

## BACHELOR PROJECT ASSIGNMENT

**Student:** David Moidl

**Study programme:** Open Informatics

**Specialisation:** Computer and Information Science

**Title of Bachelor Project:** Smartphone-based Real-time Taxi Sharing System

### Guidelines:

1. Survey existing methods and applications for real-time ride sharing.
2. Familiarize yourself with Android Platform and Google App Engine development.
3. Formalize taxi ride sharing as a multi-agent coordination problem.
4. Propose a ride matching algorithm and a coordination mechanism for shared taxi ride negotiation.
5. Implement both the client-side and server-side component of the matching algorithm and the coordination mechanism.
6. Evaluate the resulting taxi ride sharing system on a set of test scenarios.


**Bibliography/Sources:** Will be provided by the supervisor.

**Bachelor Project Supervisor:** Ing. Michal Jakob, Ph.D.

**Valid until:** the end of the winter semester of academic year 2012/2013

  
prof. Ing. Vladimír Mařík, DrSc.  
Head of Department



  
prof. Ing. Pavel Ripka, CSc.  
Dean

Prague, January 9, 2012

## ZADÁNÍ BAKALÁŘSKÉ PRÁCE

**Student:** David Moidl  
**Studijní program:** Otevřená informatika (bakalářský)  
**Obor:** Informatika a počítačové vědy  
**Název tématu:** Systém pro sdílení taxi v reálném čase pomocí chytrých telefonů


### Pokyny pro vypracování:

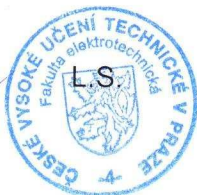
1. Prostudujte existující metody a aplikace pro spolujízdu v reálném čase.
2. Seznamte se s vývojem na platformě Android a Google App Engine.
3. Formalizujte problém taxi spolujízdy jako multiagentní koordinační problém.
4. Navrhněte algoritmus párování jízd a koordinační mechanismus pro vyjednávání taxi spolujízd.
5. Implementujte klientskou a serverovou komponentu párovacího algoritmu a koordinačního mechanismu.
6. Vyhodnoťte výsledný systém pro sdílení taxi na sadě testovacích scénářů.

**Seznam odborné literatury:** Dodá vedoucí práce.

**Vedoucí bakalářské práce:** Ing. Michal Jakob, Ph.D.

**Platnost zadání:** do konce zimního semestru 2012/2013

  
prof. Ing. Vladimír Mařík, DrSc.  
vedoucí katedry



  
prof. Ing. Pavel Ripka, CSc.  
děkan

V Praze dne 9. 1. 2012

Czech Technical University in Prague  
Faculty of Electrical Engineering  
Department of Cybernetics



Bachelor's thesis

## **Smartphone-based Real-time Taxi Sharing System**

*David Moidl*

Supervisor: Ing. Michal Jakob, Ph.D.

Study Programme: Open informatics

Field of Study: Informatics and computer science

May 20, 2012

## Acknowledgements

First of all I would like to thank Ing. Michal Jakob, Ph.D. for his professional leadership. I would also like to thank my family for their support and understanding and at last but not least my friends Petr Mezek and Adam Kratochvíl for consultations on various subjects.

## Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 20. 5. 2012



.....

# Abstract

Transportation is one the pillars of today's society. Substantial percentage of people transported all over the world every day is served by public transportation. One kind of public transportation service allowing people to travel from somewhere to somewhere else is taxi service. Despite the fact that systems for planning shared rides using single taxi vehicle do exist, in many countries (including Czech republic) it's still rather unfavourable to use taxis for transportation, all the more so with some of these systems having insufficient potential to be actually helpful.

I believe, that there is enough space for another system of such a kind based exclusively on "smartphones" which are so widely spread these days. A system allowing its users to quickly and easily plan a shared ride using single taxi vehicle. It would not only help its users to save their money, but it would also make the whole taxi transportation more efficient. Such a system - its architecture, implementation and overall workflow - is the topic of this work.

# Abstrakt

Doprava je jedním z pilířů dnešní společnosti. Veřejná doprava se pak významně podílí na celkovém počtu osob přepravovaných každý den po světě. Jednou z možností, jak se dostat z jednoho místa na druhé je využití taxislužby, kterých je dnes mnoho a svou činností pokrývají valnou část obydleného území. V mnoha zemích (Českou republiku nevyjímaje) je však přeprava pomocí taxislužby spíše nevýhodná. Ačkoli již existují systémy, které by měly pomoci lidem plánovat společné jízdy jedním vozidlem taxi, není jich mnoho a některé z nich ani nemají dostatečný potenciál, aby lidem skutečně pomohly.

Věřím, že na poli sdílených jízd taxi existuje dostatek prostoru pro nový systém založený výhradně na schopnostech dnes tak rozšířených "chytrých" telefonů, který by umožnil svým uživatelům rychle a jednoduše naplánovat společnou jízdu jediným vozidlem taxi, čímž by nejen přispěl k úspoře peněz jeho uživatelů, ale i ke zefektivnění využití taxislužeb. Takový systém - jeho návrh, implementace a fungování - je tématem této práce.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Result of my work . . . . .	2
1.3	Thesis outline . . . . .	3
<b>2</b>	<b>Related work</b>	<b>5</b>
2.1	The problem . . . . .	5
2.2	Existing solutions . . . . .	5
2.2.1	Cabcorner.com . . . . .	6
2.2.1.1	Description . . . . .	6
2.2.1.2	Interesting observations . . . . .	6
2.2.1.3	Improvement opportunities . . . . .	6
2.2.1.4	Summary . . . . .	7
2.2.2	RideAmigos.com . . . . .	7
2.2.2.1	Description . . . . .	7
2.2.2.2	Interesting observations . . . . .	7
2.2.2.3	Improvement opportunities . . . . .	7
2.2.2.4	Summary . . . . .	8
2.2.3	Taxi Share - Chicago . . . . .	8
2.2.3.1	Description . . . . .	8
2.2.3.2	Interesting observations . . . . .	8
2.2.3.3	Improvement opportunities . . . . .	8
2.2.3.4	Summary . . . . .	9
2.2.4	FareShare . . . . .	9
2.2.4.1	Description . . . . .	9
2.2.4.2	Interesting observations . . . . .	9
2.2.4.3	Improvement opportunities . . . . .	10
2.2.4.4	Summary . . . . .	11
2.3	Related work summary . . . . .	11
2.4	Surveys and external references . . . . .	11
2.4.1	Spatio-temporal clustering . . . . .	12
2.4.2	Dial a ride problems . . . . .	12

<b>3</b>	<b>Requirement analysis and specifications</b>	<b>13</b>
3.1	Requirement analysis . . . . .	13
3.1.1	Essential functions . . . . .	13
3.1.2	The additional features . . . . .	14
3.2	Specification . . . . .	15
3.2.1	Application inputs . . . . .	15
3.2.1.1	Authentication . . . . .	15
3.2.1.2	Ride specification . . . . .	15
3.2.2	Outputs of the application . . . . .	16
<b>4</b>	<b>Architecture and design</b>	<b>17</b>
4.1	Architecture . . . . .	17
4.1.1	Model . . . . .	17
4.1.1.1	Building blocks . . . . .	17
4.1.2	The workflow . . . . .	18
4.1.3	Technical point of view . . . . .	22
4.2	Design . . . . .	23
4.2.1	Android philosophy . . . . .	23
4.2.1.1	Android activities . . . . .	23
4.2.1.2	Screenshots . . . . .	23
<b>5</b>	<b>Algorithms</b>	<b>25</b>
5.1	Request matching . . . . .	25
5.1.1	K-Means algorithm . . . . .	25
5.1.2	Directional algorithm . . . . .	27
5.2	Prototype evaluation . . . . .	29
5.3	Ride plan creation . . . . .	33
5.4	Negotiation . . . . .	33
5.5	Data cleaning . . . . .	33
5.6	Detailed description . . . . .	34
5.6.1	The distance . . . . .	34
5.6.2	Etalon adjustment . . . . .	34
5.6.3	Etalon postprocessing . . . . .	35
5.6.4	Online processing . . . . .	35
5.6.5	Cross-bearing difference . . . . .	36
<b>6</b>	<b>Implementation</b>	<b>39</b>
6.1	Specific description . . . . .	39
6.2	Server-side specialities . . . . .	39
6.2.1	Single threading and responsiveness . . . . .	39
6.2.2	The datastore . . . . .	40
6.2.2.1	Description . . . . .	40
6.2.2.2	Limitations . . . . .	40
6.2.2.3	Usage . . . . .	40
6.2.2.4	Frameworks . . . . .	41
6.3	Client-side specialities . . . . .	41



6.3.1	XML based resources . . . . .	42
6.3.2	Painless threading . . . . .	42
6.3.3	Using maps . . . . .	43
6.3.4	System-user cooperation . . . . .	43
<b>7</b>	<b>Evaluation</b>	<b>45</b>
7.1	Experiments introduction . . . . .	45
7.1.1	Metrics . . . . .	45
7.2	Testing of the system . . . . .	46
7.2.1	Input parameters description . . . . .	46
7.3	Tests specification . . . . .	47
7.4	Tests of clustering algorithms . . . . .	48
7.4.1	[K-Means algorithm] Relations between maximal time difference, maximal spatial distance and number of created rides . . . . .	48
7.4.2	[K-Means algorithm] Relations between maximal time difference, maximal spatial distance and ride efficiency . . . . .	49
7.4.3	[K-Means algorithm] Relations between maximal time difference and average time deviation . . . . .	51
7.4.4	[Directional algorithm] Relations between maximal time difference, maximal cross-bearing difference and number of created rides . . . . .	51
7.4.5	[Directional algorithm] Relations between maximal time difference, maximal cross-bearing difference and effectiveness of the ride . . . . .	53
7.4.6	[Directional algorithm] Relations between maximal time difference and average time deviation . . . . .	55
7.5	Tests of online algorithm . . . . .	56
7.5.1	[Online algorithm] Relations between input parameters and number of created rides . . . . .	56
7.5.2	[Online algorithm] Relations between input parameters and efficiency of a ride . . . . .	57
7.6	Test results summary . . . . .	59
<b>8</b>	<b>Conclusion</b>	<b>61</b>
8.1	Recapitulation . . . . .	61
8.2	Work summary . . . . .	61

# List of Figures

4.1	Workflow of clustering algorithms . . . . .	19
4.2	Workflow of the online algorithm . . . . .	19
4.3	Workflow of server-client intercommunication for clustering algorithms . . . . .	20
4.4	Workflow of server-client intercommunication for online algorithm . . . . .	21
4.5	Screenshots of the client application . . . . .	24
5.1	Graphical explanation of cross-bearing difference . . . . .	37
7.1	Relations between maximal time difference, maximal spatial distance and number of created rides . . . . .	49
7.2	Relations between maximal time difference, maximal spatial distance and average detour . . . . .	50
7.3	Relations between maximal time difference, maximal spatial distance and average saving . . . . .	50
7.4	Relations between average time deviation and maximal allowed time difference . . . . .	51
7.5	Relation between maximal time difference, maximal cross-bearing difference and number of created rides . . . . .	52
7.6	Relations between maximal time difference, maximal cross-bearing difference and average saving of passengers . . . . .	53
7.7	Relations between maximal time difference, maximal cross-bearing difference and average detour . . . . .	54
7.8	Relations between average time deviation and maximal allowed time difference . . . . .	55
7.9	Relations between maximal time difference, maximal bearing difference and number of created rides . . . . .	57
7.10	Relations between maximal cross-bearing difference and efficiency of a ride . . . . .	58
7.11	Relations between maximal time difference and efficiency of a ride . . . . .	58
7.12	Comparison of speed of individual algorithms . . . . .	60

# List of Tables

7.1	Shortcuts for scenarios . . . . .	48
7.2	Configuration for <i>K-Means</i> test 1 . . . . .	48
7.3	Configuration for <i>K-Means</i> test 2 . . . . .	49
7.4	Configuration for <i>K-Means</i> test 3 . . . . .	51
7.5	Configuration for <i>Directional</i> test 1 . . . . .	52
7.6	Configuration for <i>Directional</i> test 2 . . . . .	53
7.7	Configuration for <i>Directional</i> test 2 . . . . .	55
7.8	Configuration for <i>Directional online</i> test 1 . . . . .	56
7.9	Configuration for <i>Directional online</i> test 2 . . . . .	57
7.10	Optimal values of parameters . . . . .	60
7.11	Comparison of average saving for individual algorithms . . . . .	60

# Chapter 1

## Introduction

### 1.1 Motivation

Taxi - an international word with the meaning of “a motor vehicle licensed to transport passengers in return for payment of a fare and typically fitted with taximeter.”<sup>1</sup> The fact, that this word is international somehow implies that companies running a taxi business are everywhere. And in most cases, they really are. Usually, taxi services are also easily reachable: one can find their advertisements on a public places and see their vehicles running across the city. No matter where one currently is, all one needs to do to get a ride is dial the right number or simply wave on a passing vehicle. Eventually, a taxi is always “just around the corner” ready to pick up it’s passenger and transfer him to desired destination.

But ubiquity of companies providing taxi services is the only thing they have in common. There are huge differences around the world when it comes to prices, number of cars and mainly their usage. On one side, there are localities like New York, Chicago, London and other major cities where the number of taxis on the streets is almost the same as the number of ordinary cars. These “taxi fleets” are used on a regular basis by a huge amount of passengers as an independent type of public transportation. On the other side, there are many countries where this schema does not apply. Consider taxi services in Czech republic: there are multiple companies in taxi business operating on almost entire area of the country, each with sufficient number of cars. But they are far from being used as much as the ones in cities listed above. Also the price of a ride is quite high which is a significant drawback preventing taxis from being used by the general public. But what if people had the opportunity to easily arrange a single ride for multiple passengers using only one vehicle? Would it result in higher usage of taxis? I believe the answer is yes.

Imagine a system enabling it’s users to do so. What would be the advantages of such a system? Well, the obvious one is lowering the price of a ride per passenger. In a ride shared between two people, the amount of money each of them has to pay would drop to a half of the original value. That’s definitely positive. But what if one would share a ride with two or even three more people...? The benefit is evident. Another advantage is a significant increase of efficiency. Simply put, the more people in one car, the more effective usage of

---

<sup>1</sup>Oxford online dictionary: <http://oxforddictionaries.com/definition/taxi>

that car. That would logically result in lower fuel consumption, thus the ratio of produced greenhouse gasses per passenger would drop making taxis more environment friendly. At least but not last there is the advantage of giving transportation represented by buses and trains. Unlike train or bus, taxi is highly flexible. It doesn't matter whether the passenger needs to travel from the airport to downtown or to another city. Taxi is an independent unit with no predefined route and no stopovers, so it's ride is straight and fast.

With all these advantages it would be great if such a system existed. But it's apparent that the type of a system would have to be quite specific. It would have to allow it's users to immediately share their decision about going somewhere by taxi. That excludes a computer application-based system right away. But it seems that a mobile-based system would do the job. Since lots of people own a "smartphone" these days, phones sophisticated enough provide a good platform for a system of this type. An important note is that a noticeable amount of today's smartphones is running operating system Google Android, which is the operating system for which the application will be developed. System could then utilize abilities of these phones such as determining it's geolocation, internet connectivity and so on. Packed in a single mobile application and delivered to the people by well known channel, such as Android Market, it may find it's users and get quite popular.

And that's the whole motivation behind this project: to make something for the people, that will help them to spare their money or at least to spend them for comfortable and fast transportation as well as to increase usage of existing taxi services by taking advantage of so many smartphones spread amongst the people.

## 1.2 Result of my work

With a vision of a system described above, I have implemented client-server application with a goal of providing simple mobile-based solution for as-much-as-possible realtime ridesharing. Main features of the system are:

- Availability - the only requirement is an Android-powered phone with its operating system of version 2.2 or higher
- Scalability - thanks to *Google App Engine*<sup>2</sup> cloud-based framework
- Dynamic behaviour - almost instant response about whether or not there is a match for users request
- Smartphone capabilities utilization - allowing users to use their current geological location or built-in maps for specifying starting points and endpoints of their rides
- Multipart rides support - users are free to participate on just a part of a ride as long as it is still favourable

---

<sup>2</sup>Google App Engine: <https://developers.google.com/appengine/>

## 1.3 Thesis outline

The thesis is divided into following chapters:

1. **Introduction**

Explains what is the goal of this work and why would someone do that.

2. **Related work**

A research on already existing solutions related to this work, their evaluation, discussion and most important observations.

3. **Requirement analysis and specifications**

Discussion on what is necessary for an application like this and general information about how the result should look like.

4. **Architecture and design**

Describes of what pieces is the application made and how are they connected together and provides some pictures of its actual appearance.

5. **Algorithms**

Explains all the behaviour hidden to a user, i.e. what happens to a request for shared ride once it's sent to the server.

6. **Implementation**

Covers all of the interesting parts of the actual Java implementation of both client and server side of the system.

7. **Evaluation**

Test were performed on the system to measure its qualities. Results of those experiments are discussed in this chapter.

8. **Conclusion**

Summary of what was done and achieved in this work.

## Chapter 2

# Related work

### 2.1 The problem

In the previous chapter an interesting system was introduced. Let's take a look on a problem that this system is supposed to solve. Informally, the problem is to "allow people to easily plan a shared ride using a single taxi". That is quite a general problem which can be potentially solved in many different ways. And, of course, some of them are already implemented and available. These attempts of solving given problem come in different shapes with different ideas behind them, but are all related to this paper since they all aim at "allowing people to easily plan a shared ride using a single taxi".

Some of these applications are listed and described below. Following list contains all software solutions for our problem which I was able to find on the internet and which seemed to be usable and functional at that specific time when I was searching for them.

### 2.2 Existing solutions

There are lots of applications (not exclusively for the mobile platform) providing taxi services in different ways. However most of them can offer nothing more to a user than approximation of price for a ride or simple booking of the vehicle for specific date and time. Only a very few are capable of organizing a shared ride.

These ones differ one from another significantly, each having its pros and cons. For every single one of them I will provide following properties: both brief and more detailed description, interesting observations with relation to my work, possible opportunities for improvement and a brief summary.

### 2.2.1 Cabcorner.com

Cabcorner.com<sup>1</sup> is originally web-based application for planning shared taxi rides. Currently there is also mobile version available, but only for iOS-powered devices.

#### 2.2.1.1 Description

According to available information, cabcorner.com is web-based application allowing its users to plan a ride and display it on cabcorners website. Other users can then find this ride using the same website and potentially join it.

Cabcorner provides its services only in some of the major cities around the world, not including Prague, while the largest community of active users seems to be located in New York. It's difficult to make a reliable estimate about how many people use this application regularly, but from relatively low numbers of available rides I would say not much.

As I mentioned before, a mobile version of cabcorner for iOS devices is also available and according to available videopresentations it works in exactly the same way as its web-based counterpart.

#### 2.2.1.2 Interesting observations

**HotSpots** I personally consider cabcorners solution of HotSpots<sup>2</sup> interesting. Not only that cabcorner placed its HotSpots to important traffic nodes (such as railway stations), but it placed most of them inside shops while additionally providing its users with vouchers enabling them to buy local goods for lowered prices. I'm not sure that stores are ideal places for serving as HotSpots, but it's quite a unique solution worth mentioning.

#### 2.2.1.3 Improvement opportunities

**HotSpots** Despite the fact that I stated this feature as "an interesting observation", at the same time it is an improvable weakness, mainly because of these reasons:

- Shops can be crowded making it difficult for the passengers to find each other.
- The main intention of users of this service is to travel from somewhere to somewhere else, not to go shopping.
- This concept is much more vendor-friendly than user-friendly.

**Optimality of the route** Cabcorner presumably supports onsets of the passengers along the way. That's the case when one passenger starts the ride and navigates the driver to pick up the others. However, there seems to be no guarantee, that such a ride won't be eventually longer and less advantageous than a ride of a single person due to possible detours. This is especially important on short rides where the effectiveness of a shared ride could drop significantly with each detour.

---

<sup>1</sup>Cabcorner: <http://www.cabcorner.com/>

<sup>2</sup>HotSpot is a place defined by the taxi sharing service where people participating in the same ride should gather before the ride begins.



**Fare sharing** Along with the problem of effectiveness there is another major issue when it comes to shared ride with multiple start points: how to split the fare amongst the passengers fairly? Since every passenger may travel different distance, every one should logically pay different price. This is something that probably didn't come to mind of cabcorner's creators as their product doesn't handle this in any way and I can imagine that leaving the splitting of the fare to users themselves could lead to unpleasant situations.

#### 2.2.1.4 Summary

The way cabcorner works is interesting, but it still reflects the fact that it originated as a web-based service. A mobile-based application should work in more dynamic way allowing its users to plan the rides just dozens of minutes or few hours ahead, not whole days in the future.

### 2.2.2 RideAmigos.com

A web application for ride-sharing.

#### 2.2.2.1 Description

RideAmigos<sup>3</sup> is a web-based service providing ride-sharing capabilities for different types of vehicles, not only taxis. It's not even limited to sharing rides between individuals, but allows to plan rides for larger groups of people. It all sounds very promising and is supported by positive feedback from "happy users", but reality is quite different.

#### 2.2.2.2 Interesting observations

**Ridesharing for larger groups** The only interesting thing about RideAmigos is the option to plan a shared ride for larger groups of people (like school trip or business trip for whole company).

#### 2.2.2.3 Improvement opportunities

**Various issues** Even after registering and trying to use the service I still don't know, how it would look like if someone really found my ride and wanted to join it. How would we arrange from where will we start? Would I obtain any information about the other user from the RideAmigos service or not? Would I have to contact him and arrange everything? The answer to all these questions is "I don't know". It is obvious, that RideAmigos doesn't provide enough information to its user (frankly, it hardly provides any information) which is enormous issue since being informed is the only way to smoothly arrange a ride in which multiple users are involved.

---

<sup>3</sup>RideAmigos: <http://www.rideamigos.com/>

#### 2.2.2.4 Summary

Based on my experience, RideAmigos claims to provide wide range of ridesharing options, but completely fails to deliver them. Its interface is non-intuitive and lack of information about everything disables potential users from using it conveniently.

### 2.2.3 Taxi Share - Chicago

Android ridesharing application designed specially for Chicago.

#### 2.2.3.1 Description

As the name suggests, Taxi Share - Chicago<sup>4</sup> was made exclusively for people living in or visiting Chicago. Since its focused on a single city, it works in a slightly different way compared to other applications.

#### 2.2.3.2 Interesting observations

**The way the rides are created** As was mentioned before, this service is tailored to fit just and only Chicago city. That fact is reflected in a way the rides are created. Unlike other services which usually let a user pick the start and final destination, Taxi Share - Chicago allows user to freely specify just the start location. The final destination has to be picked from a list of predefined locations which covers pretty much all points of interest in the city. The real use case scenario then look like this: one picks a destination from the list and then specifies from where he/she wants to go. Other users can then see this ride and join it.

**Direct communication** Quite rare, but very smart and handy feature of Taxi Share - Chicago is kind of instant-messaging client built into the application enabling users to communicate directly before the ride begins. At first sight it may look unnecessarily but I consider this functionality highly usable. Not only that it solves the problem of identifying users at the start location, but it also provides an option to share other any additional information (like that one of passengers needs to transport large-sized luggage that will take up the whole trunk). According to developers of the application, this feature was designed to provide a way for the users participating on the same ride to confirm their intentions and to share any necessary information.

#### 2.2.3.3 Improvement opportunities

**Onsets along the way** As this application allows onsets during the way, it should also ensure effectiveness of the ride and handle appropriate splitting of the fare amongst passengers. Since it does not do the former nor the latter, its opportunities for improvement are exactly the same as those for cabcorner in 2.2.1.3 and 2.2.1.3.

---

<sup>4</sup>Taxi Share - Chicago: <http://taxisharechicago.com/>

#### 2.2.3.4 Summary

Taxi Share - Chicago is highly specialized application limiting its functionality to one single city. The way the rides are planned and created is also adapted for this philosophy. One can assume that this application works quite well (or is at least usable) in Chicago, but it would be hardly possible to generalize its concept for being usable anywhere else. Noticeable feature and a good idea is built-in instant-messaging client.

### 2.2.4 FareShare

Mobile application for iOS devices owners. From all applications related to my work that I have found during my thorough search throughout the web, this one is probably closest to the ideal “mobile application for taxi ridesharing”.

#### 2.2.4.1 Description

FareShare<sup>5</sup> is the only mobile-only application providing taxi ridesharing capabilities that I managed to find. As such, it encompasses few utilities that come in handy while planning a common ride using a mobile phone just a few minutes before it happens. The application is designed to work exclusively in New York and one can hardly estimate by how many people its used.

#### 2.2.4.2 Interesting observations

**Redefining HotSpots** For almost every single application listed above I have mentioned its own solution for HotSpots. Without an exception, all other applications see HotSpot as places where people gather before the ride starts. FareShare, however, comprehends this term in quite a different way. For FareShare, a HotSpot is a place that most frequently ends up being someones destination (such as pub, club, theater, historical monument and so on). These places are then visible to users, presumably just for inspiration. Despite the fact this is something unprecedented, I don’t consider this functionality being necessary for application of such a kind.

**Meeting places** Since FareShares understanding of HotSpots is unique, I will use “meeting place” instead to denote a place where involved passengers meet each other before the beginning of their ride. FareShare does not define any fixed spots as meeting places, it rather generates these dynamically for each single ride according to current position of participating users. If a spot generated by the application does not fit ones needs, it can be adjusted by selecting another point from the map. Note that I was not able to find out whether this option is available only to the user who creates the ride or to other users as well.

---

<sup>5</sup>FareShare: <http://faresharenyc.com/>

**Passenger identification** Unlike other systems which either not handle this issue at all or provide just a partial solution, FareShare is equipped with rather elaborate technique for allowing its users to easily recognize each other: every user has an option to upload his/her photo that will be displayed to others enabling them to simply and positively recognize corresponding person, or one can select a notable piece of ones clothing from a list of predefined items (such as “yellow glasses” or “pink vest”), that should help other distinguish that person from the crowd. I don’t know if selecting multiple pieces of clothing is possible but even one would definitely do the trick.

**Final destination selection** It seems that FareShare does not allow selecting a specific address as the endpoint of ones ride. It only permits districts or city blocks to be selected. This behavior may look flawed but it does make sense together with possibilities described below.

**Multiple endpoints** According to promotional video FareShare manages to arrange a ride from common starting point to multiple endpoints. In a situation when users travel in the same direction but not to the identical place, the one that arrives to his endpoint earlier pays relative amount of fare and gets off leaving the other passenger to continue to his desired destination.

**Fare splitting** To provide comfortable user experience, FareShare deals with computation of relative prices for individual users. When in the first endpoint, application offers an estimated price to the user currently getting off who then adjusts this approximation to correspond with reading from the taximeter, pays the price and leaves the vehicle.

**Payment method** Very rare and extraordinary is a possibility to pay cashless for the ride. In case both users have their PayPal accounts set up correctly and they both have it linked with the app, then the one who gets off earlier can transfer appropriate amount of money that he is supposed to pay completely cashless to the bank account of the other user who then pays the whole price for the ride at its very end. This idea is remarkable especially because it eliminates delays making the ride even more effective.

**Passenger rating** FareShare uses simple yet effective system for rating its users by other users with whom they shared a ride. After each ride every passenger is asked following question: “Would you travel by this person again?”. The only possible answers are “Yes” and “No” each resulting in rising or lowering credibility of rated user respectively.

#### 2.2.4.3 Improvement opportunities

**Limited number of users per ride** Although it’s not explicitly stated, FareShare seems to support rides shared by no more than two users. Even the promotional video clip shows just two people sharing a single taxi. With some of its features looking like they were built just for two users per ride, it doesn’t seem probable that FareShare will ever support more than two people in a vehicle.

#### 2.2.4.4 Summary

FareShare is sophisticated mobile application with lots of useful tweaks, but it all comes at price of not being able to serve more than two people per ride. For that twosome it provides an ideal solution for quick and easy planning and realization of a shared ride though. I was particularly impressed by the option of cashless transactions and implementation of users rating.

### 2.3 Related work summary

Numerous already existing solutions were listed and described above, each of them with different properties and features. So what is the conclusion about current taxi ridesharing possibilities? Well, I would sum it up like this: currently, there are few ridesharing systems out there allowing one to plan shared ride in one way or another, but only few of them are based on mobile platform and even those running on mobile phones have a potential for improvement.

These are the most important observation I have learned during my research on existing ridesharing solutions about how newly created application should look like:

- The application should be mobile based so anyone could reach it virtually from anywhere
- The application should work in as much dynamic way as possible, ideally returning a ridesharing opportunity immediately after receiving new request
- The application should allow its users to utilize capabilities of smartphones it would run on such as determining users geolocation etc.
- The system should allow multiple starting points and multiple endpoints and it should take care of fair splitting of fare amongst passengers
- The system should be able able to work anywhere, not just in specific area (such as single city)

I believe that an application meeting these standards would serve its purpose very well and would help its users to spare their money and use taxis effectively.

### 2.4 Surveys and external references

While performing my research on already existing solutions, I have also searched for existing algorithms usable for spatio-temporal clustering. I have found some papers about spatio-temporal clustering methods, but virtually none of them was directly related to my work.

### 2.4.1 Spatio-temporal clustering

Since spatio-temporal (ST) clustering is a kind of data mining, many papers cover this problem from scientific point of view presenting various methods for different types of ST data [5]. Majority of these methods aim at something completely different than I need in my work like analyzing time series data, discovering common patterns in ST data or analyzing trajectories of moving objects. Other methods based on ST clustering are focused on analyzing events, like crime in urban localities [4] or spreading of diseases amongst people [8]. Therefore none of them is quite suitable for my work.

### 2.4.2 Dial a ride problems

There is another type of problems that seems to be much closer to my work, though. The problem is called dial-a-ride problem (DARP) and in general the goal is to propose set of routes as cheap as possible satisfying certain criteria. There are again various methods for solving this kind of problem ([2], [3]), and various instances of DARP with different constraints (like time windows [6] or limited vehicle capacity [1]).

However, even DARP solving algorithms are not quite applicable to the problem my system is supposed to solve, since DARP describes and solves routing problem from fleet owner's point of view. That means DARP solver needs to know certain information about vehicles (such as their numbers and properties). This is something that my system can never know as it works solely with requests from people who want to travel and do not possess any knowledge about what vehicle will pick them up or even if there is any vehicle available. That's why I eventually came up with my own algorithm (in fact three of them) for matching individual requests (see 5).

## Chapter 3

# Requirement analysis and specifications

### 3.1 Requirement analysis

Based on my observation gathered during examination of already existing applications, I made up a list of features that I consider being vital for any new application aiming at providing ridesharing capabilities. These substantial functionalities are following (in order of decreasing importance): ride effectiveness maximization, providing enough information, ride organization and fare splitting. The additional ones are: passenger rating, passenger recognition and methods of payment. Lets now go through them and have a closer look on each of them.

#### 3.1.1 Essential functions

Following features are those that I consider being essential for any new ridesharing application in order to work both properly and effectively.

**Efficient shared ride matching** Although anyone would automatically expect any application of this kind to create rides that are as much effective as possible (after all that's what these applications are meant to do), I think this part is not only worth mentioning, but I consider it being most important since not all of the existing solutions guarantee maximal effectiveness, for example by not addressing the problem of multiple starting points that could eventually make the ride not effective at all because of long detours. This is something that I think should never happen in such an application.

**Providing enough information** Providing enough information, another aspect one may take for granted yet it's not incorporated in every existing system so far. RideAmigos is perfect example of application which doesn't provide enough information. Briefly, it is virtually unusable. A user should be aware of a state of the application at all times and should be able to use the application without any extensive learning. RideAmigos, for example, completely fails in both the former and the latter. By "enough information" I mean mainly the actual date and time of confirmed ride, who else is participating in the ride, estimated total length and estimated total price of the ride.

**Ride organization** In other words "How will the ride be organized?" There are several approaches in already existing applications:

- predefine fixed HotSpots and let users start from one of them
- create HotSpots dynamically and let users start from one them
- let users choose common starting point and general destination area
- let users go from anywhere to predefined locations

All of those approaches have their pros and cons, but the list lacks the one I find most flexible and convenient: let the user freely choose both starting point and endpoint with the option to pick passengers up and drop them off along the way. The reason it was not listed is simple: none of the existing applications supports this concept. That's a pity since I see a lot of potential in this specific type of ride organization.

**Fare splitting** The application itself should handle correct and fair splitting of the fare between users involved in a ride. Not only that it's much more comfortable for each user to know exactly how many is he/she supposed to pay, but I also believe that it may prevent potential conflicts caused by arguing about each passengers relative price.

### 3.1.2 The additional features

All functions listed below are deemed to be something an application should have, but are not that crucial to prevent it from working. I can imagine adding these features into the application later once all vital parts are up and running.

**Passenger rating** Very handy yet dispensable functionality is rating of the users by other users. Such a functionality would probably result in better user experience, but the same application may work equally well without it.

**Passenger recognition** Quite an important part of the whole process when it comes to applications preferring single starting point on which individual users need to find each other. As I favor the concept of multiple starting points, this problem seems less important since with multiple starting points one just waits patiently on the edge of the road for taxi to arrive. Some kind of method for identifying fellow-passengers could be built in the application for the case it's needed, but it's not that important.



**Methods of payment** Usually people pay in cash for the ride. An interesting option was introduced by FareShare to pay cashless using PayPal. The idea is definitely brilliant, but tweaks like this may not be that usable as they require additional effort (like setting up a PayPal account) from each and every user to work properly. Because of that, it may happen rarely that all passengers participating in one ride would meet these requirements allowing that feature to work.

## 3.2 Specification

In this section I will describe specifications for my application according to key features mentioned in previous section. This specification covers users point of view and provides an overview of inputs and outputs of the application. More detailed technical specification will be given later in this work.

### 3.2.1 Application inputs

Since the goal is to create mobile-based application designed for taxi ridesharing, the resulting program should require some information in order to work correctly and should be able to accept users requests.

#### 3.2.1.1 Authentication

Users needs to be able to authenticate himself against the application. This step is necessary to prevent fake requests being created from users that are not really interested into using the service as well as to ensure applications correct functionality that requires existence of at least one Google account in users device.

#### 3.2.1.2 Ride specification

A complete specification of a ride provided by the user to the application for processing consists of following pieces:

**Starting point** The point from where the user wants to travel. This place can be specified by selecting a specific point from the map or by looking up a certain address. Note that unlike some of already existing solutions I plan to give the user an option of completely free choice by not defining any fixed HotSpots or places of such a kind. Furthermore, application will enable the user to utilize geolocation capabilities of his phone to select his current position as starting point for the ride.

**End point** Analogically to the starting point, an endpoint will be selected in the very same way by either selecting a place from the map, looking up a specific address or using geolocation services of ones mobile phone.

**Date and time of the ride** Information about when the user wishes to go. The only limitation for this entry is that it must not be in the past. No restriction on how far in the future may the ride be planned will be introduced, but it's not expected that rides more than a few days in the future will be planned since that would go against the very idea of the application. The ridesharing experience is supposed to be as much spontaneous as possible with this application preferring rides planned just a few moments ahead over those planned long time before the ride itself.

### 3.2.2 Outputs of the application

As the result of users interaction with the application, the outputs described below will be returned by the system.

**List of users requests** Once sent to the system, user will be able to list all entered rides each of them containing the same information that were given to the application in time of creation of that particular request.

**Notifications** The application will be able to actively notify the user in case that someone else joins a ride. This notification should be optional and in case of being active, it should signalize the occurrence of an event by both audible signal and a small icon displayed in status bar.

**Ride offers** Notifications should signalize that match was found for some of users requests. Ride offer, based on this match, will be proposed to the user for confirmation.

## Chapter 4

# Architecture and design

### 4.1 Architecture

This section shall describe the overall architecture of the system covering both client and server side by presenting an essential building blocks of the whole application and the way these are connected to each other in order to create an operational system.

#### 4.1.1 Model

The system is planned to follow the classical client-server model providing reliable platform for its users. As usually, the whole computational power will be provided by the server making clients relatively thin by reducing their task to simply being able to send users input and to display servers outcome.

All algorithms and processing done by the server will be done in centralized way meaning that interactions will only occur between the server and individual clients, not between the clients themselves.

##### 4.1.1.1 Building blocks

Both the client and the server side can be decomposed to the individual blocks they are made of (in this case these will be plain Java objects). Since the server part makes all the work, I will mainly describe the parts forming the server side of the system.

**Ride request** An important and quite essential is the “ride request” block. Each of these represents a single request created by the user and stored somewhere on the server for further processing each encapsulating the information described in [3.2.1.2](#)

Algorithms then work on the complete set of currently available requests trying to match them together in the best possible manner.

**Ride prototype** Once the algorithm finishes processing of requests, it may or may not produce a set of “Ride prototypes”, candidates that again may or may not make it to the stage of a ride offer. Each of them contains following type of information:

- Direction of the ride
- Involved users
- Ride requests this prototype consists of
- Additional, implementation specific necessities

Ride prototypes are subjects to another process that either approves or rejects the prototype based on certain criteria.

**Ride plans** Prototypes lucky enough to make it through the approval process are considered being sufficiently good to be offered to users as a possibility for shared ride. They come to them in a form of “Ride plans”, more complex objects than “Ride prototypes” additionally describing the ride by:

- A set of waypoints along with an order they will be travelled through
- Estimated total length and total price of the ride
- Relative prices for each individual user

#### 4.1.2 The workflow

All the building blocks are used in a certain moment during applications lifetime. I used BPMN flowcharts to capture the way the system internally works. One flowchart is provided for each individual part of the system as well as for both of them interacting with each other.

**Server part** Following figure demonstrates how the server internally works. Two figures are provided - one for clustering algorithm and the other for online one. Since initialization related processes are not really relevant, this figure captures servers internal workflow after it receives new request.

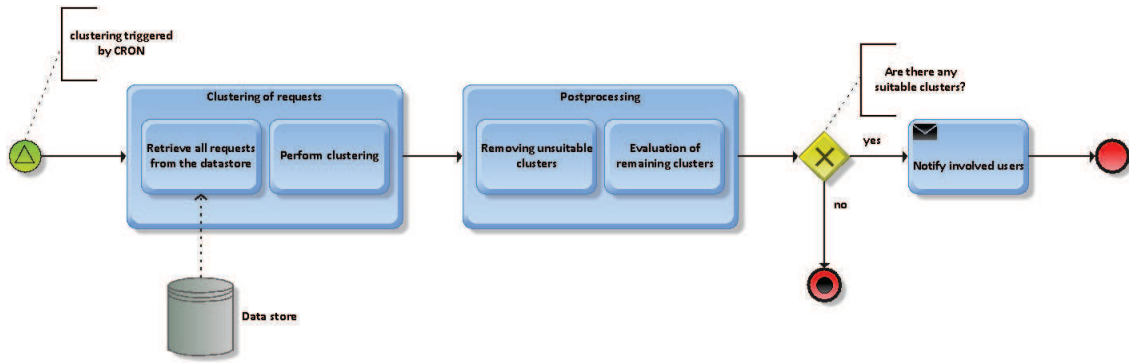


Figure 4.1: Workflow of clustering algorithms

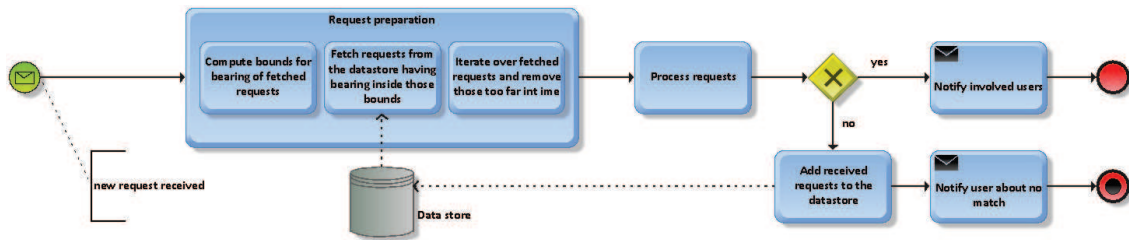


Figure 4.2: Workflow of the online algorithm

**Client part** As I mentioned earlier, client application is very thin. It basically only sends and received requests, therefore it wouldn't make sense to provide client-only workflow as it is bound closely to the server part. Clients behavior will be better illustrated by its interaction with the server. Note that the parts of the applications lifetime in which the application is being suspended, terminated or brought back to active state will be omitted since these are Android OS specific and as such they are irrelevant for the general insight on the application.

**Mutual interaction** Finally, the most interesting part - mutual interaction between the server and the client - is represented by following figures. There are two of them for the same reason as there are two figures in "Server part" section - the two implementations behave differently not only internally inside the server but also while interacting with the client application.

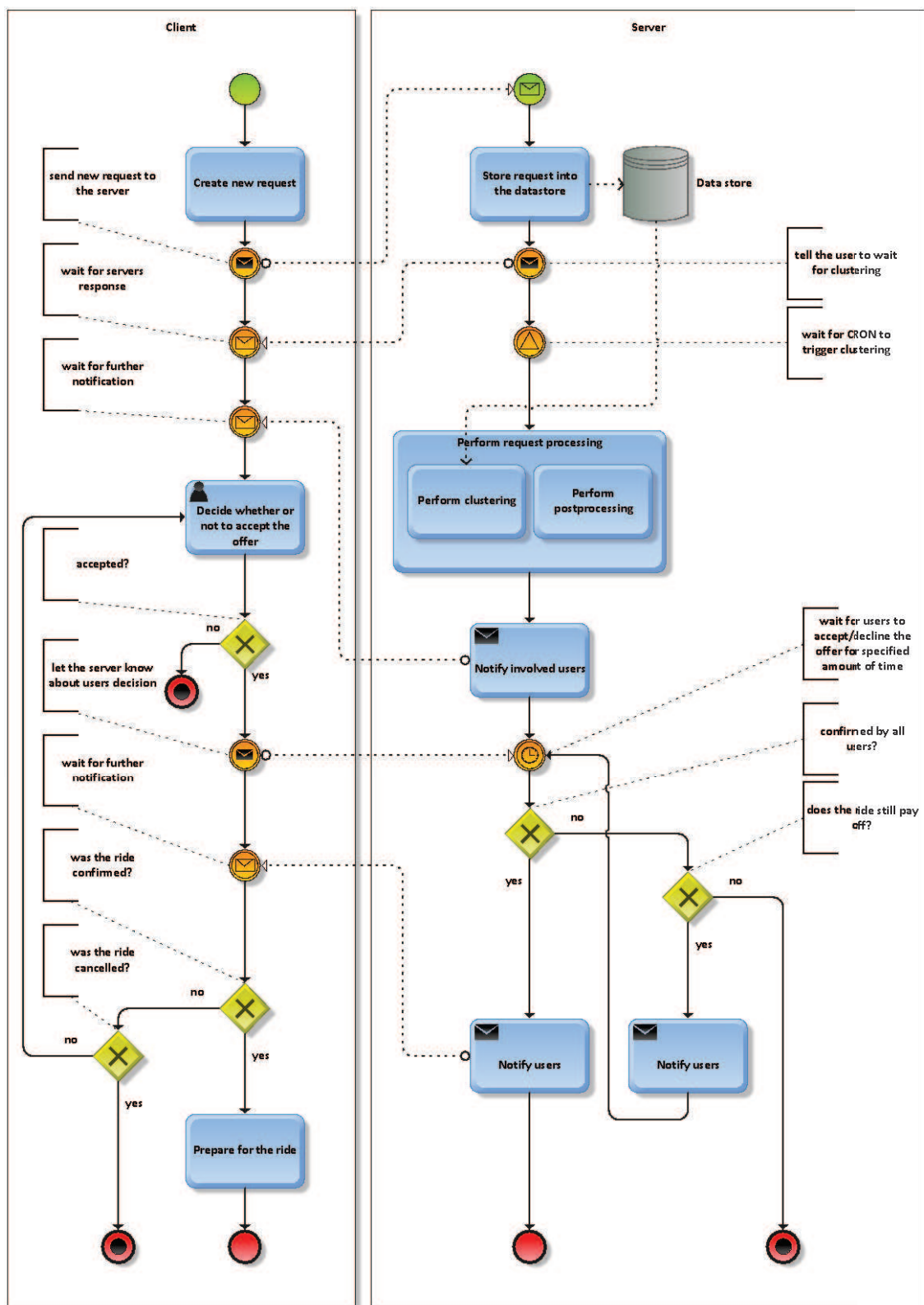


Figure 4.3: Workflow of server-client intercommunication for clustering algorithms

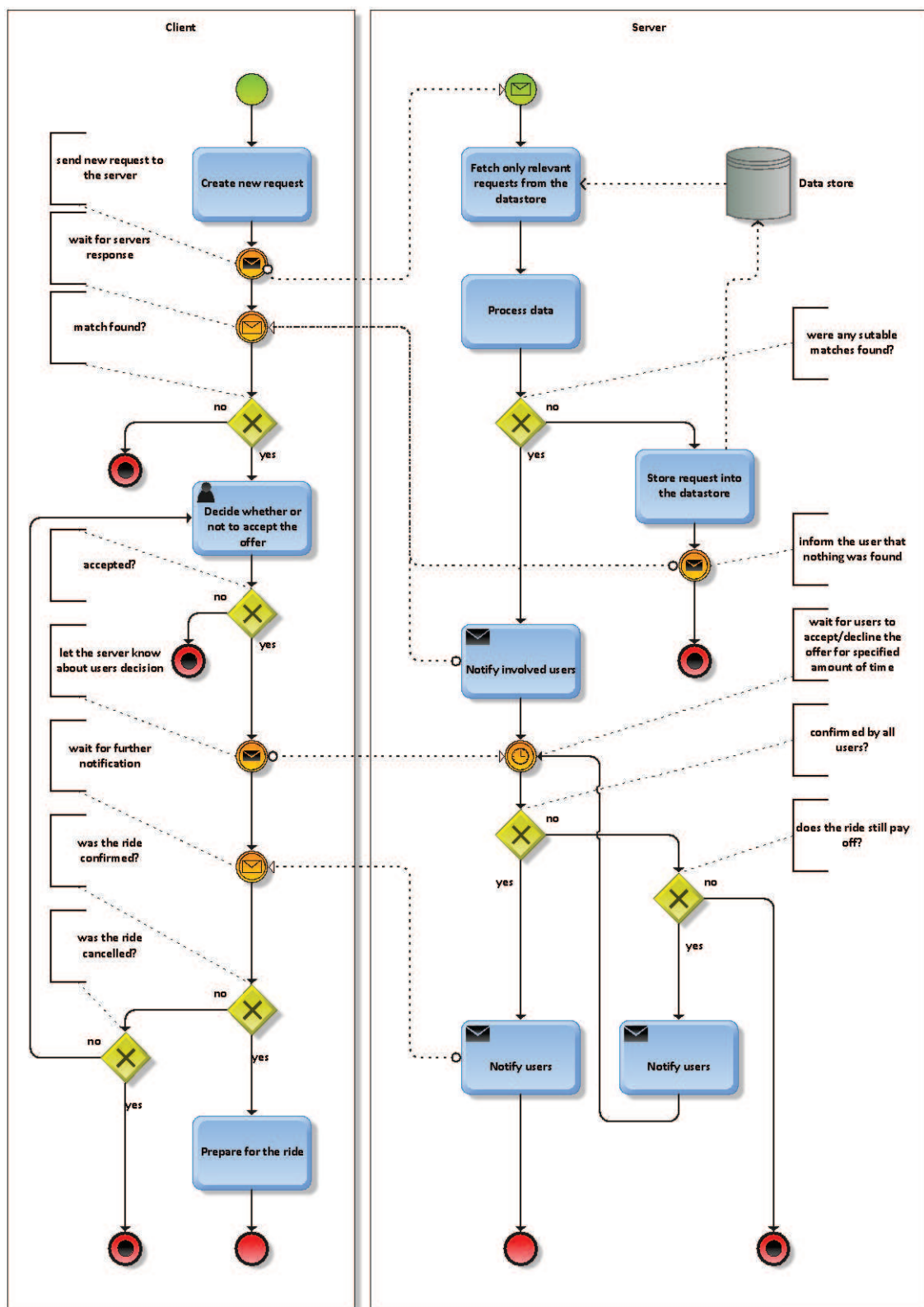


Figure 4.4: Workflow of server-client intercommunication for online algorithm

### 4.1.3 Technical point of view

So far the architecture of the system was presented in a general, almost in theory. But the implementation itself is nearly as important as the concept it is based on, therefore I will briefly introduce technical solutions on top which the system will be built.

**Google App Engine** Server part will be implemented using Google App Engine - cloud solution providing highly scalable platform for varying kinds of web applications.

Since the App Engine handles deploying, redeploying, starting and terminating instances by keeping track of number of requests the application needs to serve, it fits my needs very well.

App Engine currently supports web application developed in Java, Python and (still experimentally) Go. As my preferred programming language is Java, my app will be written in it.

**Android operating system** Android OS - developed by Google - is quite popular these days. Millions of devices powered by a system with green robot in logo are activated every day. Having such a huge user base is a vital property that qualified Android to be the system hosting my client application. That, and the fact that Android application are developed in Java as well.

Another big advantage of Android with relation to server-side implementation is that it's developed and maintained by the same company - Google. That makes both sides of the application highly compatible and eliminates some obstacles one would need to overcome while developing each part of the system in an environment from different provider.



## 4.2 Design

Since the application is supposed to be used by whoever downloads it, it has to implement a user interface. In this specific case the user interface will form major part of the client-side application.

### 4.2.1 Android philosophy

By the Android philosophy, applications should be built in such a way that they both use as much already available resources (including parts of other applications) and allow their parts to be used by other applications.

This approach directly implies that all Android applications are built from common building blocks that may be potentially reused or shared. More complex structures and behavior can then be achieved by combining these blocks together.

#### 4.2.1.1 Android activities

The most essential building block of each Android application is so called “Activity”. An activity is a container for both application logic and user interface elements designed specifically to allow a user to perform a specific task. Each activity is usually represented by one window taking up whole screen of the device. Activities have a non-trivial lifetime, but that is irrelevant from users point of view. What matters to a user is how the activity looks like.

#### 4.2.1.2 Screenshots

Following pictures capture individual activities of the client application in a form they actually look like on a real<sup>1</sup> device.

Figure 4.5 shows all significant parts of clients interface. Main window of the application (4.5(a)) and new ride activity (4.5(b)) are the most important activities.

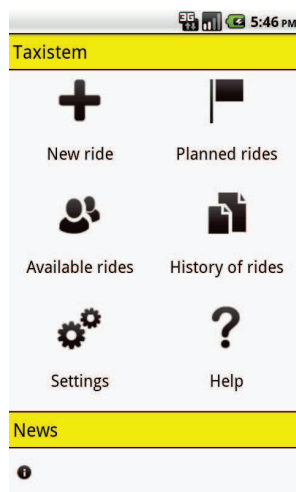
While creating a request for a ride, user can select starting point and endpoint either by picking up a point from a map (4.5(c)), by using geocoding service which translates textual address into geographical location (4.5(d)), or by using current geographical location determined by the device.

Once the user submits the request, two things can happen: either a match is found on the server and shared ride is offered to the user immediately (4.5(f)) or no match is found and user is instructed to wait until one occurs (4.5(e)). Once a match is found, user is notified via standard Android status bar notification (4.5(g)).

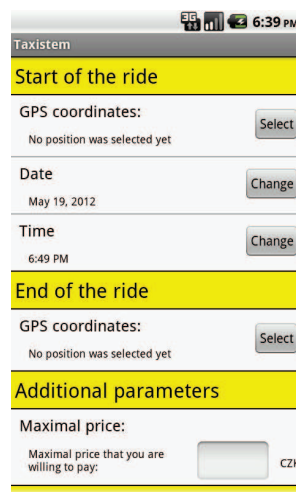
To make it easier for a user to work with this application, the application once requests users permission to use credentials stored in the device (4.5(h)). In case user grants the permission, application never asks for username nor password making user experience more comfortable.

---

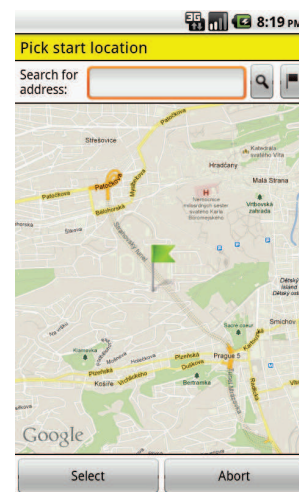
<sup>1</sup>In fact, almost all images were taken using Android emulator, not a physical device, but that makes absolutely no difference



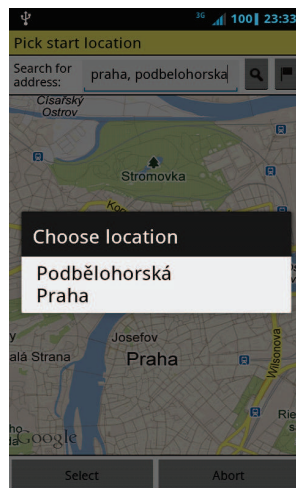
(a) Main window of the application



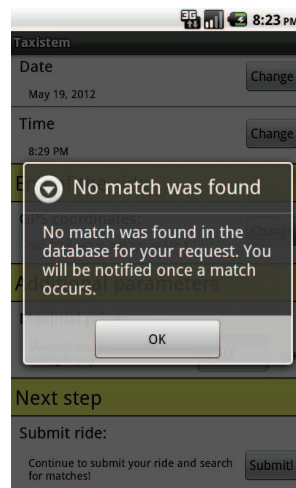
(b) New ride activity



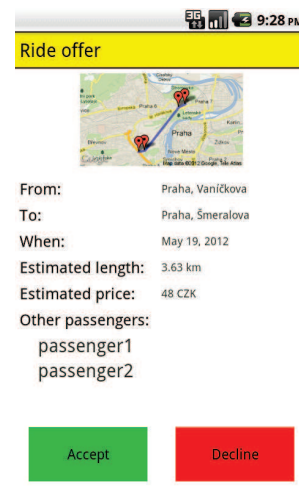
(c) Picking starting point from a map



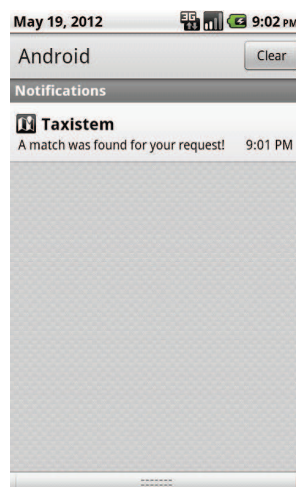
(d) Finding a point on map using geocoding service



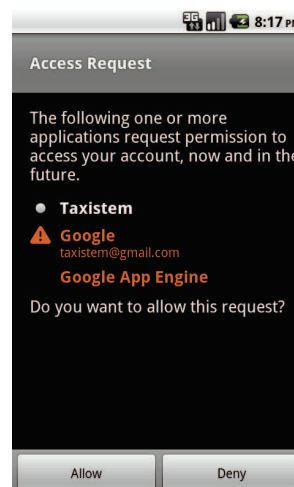
(e) No match found for request



(f) Ride offer - match found for request



(g) Notification about found match



(h) Request for permission to use users credentials

Figure 4.5: Screenshots of the client application

## Chapter 5

# Algorithms

Earlier in this paper I have mentioned processing of requests several times, but I have never explained what exactly that means. In this chapter an explanation will be given along with few figures depicting the working process of algorithms.

### 5.1 Request matching

The vital functionality of the server part of the system is an algorithm matching active requests and creating groups of similar ones that could potentially form a shared ride. As my system ended up with two different methods for request matching, I will provide two descriptions, one for each of them.

#### 5.1.1 K-Means algorithm

The very first idea how to match individual requests from users was to use well-known clustering algorithm called “K-Means”. As its name suggests, it works with mean values, specifically with  $k$  of them. A general definition-like description follows.

Let  $S$  be a set of  $n$  objects (in my case requests for a shared ride, but lets just call them “points” for the sake of generality) on which the clustering is about to be performed. Let  $K$  be a set of  $k$  objects (lets call them “etalons”) that will represent the individual clusters once the clustering process is finished.

An essential condition for *k-means* to work is that one needs to be able to determine a “distance” between any two objects from  $S$ . More formally, a function must exist such that for any pair of objects from  $S$  on its input returns a single value from  $R$  - their distance.

The algorithm works in following steps which are repeated until terminating condition is met.

1. Etalon initialization - this is usually done by randomly assigning each etalon to one of the points  $S$
2. For every point compute its distance from each etalon and assign this point to the closest of them.

3. Adjust etalons so that they become mean values of points assigned to them.
4. If assignment of all individual etalons did not change during last two iterations, terminate. Otherwise continue with step 2.

Once finished, algorithm returns  $K$  with its elements (*etalons*) representing cluster centers and each assigned with some of the *points*.

---

**Algorithm 1** Matching requests using K-Means method
 

---

**Input:** number of clusters

**Output:** ride prototypes

*requests*  $\leftarrow$  all requests from the datastore

*etalons*  $\leftarrow$  *numberOfClusters* new etalons

**for all** *etalons* **do**

*randomRequest*  $\leftarrow$  *requests.get(random)*

*etalon.startLocation*  $\leftarrow$  *randomRequest.startLocation*

*etalon.endLocation*  $\leftarrow$  *randomRequest.endLocation*

*etalon.time*  $\leftarrow$  *randomRequest.time*

**end for**

*assignmentsChange*  $\leftarrow$  *true*

**while** *assignmentsChange* **do**

**for all** *requests* **do**

*mindistance*  $\leftarrow$  *infinity*

**for all** *etalons* **do**

*distance*  $\leftarrow$  *computeDistance(etalon, request)* {using formula in 5.6.1}

**if** *distance*  $\leq$  *minDistance* **then**

*mindistance*  $\leftarrow$  *distance*

*closestEtalon*  $\leftarrow$  *currentEtalon*

**end if**

**end for**

*closestEtalon.assignedRequests.add(request)*

**if** *closestEtalon*  $\neq$  *closestEtalonFromPreviousIteration* **then**

*assignmentsChange*  $\leftarrow$  *true*

**end if**

**end for**

**for all** *etalons* **do**

*currentEtalon.startLocation*  $\leftarrow$  mean value of start positions of assigned requests

*currentEtalon.endLocation*  $\leftarrow$  mean value of end positions of assigned requests

*currentEtalon.time*  $\leftarrow$  mean value of time positions of assigned requests

**end for**

**end while**

**return** *etalons*

---

### 5.1.2 Directional algorithm

My so called *Directional matching algorithm* is an algorithm I came up with which is based on *k-means* idea, but utilizes additional, problem-specific knowledge. This algorithm uses a direction (a bearing) of each individual ride request which can be computed quite easily using starting point and endpoint of that particular request.

Let  $S$  again be the set of  $n$  points. Let  $K$  be a set of  $k$  etalons.

The algorithm itself then repeats following steps:

1. Initialize etalons - again by randomly assigning etalons to points
2. For every point compute its distance from each etalon and assign this point to the closest of them.
3. Adjust etalons so that they become **median** values of points assigned to them.
4. If assignment of all individual etalons did not change during last two iterations, terminate. Otherwise continue with step 2.

Once this algorithm terminates, it returns exactly the same output as *k-means*, that is a set of etalons adjusted to be centers of requests. The only, but greatly important, difference is in an assignment of individual requests to etalons.

**Note:** By replacing mean value by median value in step 3 the clustering method changed from *k-means* to *k-medians* which is an algorithm with exactly the same properties and workflow as *k-means* up to the step of adjusting etalons.

---

**Algorithm 2** Matching requests using Directional method
 

---

**Input:** number of clusters**Output:** ride prototypes*requests*  $\leftarrow$  all requests from the datastore*etalons*  $\leftarrow$  *numberOfClusters* new etalons**for all** *etalons* **do**    *randomRequest*  $\leftarrow$  *requests.get(random)*    *etalon.startLocation*  $\leftarrow$  *randomRequest.startLocation*    *etalon.endLocation*  $\leftarrow$  *randomRequest.endLocation*    *etalon.time*  $\leftarrow$  *randomRequest.time***end for***assignmentsChange*  $\leftarrow$  *true***while** *assignmentsChange* **do**    **for all** *requests* **do**        *mindistance*  $\leftarrow$  *infinity*        **for all** *etalons* **do**            *distance*  $\leftarrow$  *computeDistance(etalon, request)* {simply as difference in bearings}            **if** *distance*  $\leq$  *minDistance* **then**                *mindistance*  $\leftarrow$  *distance*                *closestEtalon*  $\leftarrow$  *currentEtalon*            **end if**        **end for**        *closestEtalon.assignedRequests.add(request)*        **if** *closestEtalon*  $\neq$  *closestEtalonFromPreviousIteration* **then**            *assignmentsChange*  $\leftarrow$  *true*        **end if**    **end for**    **for all** *etalons* **do**        *currentEtalon.bearing* = *findMedianBearing(assignedRequests)*    **end for****end while****return** *etalons*


---

## 5.2 Prototype evaluation

Both algorithms described above produces identical results: sets of etalons with requests assigned to them. But these clusters are not yet ready to be presented to users as ridesharing offers. In fact, it may happen that several requests assigned to the same etalon differ greatly one from another being completely incompatible in terms of ridesharing.

This is where the second phase of servers work takes place. This time returned etalons (lets call them “cluster” from now on since that is what they represent) are fed to an evaluation algorithm which task is to make a verdict about every etalon: approved or denied.

The way the algorithm works differs slightly for each of the clustering algorithms, but generally it works as follows:

- A representative specimen for the cluster is selected (that may be the cluster itself or one of assigned requests)
- Other requests are compared appropriately to this specimen
- If any of those requests is not “similar enough” (similarity conditions differ for each implementation) to the specimen, it is removed from the cluster
- If one or more requests were removed, run the evaluation again on the reduced cluster
- If too many requests were removed during the process (only one request remains assigned to the cluster), terminate and discard current cluster
- If all remaining requests are similar enough and there are two or more of them, terminate and approve this cluster

The result of evaluation algorithm is set of clusters, each of them possibly reduced by a few requests. This set can be empty if none of clusters on the input was good enough to form a shared ride. In case the returned set is non-empty, all clusters it contains are from now on considered to be good enough to be presented to users as possible ridesharing opportunities.

---

**Algorithm 3** Cluster postprocessing in K-Means method
 

---

**Input:** set of clusters (etalons)**Output:** set of possible ride offers

```

for all etalons do
  if etalon.assignedRequests.size < 2) then
    continue
  end if
  for all assignedRequests do
    distance  $\leftarrow$  computeDistance(etalon.startLocation, request.startLocation)
    distance  $\leftarrow$  distance + computeDistance(etalon.endLocation, request.endLocation)
    timeDistance  $\leftarrow$  abs(etalon.time - request.time)
    if distance > maxSpatialDistance  $\vee$  timeDistance > maxTimeDifference then
      assignedRequests.remove(currentRequest)
      etalon.startLocation  $\leftarrow$  mean value of start positions of assigned requests
      etalon.endLocation  $\leftarrow$  mean value of end positions of assigned requests
      etalon.time  $\leftarrow$  mean value of time positions of assigned
    end if
  end for
  if etalon.assignedRequests.size < 2 then
    continue
  else
    evaluationResult  $\leftarrow$  evaluate(etalon) {evaluation is described by Algorithm 5}
    if evaluationResult  $\neq$  nothing then
      ridePlan  $\leftarrow$  createRidePlan(evaluationResult)
      output.add(ridePlan)
    end if
  end if
end for
return output

```

---



---

**Algorithm 4** Cluster postprocessing in Directional method
 

---

**Input:** set of clusters (etalons)**Output:** set of possible ride offers

```

for all etalons do
  if etalon.assignedRequests.size < 2 then
    continue
  end if
  for all assignedRequests do
    timeDifference = abs(etalon.time - request.time)
    if timeDifference > maxTimeDifference then
      assignedRequests.remove(request)
      etalon.bearing = findMedianBearing(assignedRequests)
      continue
    end if
    crossBearing1 ← computeBearing(etalon.startLocation, request.endLocation)
    crossBearing2 ← computeBearing(request.startLocation, etalon.endLocation)
    if crossBearing1 > maxCrossBearing ∨ crossBearing2 > maxCrossBearing
    then
      assignedRequests.remove(currentRequest)
      etalon.bearing = findMedianBearing(assignedRequests)
    end if
  end for
  if etalon.assignedRequests.size < 2 then
    continue
  else
    evaluationResult ← evaluate(etalon) {evaluation is described by Algorithm 5}
    if evaluationResult ≠ nothing then
      ridePlan ← createRidePlan(evaluationResult)
      output.add(ridePlan)
    end if
  end if
end for
return output

```

---

---

**Algorithm 5** Evaluation of cluster - common to all algorithms

---

**Input:** cluster (etalon)

**Output:** true if this cluster forms good ridesharig option, false otherwise

```

output  $\leftarrow$  true
if etalon.assignedRequests.size < 2 then
    return false
end if
while repeat do
    repeat  $\leftarrow$  false
    prices  $\leftarrow$  computeEstimatedPrices(etalon.assignedRequests)
    for etalon.assignedRequests do
        if request.maxCost < prices.get(request) then
            assignedRequests.remove(request)
            repeat  $\leftarrow$  true
        end if
    if etalon.assignedRequests.size < 2 then
        output  $\leftarrow$  false
        repeat  $\leftarrow$  false
    end if
    end for
end while
return output

```

---

### 5.3 Ride plan creation

Even though clusters approved by the evaluation algorithm are good enough possibilities for shared rides, they can't be sent to involved users right away. The reason is they do not contain all of the necessary information that users need such as:

- Estimate of relative prices for each user
- Estimate of total length and price of the ride
- The order in which users should enter and leave the vehicle

This information is computed afterwards using starting points and endpoints of requests assigned to each cluster. At that point, finally, the cluster, now represented by its corresponding ride plan, can be offered to users.

### 5.4 Negotiation

The fact that users receive an offer for shared ride does not automatically imply they will all accept it. One could find many reasons for a user not to accept the offer - from occurrence of an unexpected event to simple change of his/her mind.

The system expects such situations and is capable of handling them. It does so by using simple negotiation protocol.

The negotiation itself begins with creation of a ride plan and happens in following steps:

1. Users involved in the ride represented by this ride plan are notified
2. Each notified user is supposed to either accept or decline given offer
3. With each user who refused to take part in proposed ride, the ride plan is recomputed
4. If the system concludes that the ride is no longer favourable, the whole ride is cancelled
5. If the system comes to a conclusion that the ride still may pay off, it adjusts the ride plan accordingly and continues to step 1

Only after being accepted by all participating users, the ride is considered to be definitive for the system, making it remove corresponding requests from a set on which clustering is performed. In case the ride was cancelled, no requests are removed from that set making it possible to match them with another ones again.

### 5.5 Data cleaning

There is one more situation in which a request can be removed from that set. It is when the request is no more relevant, which would typically happen when a request has not been chosen to be part of a shared ride for so long that the date and time on which this request is planned happens to be in the past. Taking such requests into an account during the clustering may (and probably would) have an adverse impact on the results of the clustering algorithm. Therefore server has to periodically check for requests that are no longer up-to-date and remove these from the set of clustered ones.

## 5.6 Detailed description

So far algorithms were described in general. In this section, more detailed description will be given covering all important parts of these algorithms. Also third way of matching requests will be introduced.

### 5.6.1 The distance

Probably the most significant distinction between both implementations is the way they compute the distance from one request to another.

**K-means algorithm** *K-means* version uses following formula to determine a spatiotemporal distance between two requests -  $x$  and  $y$ :

$$D(x, y) = \frac{d(x_{start}, y_{start}) + d(x_{end}, y_{end})}{2} + \omega \cdot t(x, y) \quad (5.1)$$

with the following meaning of individual symbols:

- $x_{start}, x_{end}$  and their corresponding  $y$  counterparts are starting point and endpoint of requests respectively
- $d(a, b)$  is a function returning real distance between given points in space in kilometers
- $\omega$  is a constant used to adapt temporal distance and make it comparable with the spatial one (this constant has a value of  $0.08\overline{33}$  which is a number of kilometers walked in one minute by a person walking at 5 kmph)
- $t(x, y)$  is a function computing time difference between given points in minutes

**Directional algorithm** *Directional* version computes the distance in much simpler manner: it just computes the difference in bearings of requests. Being a plain decimal numbers in range from zero to 360 degrees (non-inclusive), bearings can be easily compared by simple subtraction. The only complication is a possible underflow or overflow around values close to zero or 360.

### 5.6.2 Etalon adjustment

Each algorithm handles adjusting of etalons in its own way.

**K-Means** *K-Means algorithm* algorithm adjusts etalon by performing following steps:

- A position is adjusted by computing mean values of starting points and endpoints and assigning these to be new starting point and endpoint of the etalon respectively
- A time specification is adjusted by computing mean value of all requests (this step is performed using millisecond representation of Java `Dates` in which these are represented as `long` numbers)

**Directional algorithm** *Directional* algorithm adjusts etalons in following way:

- A bearing is adjusted by computing the median value of bearings of all assigned requests (while median bearing is computed in a classical way by ordering bearings and selecting the value right in the middle)
- A time specifications is adjusted the same way *K-Means* algorithms does it

### 5.6.3 Etalon postprocessing

Another aspect in which individual methods differ is a process I call “postprocessing” of etalons. This process takes places right after clustering is finished and its task is to make sure that clusters not making a good ridesharing option are discarded. This process was already described in general in 5.2, so this time I will only mention those specific passages that differ.

**Selecting representative specimen** While *K-Means* method uses computed etalon as a representative specimen to which assigned requests are compared, *Directional* algorithm uses the cluster simply as a container for requests that should belong together according to clustering process. One of the requests assigned to this etalon is selected to be the representative specimen and it is the one which has bearing most similar to mean bearing of the whole group of requests in that cluster (that means that firstly a mean bearing is computed for the whole cluster and then the request with bearing closest to that value is selected).

**Note:** In etalon postprocessing *Directional* algorithm uses **mean** value of bearings instead of median. This is caused by the fact that requests in one cluster are supposed to have very similar bearings making usage of mean value possible.

### 5.6.4 Online processing

Everything said about algorithms up to this point was meant about their “offline” versions. By “offline” I mean periodical, CRON<sup>1</sup> scheduled processing of requests. That ultimately means that a user wont get instant result from the server after adding new request. That is quite a drawback and potential opportunity for improvement.

While *K-Means* algorithm cannot be adjusted to support on-the-fly request processing just because of the way it works, *Directional* algorithm can. The result is what I call “online version of the Directional algorithm.”

---

<sup>1</sup>CRON is an acronym for “Command Run On” with the meaning of time-based job scheduler

**Online version of the Directional algorithm** The most significant difference between offline and online version of this algorithm is, that the online one does not perform any kind of clustering. It works as follows:

1. A requests is received
2. Its bearing is computed
3. All requests with similar enough bearing are withdrawn from the datastore
4. Result set is iterated and all requests which are too distant in time are removed
5. Postprocessing takes place further removing all requests that are not positioned similarly as the new one
6. If any requests remained, they create a good ridesharing opportunity and are returned in form of a rideplan

Once finished, this algorithms returns either possible shared ride opportunity for given request or it returns nothing in case no suitable requests were found that would form a ridesharing option with given one. Either way the new request is stored in the datastore - as an “active” one if no match exists or as “pending” one if a match was found.

**Note:** Selecting all requests having similar bearing and being too distant in time directly in from the datastore is currently impossible since *App Engine* does not support queries containing inequality conditions over more that one property. Probably because of its internal all-indexing implementation.

### 5.6.5 Cross-bearing difference

Although individual internal parameters are described later (7.2.1), one of them needs further explanation beforehand. I call this parameter “cross-bearing difference”. It is used by both version (offline and online) of the directional algorithm to determine whether or not is a request similar enough to specified reference request. It does so by computing bearings from startpoint of the reference to endpoint of the other request and vice versa (that’s where “cross-bearing” comes from). These bearings are then compared with bearing of the reference request. If any of these differences is greater than predefined value, requests are considered not to be similar enough.

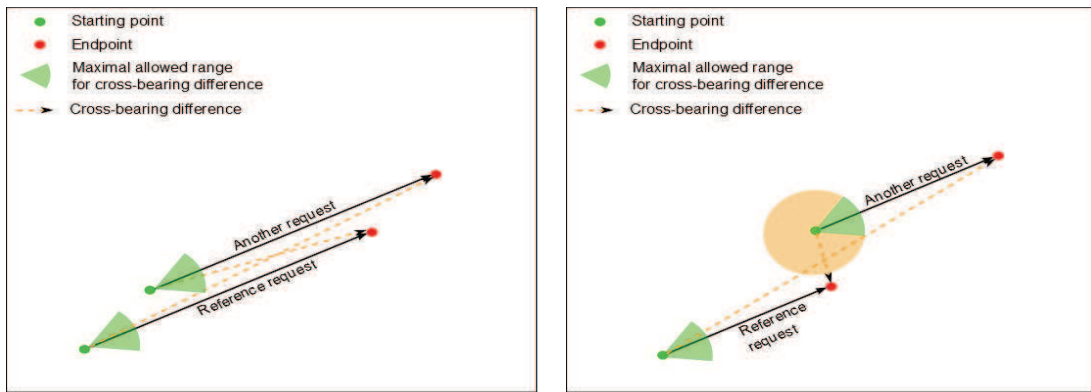
Figures below should completely clarify its meaning. Figure 5.1(a) shows two requests - reference request and another one - fairly similar to each other which would form quite good ridesharing opportunity, while figure 5.1(b) depicts two requests with very similar bearing, but with position not suitable for shared ride. With the value of “maximal cross-bearing difference” set right, similar requests are approved to create a shared ride while not-similar-enough ones are not.

**Algorithm 6** Matching requests using Directional online method**Input:** new request, maximal number of passengers**Output:** ride offer or nothing

```

bearingLowerBound  $\leftarrow$  newRequest.bearing  $-$  maxBearingDifference
bearingUpperBound  $\leftarrow$  newRequest.bearing  $+$  maxBearingDifference
requests  $\leftarrow$  all requests from the datastore having
bearing  $\in$   $\langle$  maxBearingLowerBound, maxBearingUpperBound  $\rangle$ 
for all requests do
    timeDifference = abs(etalon.time - request.time)
    if timeDifference  $>$  maxTimeDifference then
        requests.remove(request)
    end if
end for
if requests.size  $\geq$  maxPassengers then
    requests.sort {sort according to sum of distances between starting points and endpoints
    of requests and new request}
    requests  $\leftarrow$   $1 \dots \text{maxPassengers}$  from requests
end if
if requests.size = 0 then
    return nothing
end if
etalon  $\leftarrow$  new etalon {artificial etalon is created so that methods for offline algorithm can
be reused}
evaluationResult  $\leftarrow$  evaluate(etalon) {evaluation is described by Algorithm 5}
if evaluationResult = nothing then
    return nothing
else
    return createRidePlan(evaluationResult)
end if

```



(a) Cross-bearing difference of two similar requests (b) Cross-bearing difference of two non-similar requests

Figure 5.1: Graphical explanation of cross-bearing difference

## Chapter 6

# Implementation

### 6.1 Specific description

In this chapter I will describe specific features of the system once again. Why is that? Because since now, all the talk about how the system is designed, how it should work, how particular problems are solved etc. was more or less general. But this time I will provide an exact definition for specific parts of the system enlightening the way they really are implemented inside the system.

Needless to say I would select only few certain parts as providing a complete detailed specification would make this paper disproportionately long.

### 6.2 Server-side specialities

Many notable things come from using App Engine as underlying platform. Some of them - those that I consider being most important - are listed and examined below.

**Note:** App Engine runs applications in its own runtime environment which limits or restricts applications behaviour in certain ways.

#### 6.2.1 Single threading and responsiveness

Frankly I was surprised a little when I firstly learned that App Engine strictly prevents application from spawning threads. Then I thought it through (and read Googles explanation) and I realised that it makes sense. General philosophy of web-based applications is that these are meant to interact with users. As such, they should be as much responsible as possible, i.e. they should return a response in almost no time. Google forces this behavior by introducing two limitations:

- Application is prevented from running in more than one thread
- Application has to respond to a request in specified amount of time, otherwise its execution is terminated forcibly



**Note:** Despite not being able to create new threads, an application is allowed to perform standard operations against the only thread it's permitted to use.

**Note:** Time limitations for responding to a request are one minute and ten minutes for user-invoked and CRON-invoked requests respectively.

These limitations are a programmatic way of saying “your application is supposed to perform simple tasks (not requiring multiple threads) after receiving a request and it has to respond quickly enough not to make its user annoyed”

“But what can one do if one just needs an application to perform computationally extensive tasks?” you may ask. The answer is simple: not to make these computations while responding to a request. To serve needs of applications of such a kind, App Engine is equipped with a feature called “backends”. Backends are long-term, possibly both CPU- and memory-extensive tasks running separately from the request-serving part of the application. This service is billed and I will not provide any further information about it since I have no personal experience with it as there was no need to use it in my work.

## 6.2.2 The datastore

### 6.2.2.1 Description

App Engine allows its users to choose what kind of storage will their applications use. Options are two:

- High-replication datastore
- Master-slave datastore

It is obvious that both of them are designed to conform the needs of cloud-based service. According to Google, high-replication datastore uses so called “Praxos” algorithm for replication, while master-slave datastore utilizes traditional master-slave schema with one master to which inserts are made and which is replicated to several read-only slave mirrors. My application uses high-replication datastore.

### 6.2.2.2 Limitations

With the concept of highly replicated datastore comes one restriction: inserting into the datastore is limited to only one write per second. As far as I can tell it is because of internal representation of the datastore which is implemented using so called “big table” that relies heavily on indexing and, of course, because of the fact that the data has to be replicated.

### 6.2.2.3 Usage

App Engines datastore is unlike any traditional SQL-based database I have ever met (that's probably why a term “datastore” is used instead of “database”). The main difference is that there are no tables in the datastore. There is only the datastore. Whatever one needs to persist there, one just puts inside.

An elemental piece of information in the datastore is an entity. An entity is special object provided by App Engine API. An essential property of each entity is its *kind*. Entities with the same *kind* are considered to belong to the same *entity group* even if they have nothing more than kind in common. That is something completely unusual to someone with mainly SQL-based databases experience.

#### 6.2.2.4 Frameworks

**Objectify** As interesting as it may seem, working with raw App Engine entities is quite bothersome, so I implemented a framework called Objectify into my project which makes working with entities much more comfortable. Since Objectify demands that every class that will ever be converted into or from an entity is registered with the Objectify service beforehand, I have written a simple algorithm that scans for classes annotated by `@Entity` annotation using Reflection API and registers them with Objectify service automatically every time the server is started up.

**Restlet** App Engine is based on servlets. As such it handles well simple HTTP requests. I, however, needed to add support for (preferably RESTful) web services to my project. Since App Engine does natively support only SOAP web services and even that only partially, I decided to use third-party framework to add this functionality. This framework is called Restlet and allows the application to both receive and produce entities in various representations from well-known XML to some obscure ones. I chose XML as preferred representation for communication with outer world simply because its natural form is quite well human-readable which comes handy while testing.

That's pretty much it about implementation specific specialities when it comes to the server part of the system. Lets now take a look on what Android-related finesses one can encounter while developing an Android application.

### 6.3 Client-side specialities

Developing applications for Android is not quite the same as developing standard Java applications for computers. In the very beginning I encountered a paradigm shift reevaluating ones priorities of common programing practices. The most important things to keep in mind when programing mobile application are these:

- Resources are limited
- CPU time is expensive
- Applications lifecycle is not straightforward

The first and the second one of them are nothing new to anyone with just basic programing knowledge. The thing is that in mobile apps programing they gain a whole new dimension. Let me explain...

Even though no one possesses infinite resources, there is substantial difference when comparing resources of an ordinary computer and an ordinary mobile phone. Computers

these days have usually few gigabytes of operating memory for a program to fill up contrary to few hundreds of megabytes in average one can find in a mobile phone. Therefore the need of being as effective as possible becomes much more important on mobile platform.

Similar schema applies to processor power as well. Despite the fact mobile processors are currently experiencing massive boom and their development progresses rapidly, they still do not catch up with classical computer processors. On the other hand, computational power of mobile processors seems to be more than sufficient. That means that having “too weak” processor is not the reason for the CPU time to be that expensive on mobile phones. The reason lies somewhere else...close to resources. The thing is that unlike computer plugged into a socket, mobile phone is powered by a battery providing only limited amount of electrical energy which could be depleted quickly with wasteful CPU-usage policy.

Finally, there is one more Android-specific feature one needs to get familiar with - complicated application lifecycle. Unlike computer programs which have quite straightforward lifecycle - start, performing some work, termination - applications running on Android OS have their lifecycle much more intricate. Once started, Android application may be paused, suspended, woken up, notified, restarted or even terminated before it terminates naturally. This process of changing states comes from the need of allowing a user to have multitasking-like feeling while using Android device while additionally maintaining enough resources for the system and applications to run. For further information about how it all works I would recommend visiting Android developers site<sup>1</sup>.

This whole thing is fortunately managed by the Android OS (which does its job very well) thus all one needs to do is to adapt ones application to obey Androids instructions.

### 6.3.1 XML based resources

Rather unconventional property of Android is the way it works with resources. Instead of using precompiled classes representing individual objects, Android only stores XML definitions of these objects that can be then parsed and translated to Java object representation during applications runtime using special system service called “Layout inflater.” Of course layout inflater cannot be used to create objects that encapsulate any kind of application logic (these are precompiled and loaded using classloaders in typical way), but for simple UI elements consisting only of various basic Android UI components it is an interesting solution.

### 6.3.2 Painless threading

In contrast to computer programs that may not implement any kind of user interface, an Android app typically has to implement UI, therefore it must run at least one so called “UI thread”.

In Android, an application is started in its main thread called “the UI thread” which is responsible for processing users interactions with the application. A common situation is, that an application needs to perform a task that may take a while. Apparently, that has to be done in separate thread since blocking the UI thread even for a few seconds would make the application not responding to the user at all which is worst possible behaviour.

---

<sup>1</sup>Android developers - Activities: <http://developer.android.com/guide/topics/fundamentals/activities.html>

But with multithreading usually comes the issue of invoking actions on separate threads appropriately. An ordinary approach is to suspend the thread which needs results of an asynchronous operation until the other thread - which runs the operation - finishes its execution. However, this approach is not applicable since suspending the UI thread would result in application being not responsive which is what one needs to avoid. To solve this problem, Android provides an ingenious solution called “Asynchronous task”.

An asynchronous task is represented by **AsyncTask** class parametrized by three parameters. The most important method of this class is method named **doInBackground(...)** in which one places the logic that should be processed asynchronously. Once started by an **execute()** method, **AsyncTask** spawns a new thread in which **doInBackground(...)** method is executed. Once finished, it passes any results of its work as an input parameters of **onPostExecute(params)** method which is again run in the thread that invoked an execution of the asynchronous task.

This solution makes multithreading in Android very simple and developer-friendly. Asynchronous task are typically used every time an application shows some kind of progress dialog which runs in the UI thread while an operation is being executed in background by **AsyncTask**.

**Note:** My personal opinion after using multiple threads and UI in Android is that this particular practice is solved much better in Android than in Java's SWING library.

### 6.3.3 Using maps

One particular speciality of the Android system is its close bound to other Google services such as Google Maps. To enable an application to use Google maps, one needs to build the application for a special version of Android system called “Google APIs” and to obtain a key for using this API. Then the application may use specific Android elements, such as **MapView**.

**Note:** Using two different platforms from Google - the App Engine and Android - I came across the need of representing geological location in both of them. Interesting observation is that while App Engine uses class called **GeoPt** to store geological location in form of latitude and longitude represented by a pair of decimal numbers, Android uses class called **GeoPoint** that encapsulates latitude and longitude in microdegrees thus storing them as integer numbers making it necessary to convert from one to another while intercommunicating.

### 6.3.4 System-user cooperation

One more remarkable thing I have encountered while developing the Android application is a way the system cooperates with a user (by a user I mean an application developer this time). Nice example of this approach is in one of methods of an **ArrayAdapter**.

Array adapter in Android is used to populate a list of UI elements with user-specified information. The input of this method is not only an index specifying which element should be returned, but also already created UI element that is currently invisible to the one who holds the device, so it can be recycled and shown again with different data eliminating the need of creating brand new object since that could be resource-extensive.

With an insight on **ArrayAdapter** I close the chapter of implementation niceties and my personal experience with them.

## Chapter 7

# Evaluation

### 7.1 Experiments introduction

There was lot of discussion on how the application is designed, how it works, what it does etc... But what is really important is how it will behave in a real production environment. That's something nobody can tell for sure until the system is deployed and used by real people, but one can simulate this state artificially by testing.

#### 7.1.1 Metrics

Tests have two purposes: to measure important measurable parameters of the system (called metrics) and to simulate actual traffic that the system will need to handle once deployed.

One would definitely find many measurable parameters of this system, but for sake of simplicity and clarity, I decided to measure only those providing important information of the system, these are:

**Average time variation** For every ridesharing possibility found by the algorithm, an average time variation is computed as mean value of differences of time of the resulting ride from the desired time of the ride specified by each user. This information is quite important because it tells us how much will the starting time of the resulting ride differ from time specified by each user in average.

**Average detour** Along with average time variation, an average detour is computed for every ride. That is how much longer will the ride be because of the detours caused by the need of picking up users along the way. "The ideal ride" which is simply direct connection between starting point and endpoint of the ride is used as reference. Apparently, with this value growing up an effectiveness of the whole ride goes down.

**Average percent saving** Mean value of all savings of passengers involved in one ride. This value is especially important since it holds information about how much money will users of this system save in average. If this value would not be great enough, the whole system would prove itself useless.

**Average number of people per ride** Another very important reading is how many people will - in average - travel in one vehicle. More people in one car makes the ride more effective, but at the same time less likely to happen.

**Number of satisfied users** Efficiency of the system could be partially determined from this number. It tells how many of requests fed into the system will be satisfied. Obviously, the greater this number, the better the system. But note that in some cases this number could be quite low, but not meaning the system performs poorly, for example if there were many requests scattered on large area that would just not form good ridesharing possibilities.

## 7.2 Testing of the system

### 7.2.1 Input parameters description

The system has multiple parameters that can be changed. Every one of them may or may not influence its behavior in some way. These parameters are additionally divided into two groups: domain specific (those related to the problem this system is supposed to solve) and implementation specific (those that are internally used inside the system).

These parameters are listed below while the first and the second one are domain-specific and the rest is implementation-specific:

**Density of requests** Defines how “dense” are requests the system is working with. More dense requests means more requests being close to each other in space or time or both. Less dense requests means less requests on larger area with greater distances between them.

**Scenario** It is clear that requests coming into the system would not be always randomly distributed. Sometimes it would definitely happen that incoming requests would follow some kind of pattern (for example in specific daytime people are usually going from the center of a city to peripheries or the other way around). That may (or may not) affect results of the system.

The implementation specific parameters are again divided into groups according to an algorithm:

**Clustering algorithms** These parameters are used by clustering algorithms, i.e. *K-Means* method and “offline” version of *Directional* algorithm.

- **Maximal bearing cross-difference** - a value specifying the maximal difference of “cross-bearing” (see 5.6.5) and bearing of current representative specimen in degrees.

- **Maximal time variation** - specifies how far in time can individual requests be to be still considered as being “close enough”.
- **Maximal spatial distance** - used only by *K-Means* method to determine whether or not are two requests close enough in space.
- **Maximal number of passengers per ride** - specifies how many passengers can participate in one ride. This value is used to determine number of etalons in clustering algorithms.

**Online algorithm** This method works in completely different way than clustering algorithms so it has its own parameters which are following:

- **Maximal bearing difference** - specifies how much different (in terms of bearing) requests to fetch from the datastore.
- **Maximal time difference** - specifies how much different (in terms of time) requests to fetch from the datastore.
- **Maximal number of passengers per ride** - the same parameter as the one used in clustering algorithms, this time is used to limit maximal number of users for one ride.

These two types of algorithms had to be tested separately, since they work differently. Therefore there are two individual groups of tests in following section each covering testing of one type of algorithms.

### 7.3 Tests specification

Parameters listed in previous section had to be tested in different configurations. Since the number of possible configurations is enormous according to combinatorics, I relaxed the problem by introducing limitations to each of those parameters. I ended up with thousands of configurations which could be then tested in reasonable time. From these I obtained exactly the same amount of results. I won't of course list all of them, but just those depicting important or interesting relations between input values and output readings.

For every result of a test I will provide following information:

- Input configuration for the test
- Expected output the test
- Actual output of the test and its comparison with the expected one
- Graphical representation of important dependencies

**Note:** For testing I used pseudo-real data generated by simulation framework used by Petr Mezek in his work [7]. Three scenarios were generated each in three different sizes (300, 600 and 1200 requests) making nine data sets in total.

These scenarios will be referred to as shortcuts in consecutive section. Following table explains meanings of these shortcuts:

Scenario description	Shortcut
Random distribution of requests	S1
Request heading from the center to edges of the area	S2
Requests heading from marginal parts to the center of the area	S3
Average value over all scenarios	average

Table 7.1: Shortcuts for scenarios

## 7.4 Tests of clustering algorithms

These tests were performed in following way: the whole datastore was represented by appropriate dataset. That models a situation in which requests gather in the datastore during few hours. Internal parameters were adjusted according to current test configuration. Then both versions of offline algorithms were run multiple times on that data using those parameters to produce an output. On the output, all metrics were measured and averaged to reduce possible noise in the data.

Note that each of clustering algorithms uses its own internal parameters to do its job. Only common parameter is maximal allowed time difference. For this reason, following results of my experiments are separated for each algorithm. Every experiment will be introduced by an algorithm it was performed on.

### 7.4.1 [K-Means algorithm] Relations between maximal time difference, maximal spatial distance and number of created rides

Number of created rides is one of the most important readings we can obtain from the system. It contains an information about how many requests were matched together to form a ride. Since every ride is possible ridesharing opportunity, one would say that the greater this number, the better the system. That is not completely true, because this value itself doesn't tell us anything about how effective these rides are.

Surely, number of created rides will be influenced by changing input parameters of the algorithm. I expect that with rising maximal allowed time difference the number of created rides would go up, since this parameter is restrictive inside the algorithm. I also expect that with maximal allowed spatial distance going up, the number of created rides would go up as well as increasing it means loosening another restriction.

Configuration for this test was following:

Parameters	Values
Scenario	average
Density	600
Maximum of passengers per ride	2
Maximal time difference	variable
Maximal spatial distance	variable

Table 7.2: Configuration for *K-Means* test 1



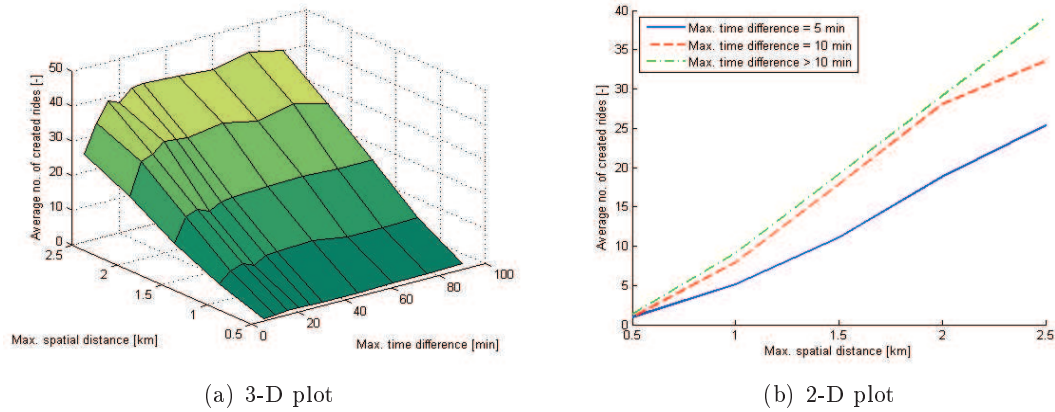


Figure 7.1: Relations between maximal time difference, maximal spatial distance and number of created rides

As expected, increasing any of input parameters results in higher number of produced rides. That is not surprising, because with higher allowed spatial distance the algorithm has more options from which a ride can be created. The same applies for maximal allowed time distance.

#### 7.4.2 [K-Means algorithm] Relations between maximal time difference, maximal spatial distance and ride efficiency

Previous experiment confirmed that increasing both internal parameters of *K-Means* algorithm results in higher number of produced rides. Now it's time to look at efficiency of these rides. I can tell how efficient each ride is from two measured value: average saving of individual passengers and average detour of the ride.

Logically I would suspect that increasing maximal allowed spatial distance between requests would result in less efficient ride. Decrease in ride efficiency should then lead to average detour going up and average saving going down at the same time.

Result was obtained for following configuration:

Parameters	Values
Scenario	average
Density	600
Maximum of passengers per ride	2
Maximal time difference	variable
Maximal spatial distance	variable

Table 7.3: Configuration for *K-Means* test 2

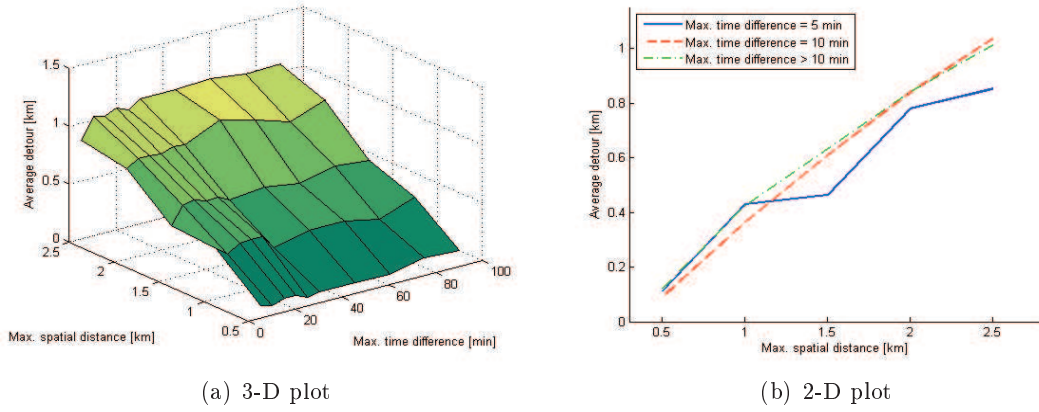


Figure 7.2: Relations between maximal time difference, maximal spatial distance and average detour

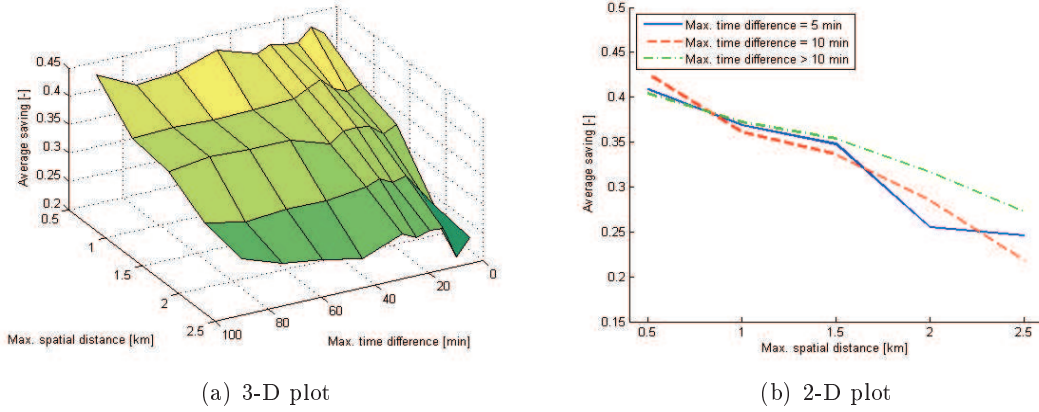


Figure 7.3: Relations between maximal time difference, maximal spatial distance and average saving

I think that results of this experiment are pretty clear. My hypothesis was confirmed and we can see (7.2(a) and 7.2(b)) that with increasing maximal allowed spatial distance, average detour rises regardless of maximal allowed time difference which was expected. What was also expected is that with increasing maximal spatial distance the average saving per user decreases (7.3(a) and 7.3(b)), again regardless of value of maximal allowed time difference.

It seems like maximal allowed time difference does not have any significant effect on efficiency of the ride. That is correct, it doesn't. What maximal allowed time difference does is that it permits requests being far away from each other in time to form a ride. But those requests are processed by clustering process beforehand in which their distance is computed from both spatial and temporal distance, therefore results of clustering should produce well-clustered groups of requests anyway. The only purpose of using maximal allowed time difference is to filter out outliers.

### 7.4.3 [K-Means algorithm] Relations between maximal time difference and average time deviation

Another important metric is average time deviation. This value represents the average difference between time of realisation of created ride and users defined time. The most interesting should be dependency of this metric on maximal allowed time difference.

From the way *K-Means* algorithm works I would expect this value not to grow much since it should be somehow corrected by the clustering process in which both spatial and temporal distance is taken into account.

Result was obtained for following configuration:

Parameters	Values
Scenario	average
Density	600
Maximum of passengers per ride	2
Maximal time difference	variable
Maximal spatial distance	variable

Table 7.4: Configuration for *K-Means* test 3

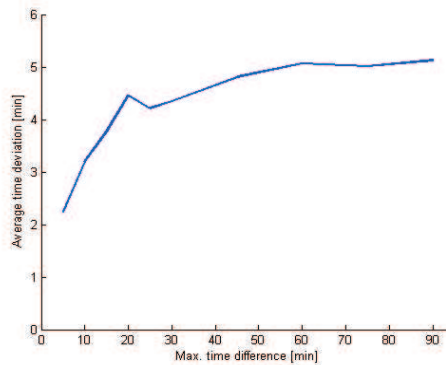


Figure 7.4: Relations between average time deviation and maximal allowed time difference

Indeed, average time deviation is not very sensitive to change of maximal allowed time difference. Although I expected it not vary enormously, the result is slightly surprising. It shows that using both temporal and spatial distance successfully eliminates extreme cases.

### 7.4.4 [Directional algorithm] Relations between maximal time difference, maximal cross-bearing difference and number of created rides

Previously, the behavior of *K-Means* algorithm was inspected. Lets now have a look at the *Directional* one. This algorithm uses slightly different input parameters. Most important one is what I call “cross-bearing difference”. For further information about this parameter and its role please consult [5.6.5](#).

I think that there is direct proportion between input parameters and number of rides produced by the algorithm. I expect that with rising maximal allowed time difference the number of created rides would go up. I also expect that maximal cross-bearing difference going up would lead to the number of created rides would go up as well. But I think that rides which would be allowed to exist due to this restriction being too loose would not be much favourable.

Configuration for this test was following:

Parameters	Values
Scenario	average
Density	1200
Maximum of passengers per ride	2
Maximal time difference	variable
Maximal cross-bearing difference	variable

Table 7.5: Configuration for *Directional* test 1

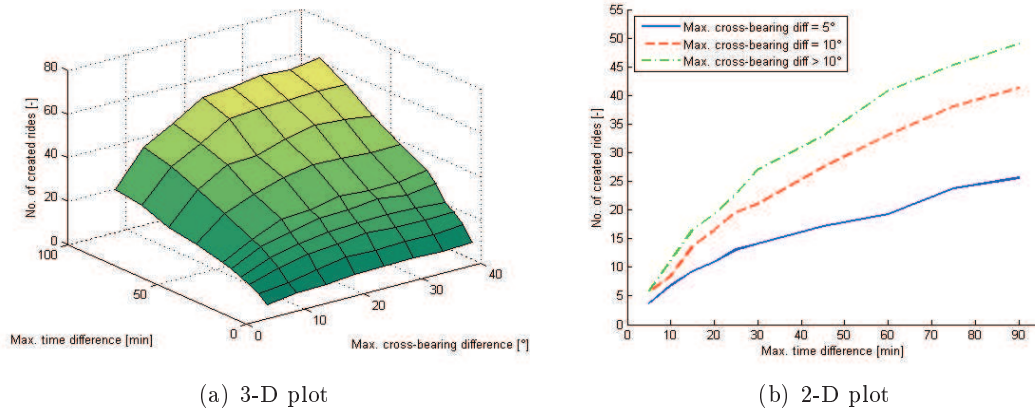


Figure 7.5: Relation between maximal time difference, maximal cross-bearing difference and number of created rides

Result of this experiment confirms my hypothesis. By looking at figure 7.5 we can see that there is a direct proportion between maximal time difference and number of created rides as well as between maximal cross-bearing difference and number of created rides. As I mentioned before, this effect is caused by loosening restrictions the algorithm uses which results in more rides being still good enough to make it to final stage. Although this doesn't mean that it's optimal to set these parameters as high as possible as there is another important factor not covered by this test: effectiveness of the ride.

### 7.4.5 [Directional algorithm] Relations between maximal time difference, maximal cross-bearing difference and effectiveness of the ride

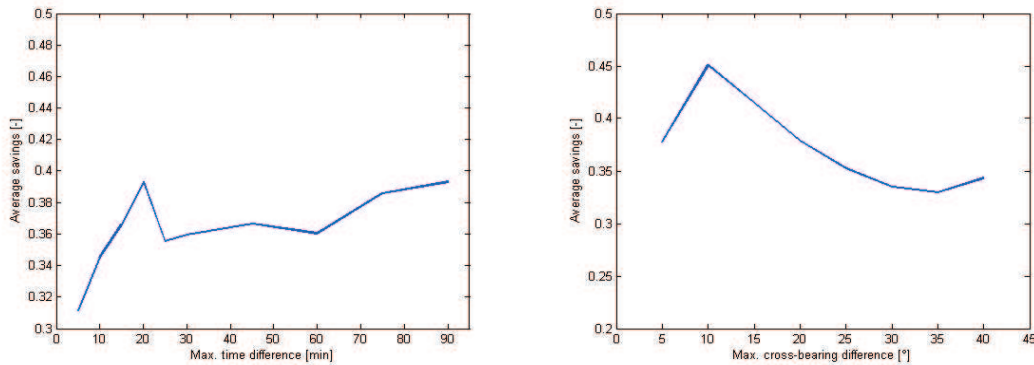
If one would try to make any conclusions solely from results of previous experiments, one could conclude that setting those two internal parameters of the algorithm as high as possible is a great idea since it results in higher number of created rides thus making the algorithm more effective. Unsurprisingly, it's not that simple.

Previous experiment doesn't take into account two important facts: firstly, peoples time is valuable and almost no one would take a ride scheduled long time before or after his desired time. That means that although setting maximal time difference to great value produces more results, it would be very rare if some of them would actually happen. And secondly, despite the fact that setting maximal cross-bearing difference to great value also results in more rides being created, there is still an important factor of passengers saving which is surely also influenced by changing that parameter. That leads to following hypothesis: average saving should go down with maximal cross-bearing difference going up. Additionally, great value of maximal time difference should not be as favourable as not-so-great one.

Following configuration was used to perform a test:

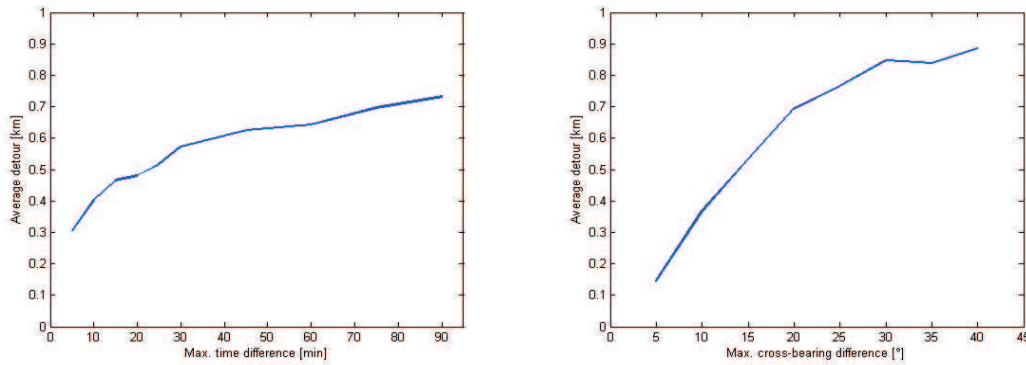
Parameters	Values
Scenario	average
Density	1200
Maximum of passengers per ride	2
Maximal time difference	variable
Maximal cross-bearing difference	variable

Table 7.6: Configuration for *Directional* test 2



(a) Relation between maximal time difference and average saving (b) Relation between maximal cross-bearing difference and average saving

Figure 7.6: Relations between maximal time difference, maximal cross-bearing difference and average saving of passengers



(a) Relation between maximal time difference and average detour (b) Relation between maximal cross-bearing difference and average detour

Figure 7.7: Relations between maximal time difference, maximal cross-bearing difference and average detour

This experiment resulted exactly as I expected: figure 7.6 clearly shows that there is indirect proportion between maximal cross-bearing difference and average saving of each user. That makes perfect sense since larger cross-bearing difference allows individual requests in every ride to differ greatly in bearing, which means that requests forming one ride may not be heading even similar direction, that makes the ride less efficient which results in less saving per user.

Figure 7.6(a) also shows an interesting fact: extending time window for rides doesn't cause average saving to drop, but it doesn't maximize it either. The peak in that plot indicates that optimal time window lays around value of 20 minutes. We can also see that average saving rises slowly with increasing maximal allowed time difference, but even if the maximal saving would be achieved by setting it to 90 minutes, one needs to keep in mind that real people would hardly accept such a great difference and computed saving would be never reached in practice.

When it comes to average detour, its relations to internal parameters are following: according to figure 7.7(a), average detour doesn't really depend on value of maximal time difference. That seems to be correct as there is no logical relation between those two values. What it depends on is maximal cross-bearing difference. As I expected, setting this parameter too high results in increase in average detour causing average saving to drop. From figures 7.7 and 7.7(b) it can be seen that optimal value of this parameter is about 10 degrees. That corresponds with what one would anticipate: too low value reduces average detour (making it almost zero) by forcing all requests in the ride to have almost the same direction, but that's not very likely to happen. Too large value, on the other side, allows too great variety of bearing of individual requests causing long detours which then result in lower average savings.

#### 7.4.6 [Directional algorithm] Relations between maximal time difference and average time deviation

Unlike *K-Means* algorithm, *Directional* method doesn't use both temporal and spatial difference at the same time to compute anything. Each value is used separately during the process of clustering, therefore I expect that with increasing maximal allowed time difference, the average time deviation would grow much more significantly, i.e. to dozens of minutes instead of just minutes.

Result was obtained for following configuration:

Parameters	Values
Scenario	average
Density	1200
Maximum of passengers per ride	2
Maximal time difference	variable
Maximal cross-bearing difference	variable

Table 7.7: Configuration for *Directional* test 2

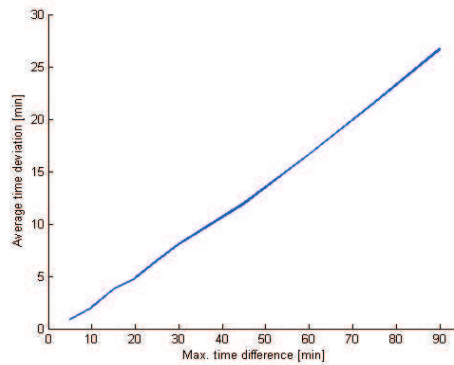


Figure 7.8: Relations between average time deviation and maximal allowed time difference

And it really does. According to figure 7.8 there is a direct proportion between maximal allowed time difference and average time deviation. It totally corresponds with the way this algorithm work: while clustering, it considers all requests to be “close enough” in time as long as their temporal distance from selected reference is lower than specified value - maximal allowed time difference. Figure 7.8 also shows that up to 20 minutes of maximal allowed time difference, the average time deviation is reasonably small (about five minutes), but with its further increase the average time deviation grows up to almost half an hour which is potentially unacceptable for users.

## 7.5 Tests of online algorithm

In this part I will present results of tests performed on online version of *Directional* algorithm. This method uses one more additional parameter - maximal bearing difference - because of which I had to generate separate testing configurations. I also needed to use slightly different testing methodology. Instead of representing the datastore by all requests from appropriate data set, I divided my data sets into two parts each in ration 4:1. Then I used the larger part to represent current state of the datastore and the rest as test set from which all requests were taken one by one and fed into the algorithm which either returned a ride or not.

All metrics measured on this algorithm are the same as those specified in preceding section. Unlike clustering methods this one is deterministic, therefore there was no need to run each configuration multiple times and take the result in average.

### 7.5.1 [Online algorithm] Relations between input parameters and number of created rides

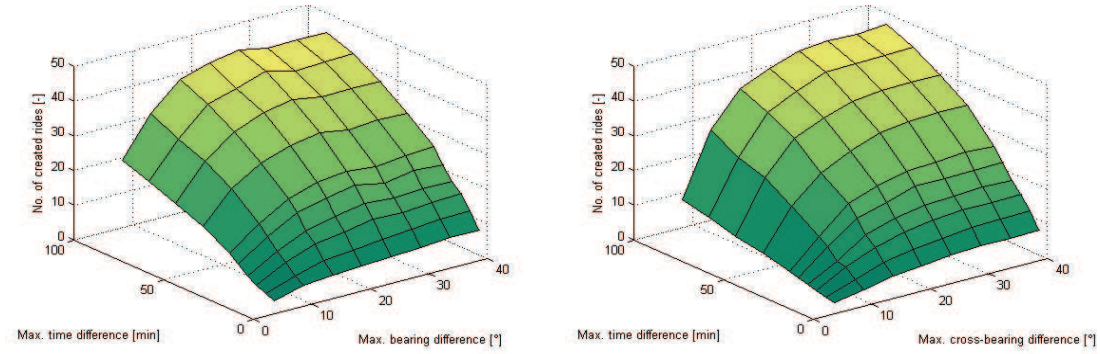
Once again there is a need for knowing how input parameters influence number of created rides. Unlike clustering algorithms tested earlier, this one doesn't work with complete set of requests currently located in the datastore, but it selects only small subset of them on which operations are performed. Size of that subset is directly affected by values of those parameters making it very important to set them correctly.

Result was obtained for following configuration:

Parameters	Values
Scenario	average
Density	600
Maximum of passengers per ride	3
Maximal time difference	variable
Maximal cross-bearing difference	variable
Maximal bearing difference	variable

Table 7.8: Configuration for *Directional online* test 1





(a) Maximal time difference, maximal bearing difference and number of created rides

(b) Maximal time difference, maximal cross-bearing difference and number of created rides

Figure 7.9: Relations between maximal time difference, maximal bearing difference and number of created rides

Figure 7.9(a) shows that number of created rides grows with both maximal time difference or maximal bearing difference going up. This is an expected result, because both parameters directly influence the number of requests being withdrawn from the datastore and the more requests there are for processing, the more rides are created. Effect of increasing maximal cross-bearing difference on number of created rides is exactly the same as for clustering algorithms.

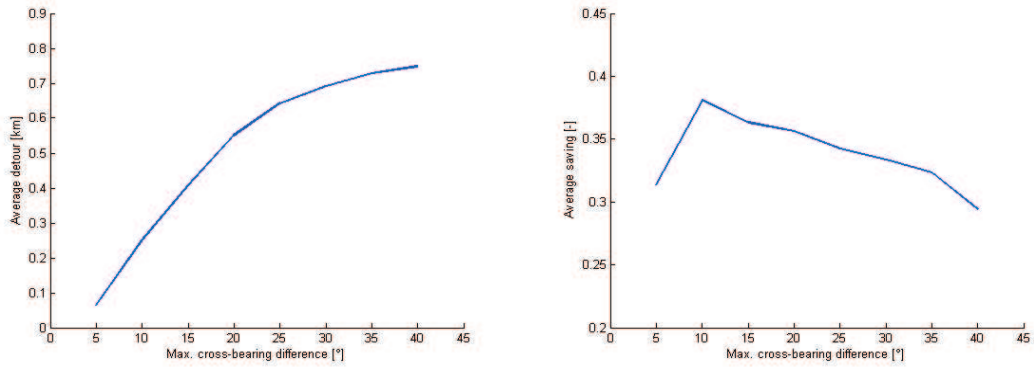
### 7.5.2 [Online algorithm] Relations between input parameters and efficiency of a ride

Once again it's time to evaluate effectiveness of created rides according to values of algorithms parameters. Expected result is the usual one: increasing max-cross-bearing difference should lead to decrease in average savings and increase in average detour. Also too great value of maximal time difference should mean worse efficiency of the ride.

Result was obtained for following configuration:

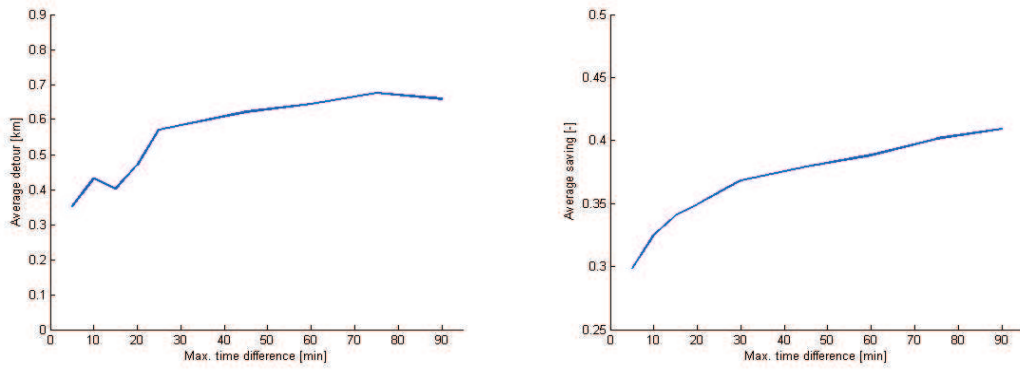
Parameters	Values
Scenario	average
Density	1200
Maximum of passengers per ride	3
Maximal time difference	variable
Maximal cross-bearing difference	variable
Maximal bearing difference	variable

Table 7.9: Configuration for *Directional online* test 2

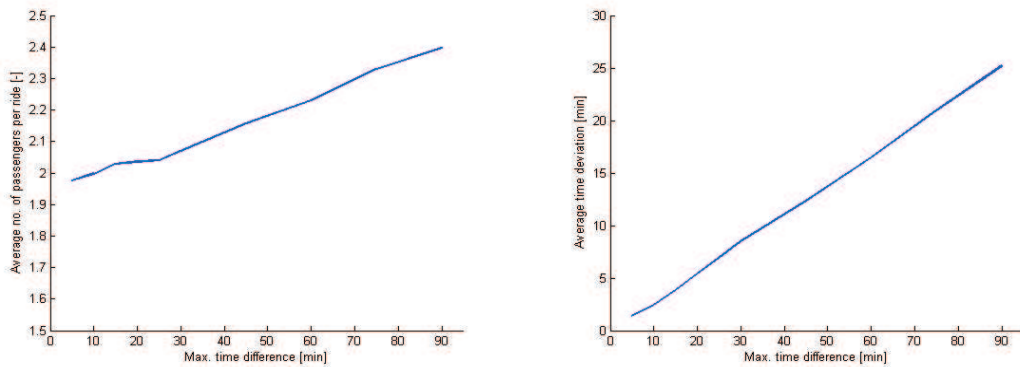


(a) Dependency of average detour on maximal cross-bearing difference (b) Dependency of average saving on maximal cross-bearing difference

Figure 7.10: Relations between maximal cross-bearing difference and efficiency of a ride



(a) Dependency of average detour on maximal time difference (b) Dependency of average saving on maximal time difference



(c) Dependency of average number of users per ride on maximal time difference (d) Dependency of average time deviation on maximal time difference

Figure 7.11: Relations between maximal time difference and efficiency of a ride

Result represented by figure 7.10 are almost self-explanatory. As usually, increasing maximal cross-bearing difference results in less efficient ride meaning longer detours and less savings.

The other four figures are more interesting, since they depict something new: figures 7.11(b) and 7.11(c) show that both average detour and average saving increase with value of maximal time difference going up. That is unprecedented observation which may look strange or even wrong, but I believe it is correct.

An explanation is shown in figure 7.11(c) depicting dependency of average number of passengers per ride on maximal time difference. Note that with increasing maximal allowed time difference the average number of passengers per ride also increases. This is something that never happened for clustering algorithms which produce almost exclusively rides consisting of two passengers only. On the other hand, online version of the *directional* algorithm produces rides of three users often enough to influence resulting savings and detours. Increasing number of passengers per ride then explains both increase in average detour (there are more passengers to pickup along the way increasing total length of the ride) and increase in average savings (detours caused by picking up third person are compensated by third user joining the ride which results in splitting the fare amongst three people instead of two).

Even though it may look so, setting maximal time difference too high is not optimal... as usually. This time it causes average time difference to grow (fig. 7.11(d)) to potentially unacceptable values.

## 7.6 Test results summary

I performed test on all types of algorithms implemented in the system. The most important notices were discussed on previous pages. Beside these I made some other interesting observations: clustering algorithms produce almost exclusively rides consisting of two users only while online method returns rides of three passengers on regular basis. Clustering algorithms are also slow. Despite the fact that underlying clustering methods (*k-means* and *k-medians*) both have polynomial complexity, time of computation grows significantly with number of requests (for 1200 requests it took about 90 seconds to complete computation). *Online* method is much faster though. It takes just few seconds to perform the calculation.

In conclusion, all algorithms works as expected producing reasonably good results. However, clustering methods seems to be too slow for larger amount of data. The online version of *Directional* algorithms is much more suitable for this kind of problem where response time matters.

The purpose of these test was not only to see how algorithms work, but also to find out what is the best configuration of parameters. By further analyzing collected data I concluded following values being optimal:

Parameter	Optimal value	Unit
Maximal time difference	22	minutes
Maximal spatial distance	1.5	kilometers
Maximal cross-bearing difference	12	degrees
Maximal bearing difference	15	degrees

Table 7.10: Optimal values of parameters

I also compared individual algorithms on both their efficiency represented simply as average percent saving over all configurations and their speed which I measured along with other metrics.

Algorithm	Average saving
K-Means	0.33
Directional offline	0.37
Directional online	0.41

Table 7.11: Comparison of average saving for individual algorithms

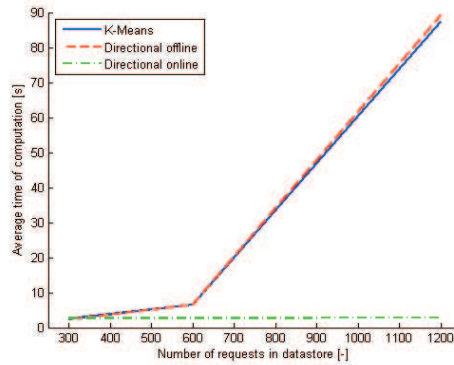


Figure 7.12: Comparison of speed of individual algorithms

By looking at table 7.10 and figure 7.12 one can see that the verdict is clear: online version of the *Directional* algorithm seems to be the best choice. Not only that its average saving over all configurations is highest amongst all implemented methods, but it also works with constant speed which is huge advantage over other two methods which wouldn't probably be able to handle very large data sets in reasonable amount of time.

## Chapter 8

# Conclusion

### 8.1 Recapitulation

The goal of this work was to develop a smartphone-based system capable of matching individual requests for taxi rides into one shared ride. The system was supposed to be build for Android OS using Google App Engine as server backend. Tests should have been performed to evaluate the system and to find out its optimal configuration. Expected benefits of this system were mainly monetary saving of its users and increased efficiency of taxis used.

### 8.2 Work summary

I have surveyed already existing solutions for shared taxi rides and concluded, that there definitely is a space for new system of such a kind. Then I familiarized myself with Android OS and Google App Engine development. I have also searched for algorithms suitable for spatio-temporal clustering and eventually came up with my own one. I have implemented both client-side and server-side parts of the system using multiple different implementations of ride matching algorithm. Then I have conducted a series of experiments with different test scenarios on the server-side part of the system to both evaluate its work and to find optimal settings for its input parameters.

All of the above was done for one purpose only: to create a system which could help people to use taxis more effectively. Evaluation of the whole system gave me quite a satisfactory result: the system is capable of creating shared rides. Two or three people are usually involved in one shared ride while every one of them pays about 40% less compared to price while going separately.

Another advantage of this system is that unlike some of already existing ones, it's not location dependent and as such, it should work virtually anywhere. That, together with making use of capabilities of today's smartphones, makes it very flexible, all the more so while running on cloud-based App Engine which provides highly scalable backend solution. And there is another difference between this application and most of already existing ones: my system fully supports shared rides with multiple starting points and multiple endpoints by which it almost completely eliminates a need for the passengers to gather somewhere before the ride begins.

In conclusion, the system seems to fulfil it's assignment by working as expected and producing shared rides based on individual requests. These rides are then cheaper for passengers and more economical than individual ones. This is especially important these days when prices of fuel are constantly rising and efficiency of each ride is more and more relevant. One day, possibly, people will use system like this (preferably this specific one) regularly to plan shared rides, thus saving their money and being more friendly to an environment.

# Bibliography

- [1] M. Charikar and B. Raghavachari. The finite capacity dial-a-ride problem. In *Foundations of Computer Science, 1998. Proceedings. 39th Annual Symposium on*, pages 458–467, nov 1998.
- [2] Jean-François Cordeau and Gilbert Laporte. The dial-a-ride problem: models and algorithms. *Annals of Operations Research*, 153:29–46, 2007. 10.1007/s10479-007-0170-8.
- [3] Roberto Cordone and Roberto Wolfler Calvo. A heuristic for the vehicle routing problem with time windows. *Journal of Heuristics*, 7:107–129, 2001. 10.1023/A:1011301019184.
- [4] Tony Grubestic and Elizabeth Mack. Spatio-temporal interaction of urban crime. *Journal of Quantitative Criminology*, 24:285–306, 2008. 10.1007/s10940-008-9047-5.
- [5] Slava Kisilevich, Florian Mansmann, Mirco Nanni, and Salvatore Rinzivillo. Spatio-temporal clustering. In Oded Maimon and Lior Rokach, editors, *Data Mining and Knowledge Discovery Handbook*, pages 855–874. Springer US, 2010. 10.1007/978-0-387-09823-444.
- [6] Quan Lu and Maged Dessouky. An exact algorithm for the multiple vehicle pickup and delivery problem. *Transportation Science*, 38(4):503–514, November 2004.
- [7] Petr Mezek. Agentní simulace taxi spolujízdy. 2012.
- [8] JL Santos, E Carrasco, AL Moore, F PÃ-Bravo, and C Albala. Incidence rate and spatio-temporal clustering of type 1 diabetes in Santiago, Chile, from 1997 to 1998. *Revista de SaÃPÃ*, 35:96 – 100, 02 2001.