Wiride - A mobile application for a carpooling system

Rui Santos*
Instituto Superior Técnico, Lisbon, Portugal
rui.santos@tecnico.ulisboa.pt

Abstract.

People in general sometimes have a hard time conciliating their schedules because of the way they move from one location to another. 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. The platform consists of an application server and a mobile application that runs on people’s smartphones, this way it is accessible everywhere. 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, mobile application, carpooling service, Android

1 Introduction

Many people every day have difficulties reaching their destination, many areas, specially rural areas, have poor transportation networks, frequency of public transports is very low and restricted to certain times of the day. The available solutions, again, especially in rural areas are a very expensive option and to the regular person, unaffordable for the every day use. Wiride is a carpooling mobile application aimed to help people find drivers and passengers sharing the same routes. The idea of developing such application came from a group of students with real difficulties in getting to their university everyday. The difficulty is managing the schedules, students in particular have different school schedules from one day to another and the transportation schedules are always the same, this means that sometimes these students have to catch a different bus or other mean of transportation everyday that may not be compatible at all with their schedule resulting in a very inefficient time schedule. The workaround to this issue is most of the times very expensive and this is where Wiride tries to intervene.

Wiride is 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. By carpooling, drivers and passengers are reducing the costs and passengers are reducing the time spent on the available transportation networks.

This work shows the architecture, the implementation as well as the results taken from the pilot test as part of the process of developing the carpooling mobile application for the Android operating system.

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 and services. 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.

2 Related Work

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. Carpooling brings many advantages, some of which:

- Reduce traffic congestion, particularly in rush hour;
- Avoid parking problems, less vehicles overall means more free parking spots;
- Energy consumption and pollutions reduction;
- Reduce travel costs;
- Alternative transportation mode to areas with low transportation services.

Carpooling also has some challenges that carpooling service providers must address to have more people to join:

- **Carpooling with strangers** increases the concerns over security and has been an obstacle sharing a vehicle with strangers. One way to address this issue is to introduce the system in small niches, people are more likely to carpool with other people from the same environment.
- **Flexibility** - it can be hard to accommodate en route stops or change what already is a very rigid pattern from a driver’s perspective. While it isn’t a barrier when long-distance travelling, the sorter the travel distance is, the less likely drivers are up to detour their own route.
- **Availability** - Carpooling needs a lot of people to get the system working. Chances of getting a match for a certain (or similar) course are extremely low and need a considerable amount of active users.

- **Legal constraints** may apply because ride sharing lets anyone be a taxi driver and these types of services 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. Carsharing also brings many of the advantages from carpooling but not all the challenges.

Currently there are a lot of carpooling/carsharing solutions being developed and deployed, some by major established companies as a branch service, others being developed by startups, being their primary service. Uber\(^1\) and Lyft\(^2\) are both startups, both carpooling services and leaders in this service in the areas they operate in. Uber is a premium service with professional drivers that exclusively drive black cars while Lyft has more of a community atmosphere where drivers are ordinary people with their daily jobs and routines.

3 Functional Architecture

The system follows a centric architecture like shown in Figure 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 main implemented features are presented next.

3.1 Lifts

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.

\(^{1}\) https://www.uber.com/
\(^{2}\) https://www.lyft.com/
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 2 shows the possible process of creating a lift and request seats for a lift with two users, a driver and a passenger.

3.2 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 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.
3.3 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 names for the same location. For example, ‘Instituto Superior Tecnico’ 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.

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

4.2 Server

As for the server side, it was built with a simple LAMP [4] 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.

5 Implementation

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. 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.

Below we will present the method used to develop this application as well as how the main features were implemented.

5.1 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.

- **Lift** - Represents every lift that is created throughout the application.

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

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 as database because the database was instantiated with this same context as a parameter.

5.2 Defining the communication protocols

Looking at the architecture in the previous chapter it is clear the kind of protocols the applications follows. The application is a web-based application in the way that the server is available through 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 4 of the HTTP codes to help with communications between clients and server.

- **200 OK** - The request has been fulfilled.

- **201 Created** - The request has been fulfilled and resulted in a new resource being created.

- **400 Bad Request** - This code is used for when the server due to malformed syntax could not understand the request.

- **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.

5.3 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.

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.

### 5.4 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'

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

5.5 Updating Service

The Wiride applications has a service that is crucial for the well behavior of the application. It is in this service that the application will fetch all the updates from the server and notify the user accordingly. 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 forcibly 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.

5.6 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 4 is used to calculate this proximity. 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 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.

5.7 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.

---


Fig. 4. Distance calculation through geographic coordinates
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.

5.8 Smart Locations

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

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.

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.

6 Evaluation

As mentioned earlier, Wiride built an application just for the summer festival ‘Paredes de Coura by Vodafone’. The prime goal of the pilot test was to test both technological and business side of the
carpooling service as well as promote the application itself. Since this was a partnership with Vodafone the application had to be rebranded to reflect Vodafone’s brand on the application. Additionally, some features had to be implemented and others had to be redesigned to have a better user experience, these changes were also part of the test since we had to submit the application to usability tests. Here are the changes that were made to the forked version to the festival:

- Included Vodafone’s brand in the application’s design;
- Simplified the login system, use login through Facebook only;
- Reduced the amount of information available in the users’ profile. No need that much information for one event only.
- Improved user experience in the active lifts screen and when creating a new lift.
- Added a messaging system between users that are in a lift together.
- Added some predefined locations to facilitate users inserting their locations.

All the features except for the simplified login system and the rebranding of the application were merged back into the original version.

The festival took place between 13th and 17th of August and the application was launched and promoted on the 8th. Below there are the metrics obtained between 8th and 22nd of August. This time frame was chosen because it is very common in summer festivals to arrive some days before the festival starts and some days after the festival is over.

<table>
<thead>
<tr>
<th>Table 1. Metrics obtained in the pilot test</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number of users</strong></td>
</tr>
<tr>
<td><strong>Number of lifts created</strong></td>
</tr>
<tr>
<td><strong>Number of users that actually joined a lift</strong></td>
</tr>
<tr>
<td><strong>Number of messages exchanged</strong></td>
</tr>
<tr>
<td><strong>Number of search lift requests</strong></td>
</tr>
</tbody>
</table>

In figure 5 the red line indicates the number of search requests during the period of the festival and the blue lines indicates the number of user registrations.

![Fig. 5. Number of search requests and users during the pilot test](image)

7 Conclusion

By the time the application was finished there are so many things that should have been done differently in terms of implementation and so many additional features that are needed to be competitive in this service. We will now present and discuss some of the additional features that need to be implemented as well as some alternate ways to implement what is already done considering that the goal is to keep the application stable, scalable and maintainable, which is not at this point.
7.1 Additional features

The services described in Chapter 2 already have these features, except for the last one, let us take a look:

- **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. Users can identify the exact locations of the driver and passengers and be aware of any delays.

- **Rating** - Users should be able to rate others so they can later identify their experience carpooling with each other. Eventually bad behavior could be punished and good behavior rewarded.

- **In-app payments** - At the moment users do not really get their costs lowered by the use of the application, transactions are up to the users for doing it manually at some point in the lift. A system to do such transactions through the application is crucial to assure there is a fair split of costs and to protect drivers.

- **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. This feature is not present in the other services because those services are a search-on-demand rather than search to plan a trip.

7.2 Software improvement

Some implementation should have been done differently, some things were not considered in the beginning and were identified in the development process, these are the ones that can compromise the overall experience:

- **Service vs Push** - The applications at the moment polls the server for updates through a service, this can be very expensive in terms of battery life and traffic because data charges may apply. The way to synchronize data between client and server is through push notifications, which instead of asking the server for updates is wait for the server to tell the application of any new updates. Google provides a push notifications service for all its applications called Google Cloud Messaging and it should be used instead of the current polling service.

- **Data 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. Applications nowadays have multiple servers handling requests, they need to assure that different servers do create the same ids or overlap each other at some point, this is a common problem in distributed systems. The ids are currently using a 32 bit integer which at some point in time may not be sufficient to uniquely identify some resources like the message type. One possible solution to address this issue is to use 64 bit longs and allocate specific bits to identify different machines. This way there will be enough ids to identify the resource making sure each server has its own signature on the id, which might come in handy.

- **REST API** - The designed API does provide a true REST API experience. REST APIs should have less endpoints and rely more on the HTTP methods to identify which CRUD operations should be done on resources.

7.3 Final conclusions

References


---