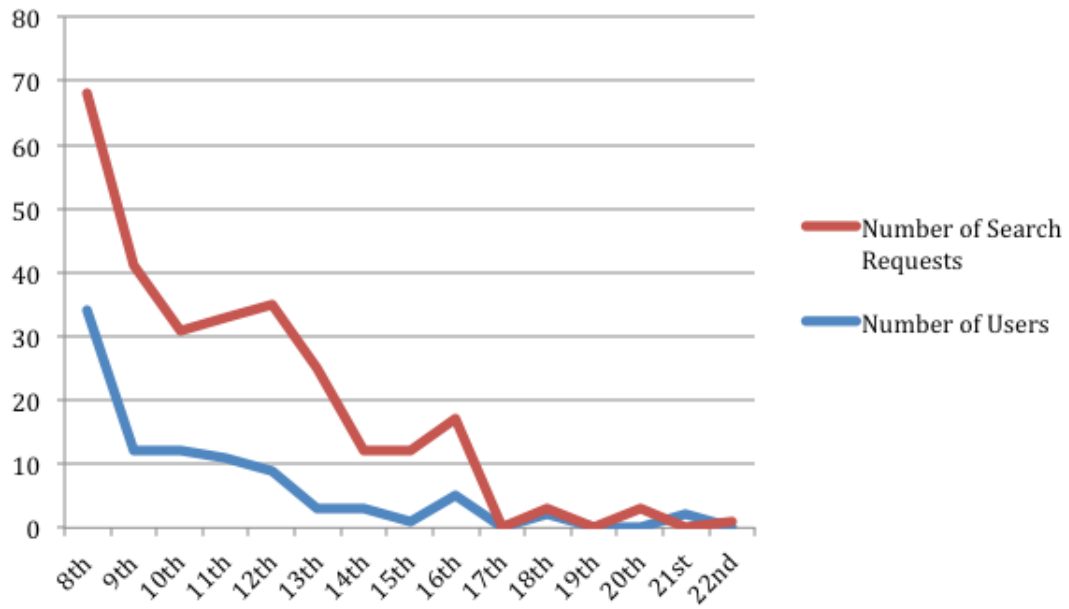

Figure 6.40: Number of search requests and users during the pilot test

It was not possible to test the server's throughput and capacity of request handling but we got far more important feedback from this experience. The kind of constructive feedback we got from this festival in terms of user experience is far more valuable and harder to obtain than the information that a heavy usage of the application could provide, simply because an app that is unappealing to the eye or it is too completed to use [8] may be a killer for users to use the application again and bad user experience was definitely something we did not want to merge back to the original version, that is also the purpose of making a pilot. Optimally we wanted the best of both worlds and get feedback in terms of efficiency and throughput as well as user experience.

From a business point of view it was also a positive experience, considering most of festival goers already have the details of their trip arranged days or even weeks before going to the festival and considering there was only a few days before the application was ready and promoted in the social networks, there was a reasonable amount of users downloading the application as well as making search requests from some parts of Portugal to Paredes de Coura, place where the festival was held.

Chapter 7

Conclusions and Future work

By the time the application was finished we can look back and tell how much we have learned from this project. So many things that should have been done differently in terms of implementation, so many additional features that an application like this has to have to be competitive. This being a software thesis in this chapter we will present and discuss those must have additional features in this application as well as alternate ways to implement some of the things that already have been done. These alternate ways are not just another way of writing software, they really are the way to do it in order to keep the application stable, scalable and maintainable by either the creators or any other programmers. Naturally when the application was first being designed there were so many concerns that were not considered and by the time the pilot test happened and after all that time developing it makes sense and not only should have been done this way since the beginning but also we would do it this way if we had to do it all over.

7.1 Additional features

This section will present the must have additional features to make a competitive service, the similar services described in Chapter 2 already have these features.

- **Real time lift tracking -** This feature aims to provide users, a map containing the driver and passengers locations for the lift that is taking place. Once the lift starts, anyone belonging to that lift will have access to a map showing everyones locations, this is important so users can easily identify their targets as well as identifying the exact locations of the pickup and be aware of eventual delays. It is the typical feature that it is not an absolute must have but increases the user experience immensely.

- **Rating -** Although it was initially planned to implement, a rating system is a plus because it helps both drivers and passengers to make decisions on what seats to requests and/or what passengers to accept. Since the proposed system is not a professional service like Uber, it is expected to have some good experiences and some not so good experiences. These experiences will result in different ratings for different drivers and passengers. Some of the provided screenshots even have examples of how the provided rating will be placed in the application interface. This feature was not implemented due to lack of time and it should be one of the next steps.

- **In-app payments -** The service already works as a carpooling system 'as is' but from a business point of view the application must include payments through the application in order to generate revenue. At this point, the only way to share the costs of carpooling is for drivers and passengers to reach out to each other and agree on the price and make the transaction themselves. With this feature implemented the transaction is made automatically, passengers have to load their virtual wallet and then the value of the lift will be calculated in one of two ways:

  1. Automatically by the system - The system selects some metrics like distance traveled, car type, gas price and number of passengers and through a predefined formula, calculates the estimated price for the lift. Passengers are then required to have that amount on their wallet before they can request a seat.
  2. Manually by the driver - The driver inputs what seems fair to charge for the lift. Naturally it is possible to assume that this type of price selection may trigger abuse of the system and make overpriced lifts. This is something that was not given a lot of thought but one option is to put a threshold for the price of a lift based on the metrics mentioned above.

- **Ahead searches -** Most of the times users cannot find any results for a given initial and final locations and users will either keep trying to search in different times or try different locations. One feature that can be implemented is to notify the user whenever the search that yielded no results before is available now, this way the user would not miss a lift because his search was off time.

7.2 Good software writing practices/Improvements

This section will show alternate ways to implement some of the written code, some things that were not considered but when implementing we crossed a few issues that are not a problem at

early stages but at some point, if the application gets a lot of users and usage, it compromises the overall experience.

- **Service vs Push -** As shown before in Chapter 5, Wiride polls an endpoint for updates. Every 20 seconds makes an HTTP request to check for any new updates. As explained before this can be very expensive in terms of resources, battery and data charges may apply. We could optimize the polling part with some improved polling method but there is one better way to synchronize data between server and application. The way to do such synchronization is through a push service, like the name suggests push services work by pushing updates, in the form of messages, to the client, the opposite of polling which is the client asking for updates. This way as soon as an update is available, the server pushes that update to the client. Google provides a push service for all the applications for it is operating system. The way the service is currently implemented drains a lot of battery and if many applications worked like this android users would constantly have a bad experience and would be reticent for using android operating devices. This is why Google provides the Google Cloud Messaging (GCM)[1] which is the messaging system for synchronization between client and server and it works in a few small simple steps:

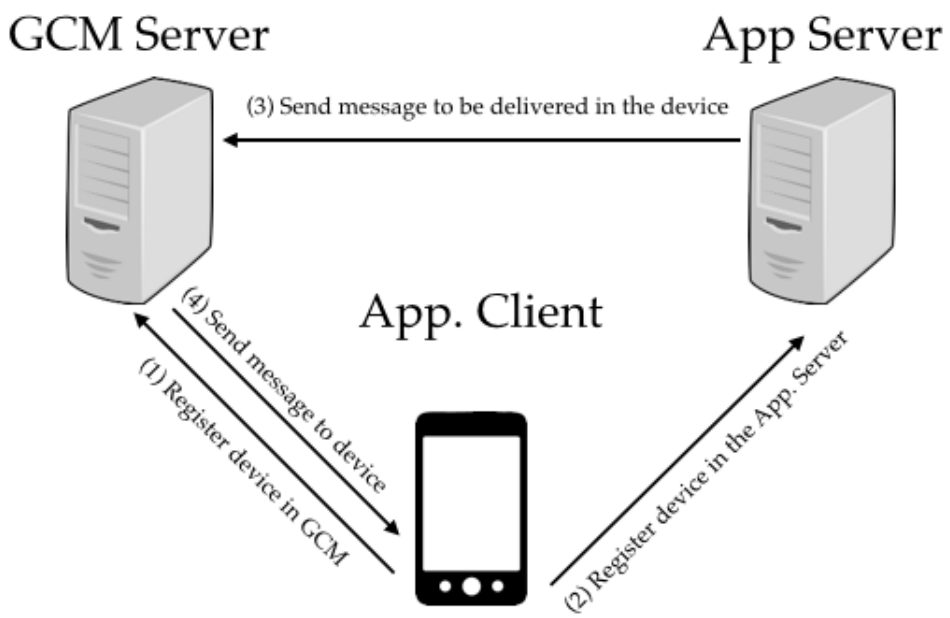

Figure 7.41: Push service registration and usage

1. Registration

[1] https://developer.android.com/google/gcm/index.html

  - The device running the application client registers itself in the GCM server.
  - The application client sends the device id to the applications server.

2. Message sending
  - The application server sends the GCM server the message and the device id of the device it should be delivered.
  - Since the application registered itself, the GCM server can forward the message to the device.
  - The application receives the message and deals with it properly, it either requests a synchronization to the server or the message itself has all the information needed to update the state.

- **Date representation -** Dates should be represented as longs rather than strings because strings maintain a format that may not be the same from platform to platform, language to language. The long data type should be used instead and each platform should transform into the desired string format.

- **Unique id generation -** The implemented resource ids are currently unique incremental integers, this type of approach is not at all scalable or maintainable, here is why:

  1. Ids are capped at a maximum of 2^{31}, it may not be a problem to the user resource but it may be a problem for the message resource for example. In current days, a 32 bit number is not enough to uniquely identify a resource, a 64 bit should be used instead.
  2. If we have multiple servers handling requests, we have to assure that different machines do not create the same ids, that they always generate different ids even when that happens at the exact same time, this is a common problem in distributed systems.

  One possible solution to deal with these issues is to do like the photo service Instagram does [9] and allocate bits to specific components. Taking that we use a 64 long id we split it into three segments:

  - 41 bits for time in milliseconds - that gives about 41 years of ids;
  - 13 bits machine id - that gives 8.192 possible different machines generating ids;
  - 10 bits sequence number - that allows to generate 1024 extra ids.

  This way it is possible to generate thousands of ids per second for many years in many different machines and still guarantee they are unique. More precisely, 1024 unique ids across 8192 different machines per millisecond for as long as 41 years.

Table 7.6: REST API for the message resource

Messages			
Endpoint	**HTTP Method**	**Parameters**	**Return**
"/circles" get the circle users	GET		User[]
"/messages" request all the messages	GET		Message[]
"/messages" creates a new message	POST	message:String to_user_id:long	Message

- **REST API -** The endpoints shown above are somewhat misleading, as it was stated before, it is not a true REST implementation. Let us recall Table 5.4 and redo it.

  The endpoints to access the resource should always be the same, the server should identify what operations is to be executed depending on the HTTP type of request. Looking at table 7.6, POST should create a new resource while GET should return all the messages, there is no point in having more complex URLs to do the basic CRUD operations on resources [10].

7.3 Final conclusions

It is interesting to see that the pilot taken had some adhesion and from both business and technical perspective it may be feasible to develop or to fork this type of application for specific events, to promote carpooling for specific events. In this case it was the 'Wiride for Vodafone Paredes de Coura' but easily enough the application can change to another type of event which is definitely something to think about.

It is part of the developing process to get feedback from users and accommodate changes as the time goes by. Most of the features and optimizations described in this chapter can certainly be merged into the existing application without compromising the user experience, on the contrary, users would see a significantly better experience and as a consequence use it more times.

Building the application from the ground up was a great experience, one thing that we learned and after showing some future work to be done is that, in this type of software/product it is very hard to reach a final state so it is very important to deliver and get a first raw version fast and optimize and develop side features later. Some decisions were even based on this premisse, having in mind that some choices are for the short-medium term rather than the long term just

because it is faster to deliver that way.

Bibliography

[1] Jim Edwards. Uber has changed my life and i will never take a taxi again. *Business Insider*, January 2014.

[2] Jianling Li; Patrick Embry; Stephen P. Mattingly; Kaveh Farokhi Sadabadi; Isaradatta Rasmidatta; and Mark W. Burris. Who chooses to carpool and why? *Journal of the Transportation Research Board*, February 2007.

[3] Todd Litman. Evaluating carsharing benefits. *Journal of the Transportation Research Board*, December 1999.

[4] Roy T. Fielding. *Representational State Transfer (REST)*. PhD thesis, University of California, Irvine, 2000.

[5] Roy T. Fielding; Richard N. Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology*, May 2002.

[6] Margaret Rouse. Lamp (linux, apache, mysql, php).

[7] Steve Kovach. Android's massive fragmentation problem in two handy charts. *Business Insider*, July 2013.

[8] Rob Girellini. Top reasons why most mobile apps fail. *NYCWebDesign*, February 2013.

[9] Mike Krieger. Sharding and ids at instagram. *Instagram Engineering*, November 2011.

[10] Martin Heller. Rest and crud: the impedance mismatch. *InfoWorld - IDG News Service*, January 2007.