Uppsala University
Department of Computer Science

Mobile Network Assisted Driving

ProjectCS 2015

Submitted in partial fulfillment of the requirements for the project CS course 2015 as requested by Uppsala University and Ericsson Research
January 2016

Abstract

The current bus system in Uppsala uses static timetables, generated every 4-6 months, to schedule bus routes. The goal of this project is to implement a system that generates daily dynamic timetables based on user requests, feedback, and other external factors such as weather conditions and special occasions like festivals.

The system monitors and logs searches made by the users and takes those searches into consideration when generating future timetables. Special days and holidays, along with weather conditions, are also given as input to the system in order to create an appropriate schedule for any given day. Furthermore, the application provides users with trip recommendations that match their habits and preferences; they can also give feedback in return to improve future suggestions. In addition, notifications are sent to the users to inform them of delays to their trips due to unforeseeable events, such as accidents.

Acknowledgements

- Supervisors

  We would like to express our gratitude to our supervisor, Edith Ngai, for her continuous guidance and support throughout the project. We would also like to thank the TAs, Stephan Brandauer and Amendra Shrestha, for their comments and helpful criticism.

- Ericsson Team

  A special thanks goes to Kostantinos Vandikas and Gábor Stikkel, who attended some of our scrum meetings and the presentation at Kista, for their feedback of the system as a whole.

- Team Members

  Mohammad Al Haj Ali, Eleftherios Anagnostopoulos, Chao Cai, Ilyass Garara, Charalampos Georgiadis, Nicholas Got, Christopher Hollmann, Tingwei Huang, Charalampos Gavriil Kominos, Jasmin Laroche, Malin Lindvall, Stavros Mavrakis, Daniel Llatas Spiers, Madhuri Pullambaku, Mohamed-Redha Redjimi, Jens Rosén, Yonatan Kebede Serbessa, Huijie Shen, Fatemeh Shirazi Nasab, Ziring Tawifque

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The following project stems from a desire to optimize the process by which schedules are generated in public transportation. As things currently stand, bus companies tend to have rather fixed timetables that are updated every 4-6 months, and static bus routes that are seldom updated, if ever. A current bus management system that has been used throughout this project as comparison is Upplands Lokaltrafik (UL). It uses its resources, which consist of 375 buses, to assist the entire population of Sweden's fourth largest city, Uppsala [1].

The aforementioned system works just fine and has for a long time. However it has a number of distinct limitations:

- It does not consider the needs of the users. The user is expected to adapt their needs according to the provided timetable and not vice versa.
- It has limited consideration for the bus load. In fact, even though bus companies deploy more buses during busy hours, there are times during the day where some buses are still bound to be full while others are empty and driven through the city for no reason at all.
- The bus routes are static, which means that many buses might pass by an empty bus stop several times during the day.

These three disadvantages lead to excess costs, in terms of fuel and working hours. This project attempts to address these issues by having a system which can dynamically allocate buses and

create timetables according to the demand of the users. The possibility of dynamically changing a bus route is also considered within this project in order to improve the cost cutting measures.

MoNAD, which stands for Mobile Network Assisted Driving, operates in a fairly different manner than the current system. As expected, the majority of systems nowadays have a mobile application available either on Apple store or Google market. Similarly, MoNAD comes with an Android application, where the users of the system are able to request a bus by searching for a bus journey with a desirable date, time, and location. However, these user requests are collected and processed daily by the back end of the system to generate new dynamic routes, flexible timetables, and personal trip recommendations. This is what makes MoNAD unlike any of the existing systems. In addition, an application is provided for bus drivers in order to display the route that should be followed by a specific line at a specific time of the day.

1.1 Statement of Originality

This is to certify that, to the best of our knowledge, the content of this document is our own work. This project has not been submitted for any degree or other purposes.

We certify that the intellectual content of this project is the product of our own work and that all the assistance received in preparing this project and sources have been acknowledged. MoNAD team

Chapter 2

Preliminaries

2.1 Concepts

Genetic Algorithms

Nature has inspired computer scientists over the years to come out with different approaches on solving complex problems. Evolutionary computation (EC) is a branch of computer sciences inspired by biological evolution. [26]. Biological evolution is based on Charles Darwins theory of natural selection [24]. According to this theory, given limited resources and stable populations, individuals compete against each other in order to survive. The survivors are considered to be the fittest individuals. This is because they were able to develop characteristics that let them achieve this purpose.

The following generations of individuals will inherit these characteristics. However, Darwin also stated that random events have an influence over the individuals. This influence will lead to the development of new characteristics. Thus, a population contains a set of desirable characteristics that help it to survive in a particular environment, but at the same time, individuals develop new ones when influenced by random events.

The implementation of all these concepts on EC is called evolutionary algorithm. An evolutionary algorithm usually consists of the following steps, which mirror some concepts from biological evolution:

- Chromosome: Encoding of solutions.
- Survival strength: Fitness function.
- Initial population: A set of semi random generated solutions.
- Selection and reproduction operators

One of the evolutionary algorithms is called genetic algorithms. Genetic algorithms (GA) has been widely used to solve optimization problems. GA is described as a parallel guided random search performed by a population of individuals. GA has been the main technology used on the implementation of the Look Ahead (LA) module. Thus, it has been found important to provide the reader with a clear explanation about this topic.

Chromosome: As it has been previously stated, a chromosome is an encoded solution. A chromosome is a data structure that is able to store several elements. For instance, an array is suitable for this purpose. Each of these elements is considered to be a dimension on the search space. The information encoded in a chromosome is also called genotype. A good way to design a genotype is to include few possible dimensions. The genotype - phenotype mapping will be discussed later.

Survival strength: Since GA aims to optimize a model, it is mandatory to have a fitness function. A fitness function will be the measure that will tell us how well the individual is as a solution for a problem. In terms of computation, a genotype is the input value of the fitness function. All the elements of the genotype will be used on the fitness function to generate a fitness value for that individual. However, the genotype is not the one that will be evaluated. This is where the genotype - phenotype mapping comes into scene. Usually, the minimal information encoded on the chromosome will be expanded to a larger set of information that the fitness function will be able to evaluate. Each individual is evaluated by means of a fitness function; moreover, the individuals fitness value will be used to decide if the individual is selected to remain on future generations or not [25].

Initial population: One of the most important features on GA is the parallel search. Parallelism is provided by means of a population of individuals. Each individual represents a possible solution to the problem. So, in order to start the search, individuals have to be semi randomly initialized. Another condition during this initialization stage is that individuals have

to be spread through the whole search space. This will reduce the chance of getting stuck on a local minima at the beginning (premature convergence). From a computational view, an implementation using threads or nodes is really suitable for this kind of algorithm.

Selection and reproduction operators: Selection operators aim to look for the fittest individuals giving a certain policy; while reproduction operators help to drive the search forward. Selection operators are closely related to the fitness value of the individuals. Usually, individuals are sorted by their fitness value. Some of them will continue to the next generation if its fitness value was good enough according to a certain policy. The rest of them will be eliminated. On the other hand, reproduction operators are represented by crossover and mutation. Crossover means that a couple of fittest individuals will exchange genes. When exchanging genes, an offspring consisting of two individuals will be created. This new pair of individuals is supposed to inherit the best characteristics from their parent (same behaviour that on the biological evolution). The assumption is that by performing crossover, the fitness value of the offspring is supposed to be improving. The general assumption is that the median fitness of a group is getting better in the long run. Crossover is operator that differentiates GA from random search. However, crossover only explores around an area that is already considered good. And again, the risk of getting stuck on a local minima appears. Thankfully, the mutation operator will prevent this. Mutation resembles the random events that affect individuals discussed earlier. Basically, mutation injects random genes into a chromosome. The purpose of doing this operation is to explore new areas on the search space. However, this operation has to be done carefully since it can destroy fit individuals.

On the previous paragraphs, a simple description of GA has been presented. Hopefully, the reader has now a better understanding about this algorithm. It is one of the main concepts using on the development of MoNAD project, especially on the LA module. Further technical about the actual LA implementation are described later on the report.

2.2 Tools & Platforms used

A summarized list of platforms and tools is presented in this section. The term platforms refers to operating systems and programming languages used during the development and the testing process. Tools, on the other hand, describe specific libraries and IDEs used for each module.

Module	Library	IDE / Text editor
Travel Planner	PyMongo	Gedit
Route Generator	Open Street Maps 0.6, MochiWeb, ErlPort	IntelliJ Idea for Python, PyCharm, VIM editor
Request Handler	PyMongo, Gevent, Gunicorn, Nginx	Gedit
Look Ahead	DEAP, Scoop, PyMongo	Sublime Text 2, PyCharm, VIM editor
Route Administrator		
Authentication Module	MochiWeb, Emysql, PyMongo, ErlPort	
Notification System	Google Cloud Messaging	Android Studio
Travel Recommendation	Spark 1.4.1, PySpark, PyMongo, GeoPy 1.11.0	Sublime Text 2, Atom

Table 2.1: Libraries used during development

Platforms

One of Ericsson Research's main requirements was to build and open source platform. This criterium was fundamental in choosing operating systems and programming languages to use for the project. The chosen operating system was Ubuntu 14.04.3 LTS. The Android applications are written in Java 1.8. Languages used in the back end are Python 2.7.6 and Erlang OTP 18. Additionally, Gama version 1.6.1 was the simulator chosen for the project. Finally, the databases selected were MongoDB 3.0.7 and MySQL 5.6.

Tools

Different libraries and IDEs were used for the front end and back end. Android API 21 was the chosen library and Android Studio 1.4+ was used by the application implementation group for the front end. The back end has a variety of libraries and IDEs due to the different requirements of each module. The details of the back end modules are shown in table 2.1.

2.3 Existing similar platforms

In this section, systems which are similar to MoNAD are presented. Ride-Sharing systems for both cars and busses exist and in this part, two examples of the most popular deployments in this area are presented: Uber and Kutsuplus. Furthermore, the SL train system is also mentioned, especially their innovative way for dealing with train delays.

UBER

Uber is a mobile application which enables users to request drivers in order to be transported and it is similar to the taxi system [1]. In order to request a taxi however,you usually have to request one from a call center. But with the Uber mobile application, everything becomes easier and faster. You request for a car, then keep track of the cars GPS location on a map until the car comes to pick you up. The application gives you a fare estimate for the ride, so you dont discover the price at the end of the trip. Finally you pay easily through the app so the need for cash is eliminated and Uber gives you the possibility to give feedback after a ride. The app gets your GPS location and look around for drivers. They keep track of the driver partners location in order to keep the maximum waiting time at 5 minutes. To solve the problem of supply and demand Uber applies surge price protocol. That means that when the demand is high and the number of driver partners is lower, the fare price is raised to encourage driver partners and discourage riders from taking a car (if they can take other means of transport). This surge in price is done in real-time depending on the requests. For example you can have a price surge for 10 minutes and then the price comes back to the standard value.

KUTSUPLUS

Kutsuplus is a bus service transport system deployed at Helsinki, in Finland [2] [3] [4]. The concept has been developed by Ajelo and is quite similar to Ubers but instead of getting a cab on-demand you get a bus on-demand. (Ajelo is the startup that has developed the algorithm and was acquired in November 2014 by Split to deploy a shared ride service in the United States).

[1] http://www.uber.com
[2] https://kutsuplus.fi/help
[3] http://www.ajelo.com/
[4] http://split.us/

You request for a bus on a mobile application choosing the starting point, the destination, the departure time and the number of seats you want to book because each bus contains 9 seats. The Kutsuplus will then look for other people requesting a bus to the same destination and will propose to you two possible trips. Once the trip is confirmed you can pay directly through the mobile app with a virtual wallet. The price is not fixed as in regular bus transit. For each trip proposition after a request, the fare is indicated. The fare can be split and get cheaper if you book a bus for a group. The system generates routes following regular fixed bus stops, but they intend to extend the fixed bus stops with virtual bus stops which might be for example popular destinations. The price for a seat varies but is more expensive than the price of standard bus transit and cheaper than Uber cab booking.

SL

Storstockholms Lokaltrafik (SL) is the swedish organization in charge of running all the land based public transport systems in the county of Stockholm. SL came up with a nice innovation that allows them to find a way to predict train delay two hours in advance. This is done thanks to The Commuter Prognosis algorithm they have developed with data scientist Wilhelm Landerholm. [5] [6] The concept consists in gathering data from the train network such as small delays of 2 minutes and processing this big data to get forecasts for train lateness. The reason behind it is that a small delay at a particular point of the train network can affect the whole network in the future. Once the system has predicted delays that have not happened yet, it automatically send this information to the SL traffic operators. This makes it possible for them to make the necessary adjustments to prevent or at least reduce the impact of the predicted delays. Also, the system informs the commuters about the predicted delays two hours in advance which let them find alternatives and rearrange their schedule for another mean of transportation. Finally and not the least, the algorithm can be adapted to be used for other types of public transportation and/or in other cities in the future.

[5] http://www.springwise.com/algorithm-predicts-prevents-train-delays-two-hours-advance/

[6] http://www.netimperative.com/2015/09/train-operator-uses-big-data-to-avoid-delays-that-havent-happened-yet/

Chapter 3

MoNAD

Two MoNAD mobile applications are used in order to communicate information between application users and the company. The Client and Vehicle Applications are designed for the passenger and driver, respectively. Usage of these two applications is very important in order to reach MoNADs goal of improving bus timetables, which will optimize routes and reduce the passengers waiting time.

The Client Application presents information to the user, including but not limited to: personalized trip recommendations, a bus search function, and notifications. The Client Application also collects information from the user, such as requests the user makes, in order to improve timetables, optimize routes, reduce passengers waiting time, and recommend interesting routes.

The Vehicle Application also presents information to the driver, including: information about routes, the maps to follow, and the number of passengers expected to board and leave at each stop. Additionally, the Vehicle Application collects information from the driver, such as alerts about accidents which have affected the route.

3.1 Search and Trip Information

The main feature in the Client Application is the ability to look for the best option to reach a destination. The Search feature is presented in two variants: The Quick Search bar and the Advanced Search screen. Both alternatives offer a list of address and bus stop suggestions to

the user when entering the origin and destination; the bus stops are capitalized, whereas the addresses remain in lowercase.

3.1.1 Quick Search

Quick Search allows users to look for the optimal way to reach a destination starting from their current position. Since it is intended to save time for the user, Quick Search is accessible from the main screen of the application, and disregards all the other criteria that the advanced search provides as well. The system saves the users location and uses it along with the entered destination to search for the appropriate bus line. Other preferences such as date and time are assigned default values: today, now, and earliest bus. The returned results are first sorted by relevance, then by the starting time of the trip, as the earlier ones top the list.

3.1.2 Advanced Search

The Advanced Search, on the other hand, gives the user full flexibility and control over the information that they would like to specify. First, the user has two options when selecting the starting point. Either the current position can be selected by pressing the pinpoint button or the user can begin to type the name of the bus stop and a list of suggestions will become visible. The destination can only be selected by typing and selecting a suggested bus stop. Next, the user has to enter the time and date of the trip. This can be done for either the departure or arrival time, which is ultimately the users decision as well. Finally, the user can define the more suitable priority in order to get a better sorting of the results. The two available options are earliest trip and shortest trip, where the earliest trip will look for the trip with the earliest departure or arrival time and the shortest trip will give more importance to the trip with the smallest duration.

3.1.3 Returned Results

Once the user proceeds with a search, regardless of the type, a list of potential trips is displayed in the advanced search screen. Each result contains information about the two bus stops where the user will board and leave the bus, as well as their respective departure and arrival

times, followed by the total duration of the trip. In order to improve the user experience, the application will display a small icon when the bus is close to reaching the departure bus stop, hence advising the user to hurry to the bus stop if needed.

When the user browses the results and finds a suitable trip, it can be selected by clicking on the arrow button on the right side. The application navigates to a new screen that provides the user with all the information available about that trip. In addition to the departure and arrival information that is displayed in the search results, this screen shows the buses that the user has to take and a full list of the stops in each bus schedule. The user can express interest in the trip simply by clicking the button "Join Trip" at the bottom of the screen and confirm their decision when the confirmation dialog appears.

3.2 Travel Recommendations

One of the components that this application provides in order to improve the user experience is the Travel Recommendations feature. Based on the idea that users tend to follow certain habits when requesting a trip, their request information (i.e. departure and arrival location and time) is collected and stored in the system before it is periodically processed. As a result, requests that have similar characteristics represent a user's habit. The system then tries to match the user's needs for this habit against the current timetable, so as to give back certain recommendations.

The goal of this module is to learn from every user's past behavior, then be able to distinct whether some trips are going to be of particular interest for him/her before giving back suggestions. As soon as the user logs in, the application displays the list of recommendations where it is possible to select and join a suggested trip without the need to search for it.

In a concrete scenario, an employee who lives in Flogsta and his workplace is close to the Central Station is expected to request a bus every morning around 8:00 am. After a certain number of requests for a given period of time, the application recognizes that this is a habit and can proactively list a number of trips that will cover this user's need, or even explicitly notify the user early enough in order not to miss the bus. This way, the application saves time for the user from configuring a search and browsing the timetable, and in the end offers a more

personalized approach to the user.

3.3 Account Management

The user has very much flexibility with the MoNAD Client Application, seen on the Login page and the Settings page. The flexibility is available in order to provide convenience, hoping to encourage the user to continue using the application without any complications.

3.3.1 Login

The first page that the user sees is the Login screen, which gives the user the option to sign in with a username and password combination or a Google account. The Google login is very convenient since most people are comfortable with using a Google account to log in to various applications, such as LinkedIn [1], Airbnb [2], and Runkeeper [3]. If the user has never signed into the application, clicking the Google Login button will allow the user to skip the registration steps because the Google account automatically provides the pertinent information.

If the user signs in with username and password credentials, the Profile page makes it possible to modify the password and the profile information while signed in. If the user has forgotten the password, it is possible for them to retrieve it by pressing the "Forgot your password?" text on the Login page. Additionally, this page provides the user with the option to create a new account if it has not already been done. If the user selects "Create New Account", a new page appears on the screen asking for personal information such as the desired username, password, email address, and phone number. After the information has been input, the user presses "Register" to continue with the application.

When the user no longer wants to use the application, a "Log Out" option can be found in the application menu, which can be accessed from the top-right of the screen. When the user selects "Log Out", the user's application information will no longer be accessible, and the application will navigate back to the Login page.

[1] https://play.google.com/store/apps/details?id=com.linkedin.android
[2] https://play.google.com/store/apps/details?id=com.airbnb.android
[3] https://play.google.com/store/apps/details?id=com.fitnesskeeper.runkeeper.pro

3.3.2 Settings

The MoNAD Client Application allows the user to change options on the Settings page. There are three types of settings: Language, Alerts, and Theme. Selecting the Language tab allows the user to select the language that they prefer reading when using the application. By swiping over to the Alerts tab, the user can decide whether they would like to see personalized suggestions about routes that may be of interest to the user by turning Travel Recommendations "on" or "off". The same can be done for Reminders about trips, as they may or may not want to receive notifications about the upcoming trips that the user decided to join. All changes are saved when the user leaves the Settings page.

3.4 Vehicle Application

The Vehicle Application is the second component of the MoNAD system. While the Client Application is intended for customers to search and join trips, the main objective behind the Vehicle Application is to serve as an assistant during a bus journey.

3.4.1 A Safe Interface

Significant efforts were dedicated towards the interface design. In fact, the application was designed in a way that minimizes its level of interaction with the bus driver, thus ensuring that the primary focus is on driving. This was achieved by making all of the displayed information accessible in at most one click once the driver is signed into the application and starts the journey. Additionally, the information is loaded at once during the sign-in and regrouped in a single page in order to eliminate the hassle caused by page navigation and information loading.

3.4.2 Journey Map & Information

The only page of the vehicle application consists of a simple map of the city and a board that displays some complementary information about the trip (see Figure 3.1). The map contains the trajectory of the journey as well as icons that specify the bus stops along the way. Since

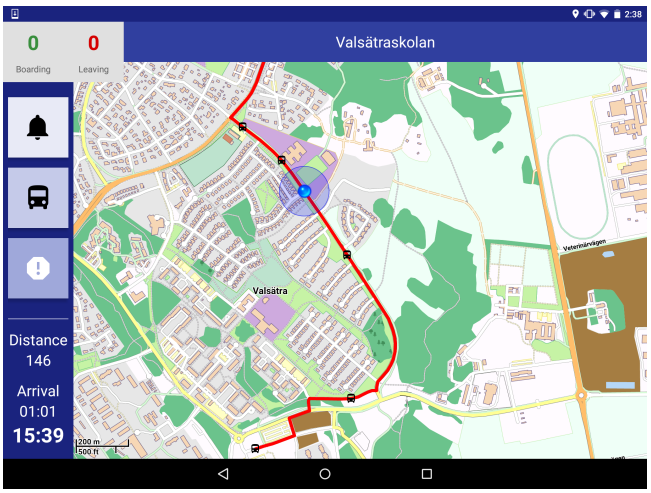

Figure 3.1: A screenshot of the main page in the Vehicle Application

the application tracks the position of the tablet, the bus position is updated in real time on the map. This, in turn, allows the application to update the distance and time left to reach the next bus stop in real time, which is displayed similarly to a GPS navigation system. In addition, whenever the bus reaches a bus stop, the application updates the name of the next destination (i.e. bus stop), the estimated number of passengers boarding or leaving, and the estimated time of arrival.

The complete schedule of the journey can be accessed from the button with a bus icon on the side bar, which is shown as a list of bus stop names with their respective expected arrival time. The two other buttons on the side bar are used to communicate with the system; one button that shows a list of notifications and updates, and a button that allows the driver to quickly send an emergency notification. The emergency system is simple and follows the same aforementioned safety principle, as it gives the driver the choice between four types of emergencies and, if necessary, add a short description of the problem.

3.4.3 Real-Time Traffic Information

Nowadays, there are many different APIs that are available on the internet to get real-time traffic information such as construction, congestion, accident and so on. All these kinds of information could be helpful in the application to avoid wasting travel time and make a better bus schedule. The purpose of using real-time traffic information is to provide more precise estimated driving time (EDT) for the bus drivers. For example, if the bus goes through a

heavy traffic congestion road, the EDT provided by OpenStreetMap will be invalid.

3.4.4 Account Management

Similarly to the Client Application, access to the Vehicle Application is granted using credentials in the Login page. The credentials consist of the driver's ID in the bus company and a password. However, since a bus company has hundreds of buses and each bus travels several journeys every day, the bus number has to be specified before signing in as well. If the entered ID and password are correct, the system can find and display the appropriate journey that matches the specified bus number, date, and time.

Chapter 4

System Description

The following section presents the technical side of the MoNAD project. It lists the functional and non-functional requirements, and includes a detailed description of the system design. In addition, it describes the interaction between the modules of the system and the various technologies used for each of them.

4.1 Requirements

Requirements in the MoNAD project are extracted from the specification document that the product owner provided during the start of the project. Additional ideas discussed internally and feedback by the product owner have also resulted in eliciting many more. The requirements are divided into functional and non-functional based on the definition [31].

4.1.1 Functional Requirements

The functions that MoNAD system should have are grouped in several tables. Table 4.1 displays the functional requirements for the Client Application (CA) and Table 4.2 displays the same for the Vehicle Application (VA). The functions of the back-end modules are listed in Table 4.3.

No.	Function	Description
1	Register User	The CA allows the user to register
2	Log In	The CA allows the user to sign in to the application using credentials or a Google account
3	Make Request	The CA allows the user to make travel requests
4	Search Trips	The CA allows the user to search for a desired trip
5	Show Trip Recommendations	The system provides personal recommendations based on past user requests and displays them in the CA
6	Receive Notifications	The CA receives notifications from the Notification Module for trip information and recommendations to alert the user
7	Record Locations	The CA records the locations of the user on a regular basis after getting his/her approval
8	Update Profile	The CA allows the user to change personal profile information (e.g. username, password, etc.)
9	Update Settings	The CA allows the user to select their preferred settings (language and alerts) of the application
10	Sign Out	The CA allows the user to sign out

Table 4.1: Functional requirements for the Client Application

No.	Function	Description
1	Log In	The VA allows the driver to sign in from the login screen
2	Show Route	The VA displays the route received from the database on a map according to the specified bus line
3	Show Bus Stops	The VA lists the bus stop names for a route
4	Receive Notifications	The VA receives and shows notifications from the system
5	Record Emergencies	The system records emergency types to the database from the VA and notifies the user at certain times

Table 4.2: Functional requirements for the Vehicle Application

No.	Function	Module
1	The system receives user requests	Request Handler
2	The system matches a search request to the appropriate trips	Travel Planner
3	The system generates personal trip recommendations for every user	Travel Recommendations
4	The system generates new timetables based on past requests	Look Ahead
5	The system sends notifications to the Client and Vehicle Applications	Notification System
6	The system allows the user to give feedback about the trip	Feedback System
7	The system receives and processes geographic information for use in other modules	Route Generator

Table 4.3: Functional requirements for other modules

4.1.2 Non-Functional Requirements

- *Scalability:* The different modules used in MoNAD shall scale accordingly when the number of requests grow.
- *Security:* The system, especially the back end, shall be secure and ready to stop any malicious attacks. Passwords shall be encrypted for both Android applications.
- *Privacy:* The system shall only use location-based features if the user gives explicit consent to access their location information. Personal and behavior information such as travel requests shall not be disclosed and shall be used for the sole purpose of generating better timetables and recommendations.
- *Usability:* The Client and Vehicle Applications shall have simple and user-friendly interfaces.
- *Availability:* The system as a whole shall be operational and not experience any down time.
- *Performance:* The system back end shall receive, process, and deliver data in timely fashion without any timeouts.

4.2 Design

4.2.1 System Architecture

Architecture Overview

System Architecture describes the overall structure and high-level abstraction of the system and top-level interaction between different components. Since an Android application is used in this project, the architectural pattern that was followed was the MVC (Model View Controller) model. In MoNAD, requests are made through the Client Application and the response is returned from the Request Handler.

Main Components

The main modules of the system are listed as follows:

1. **Client Application:** An Android application that serves as the interface to the system and provides the user with functionality such as registering, logging in, searching for available buses, booking a trip, etc.

2. **Simulated Client Application:** A simulator that imitates the Client Application's functionality and is useful for testing and simulating large numbers of requests.

3. **Vehicle Application:** An Android application that is designed to navigate the bus drivers and send them information, such as the number of passengers leaving and boarding at each stop.

4. **System Database:** All data related to the system is stored in the System Database. This does not include user data, such as their profile information. Information which this component stores include user requests, user trip history, reservation information, and recommendations generated for the users.

5. **Request Handler:** The Request Handler is responsible for receiving user requests and communicating with other modules, such as Travel Planner.

6. **Travel Planner:** This module generates optimal traveling plans based on a user request.

7. **Notification System:** Proactively sends different kinds of notifications to the Client Application and Vehicle Application.

8. **Authentication Module:** The Authentication Module uses the User Database in order to authorize and authenticate the user's access to the system. In general, it is responsible for making a secure connection between users and system.

9. **User Database:** The User Database stores the user's profile information.

10. **Travel Recommendation** The Travel Recommendation module provides passengers their potential destinations based on their history data.

11. **Look Ahead:** Responsible for generating new timetables based on user requests.

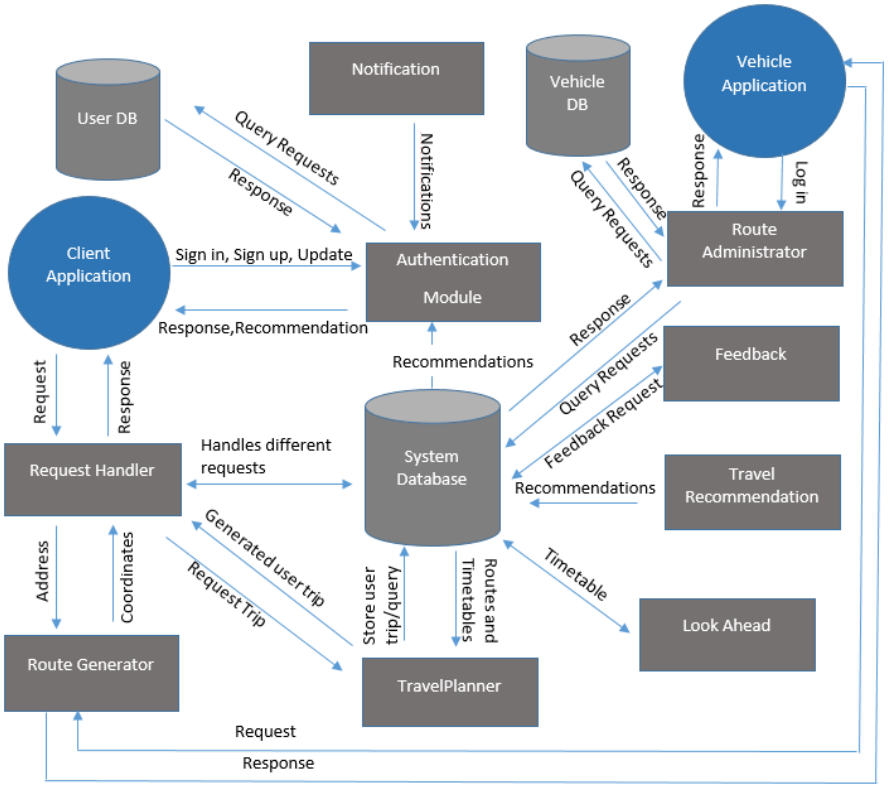

Figure 4.1: A component diagram which illustrates the system architecture

12. **Feedback Module:** Responsible for getting the feedback from users regarding the quality of the user's trip.

Component Diagram

The following diagram illustrates the general interaction between the main modules of the system 4.1.

4.2.2 Sequence Diagrams

The purpose of this section is to illustrate the interaction between different modules involved in the MoNAD application system, by providing a set of sequence diagrams.

Client Application

- **Case 1: User Authentication**

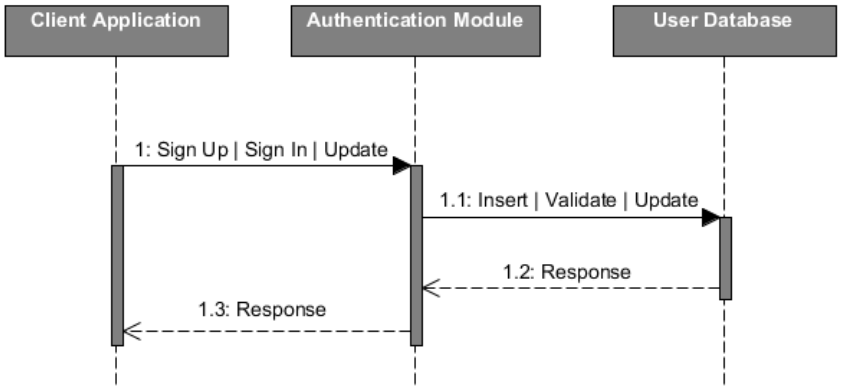

Figure 4.2: A sequence diagram which shows the interaction between modules when the user starts using the application

- **New User:** A new account should be registered to the **User Database**, containing information regarding username, password, email, and phone number. Moreover, a Google registration token is stored, corresponding to the mobile device of the user, allowing the application to receive notifications using the **Google Cloud Messaging (GCM)** service.
- **Registered User:** If the user has already registered, the user can log into the system by providing credentials (i.e., username and password) which will be compared to the existing entries in the database. After the login operation, it is also possible to update personal information.
- **Google Account:** Users have also the option to use a Google Account for logging in. Figure 4.2 shows this process in a sequential diagram.

- **Case 2: Travel Recommendations**

  Recommendations are an important feature of MoNAD. Users can browse their recommendations at the main screen of the application and make direct bookings.

  - **Travel Recommendation:** Generates new recommendations once per day and stores them to the System Database.
  - **Client Application:** Communicates with the **Authentication Module** in order to receive daily recommendations. See figure 4.3.

- **Case 3: Travel Requests**

  MoNAD offers the possibility of making travel requests. Users provide information re-

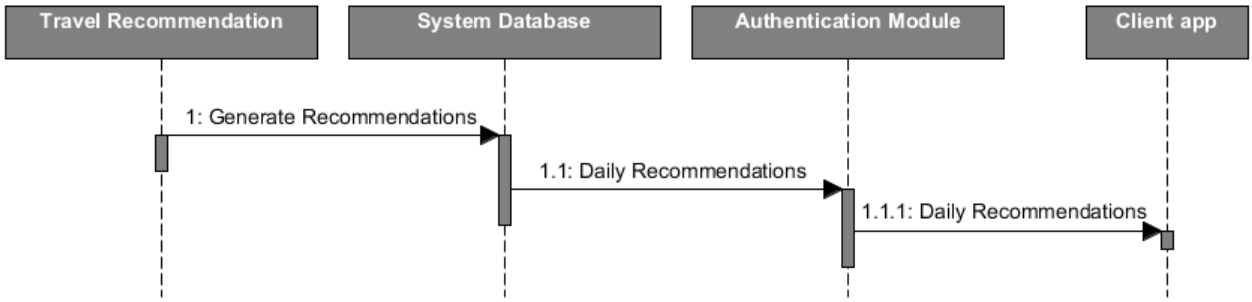

Figure 4.3: Interaction between modules to send recommendations to the user

garding their starting and destination locations and departure or arrival times, and get back trip proposals based on the current timetables of the system.

- **Client Application:** Communicates with the Request Handler in order to post a travel request.
- **Request Handler:**
  1. Stores a request to the **System Database**
  2. Communicates with the **Route Generator** in order to correspond the starting and ending points of the request to existing bus stops
  3. Forwards the request to the **Travel Planner**
- **Travel Planner:**
  1. Selects the bus trips which could possibly satisfy the request
  2. Creates the user trips corresponding to the search results and stores them to the **System Database**
- **Request Handler:**
  1. Forwards the response, which contains search results, to the **Client Application** as shown in figure 4.4.

- **Case 4: Booking**

  Users can make bookings by selecting either travel recommendations or search results. Reservations should be stored at the **System Database**, to allow users to receive reminders about their upcoming trips, notifications concerning delays or cancellations, and similar future recommendations. Moreover, reservations can also be canceled; the process is showed in figure 4.5.

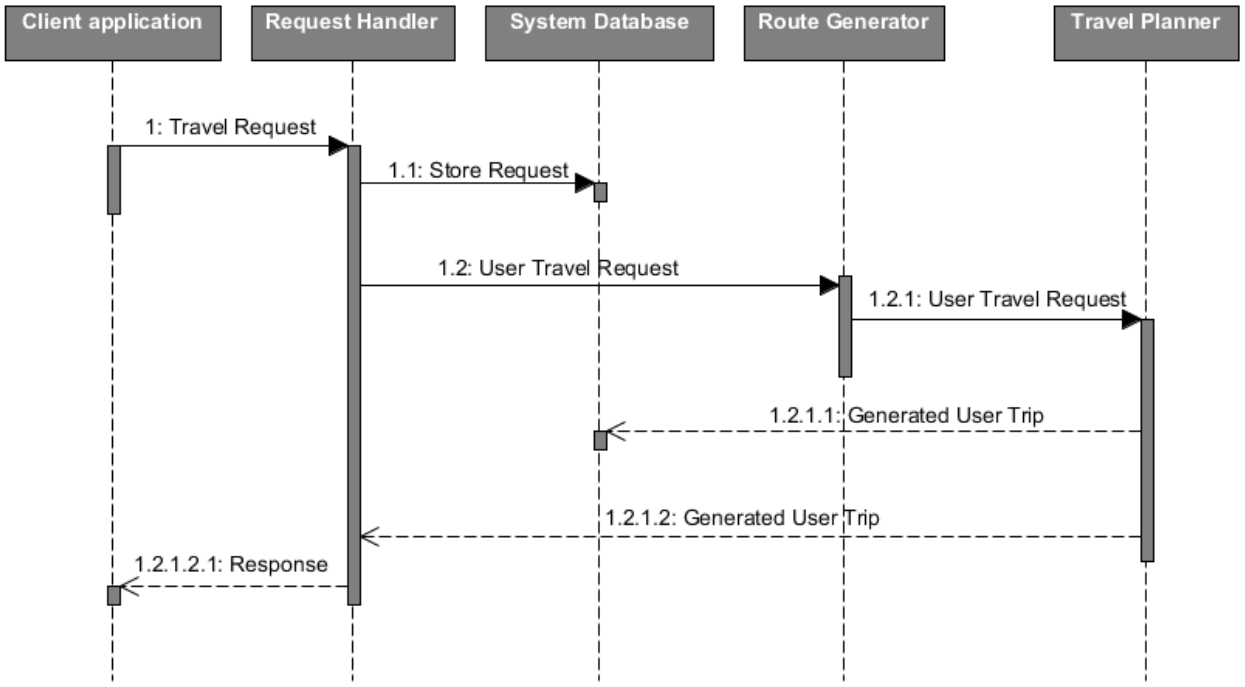

Figure 4.4: Interaction between modules to send trip information to the user

- **Client Application:** Communicates with the **Request Handler** in order to register or cancel a reservation.
- **Request Handler:** Stores new reservations to the **System Database** or removes existing ones in case of cancellation.

- **Case 5: Notification**

  Notifications are used in order to remind users about upcoming trips or inform them about delays, cancellations, or possible reservation changes. MoNAD takes advantage of the **Google Cloud Messaging (GCM)** service, which allows for messaging communication between the **Authentication Module** and the **Client Application**. Furthermore, notifications are also stored in the **System Database** to allow users to browse past notifications.

  - **Client Application:** Communicates with the **Authentication Module** in order to register the Google registration token of the user's Android device and receive past notifications.
  - **Authentication Module:**
    1. Stores the Google registration token to the **User Database**

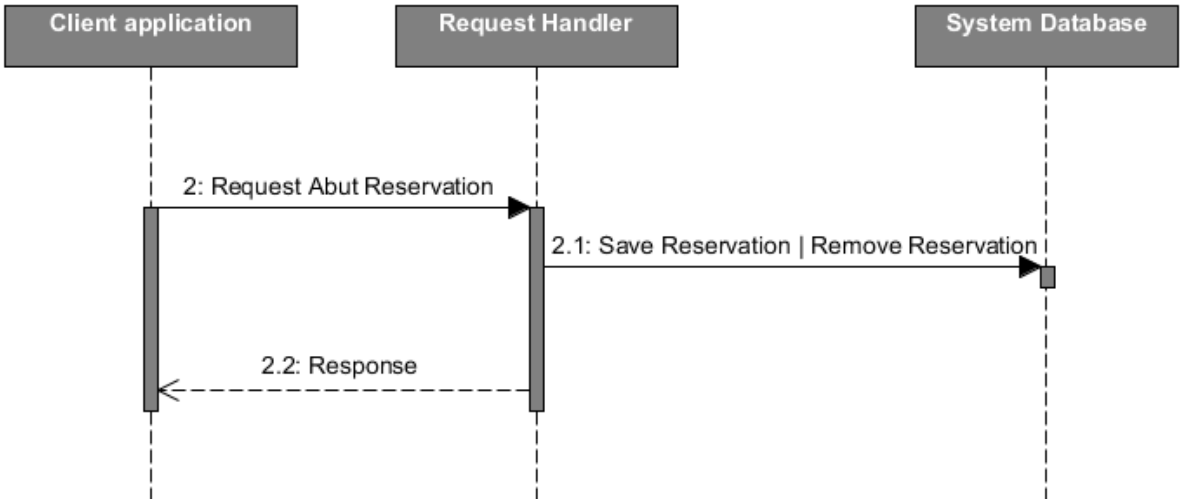

Figure 4.5: Interaction between modules when user wants to pursue a trip

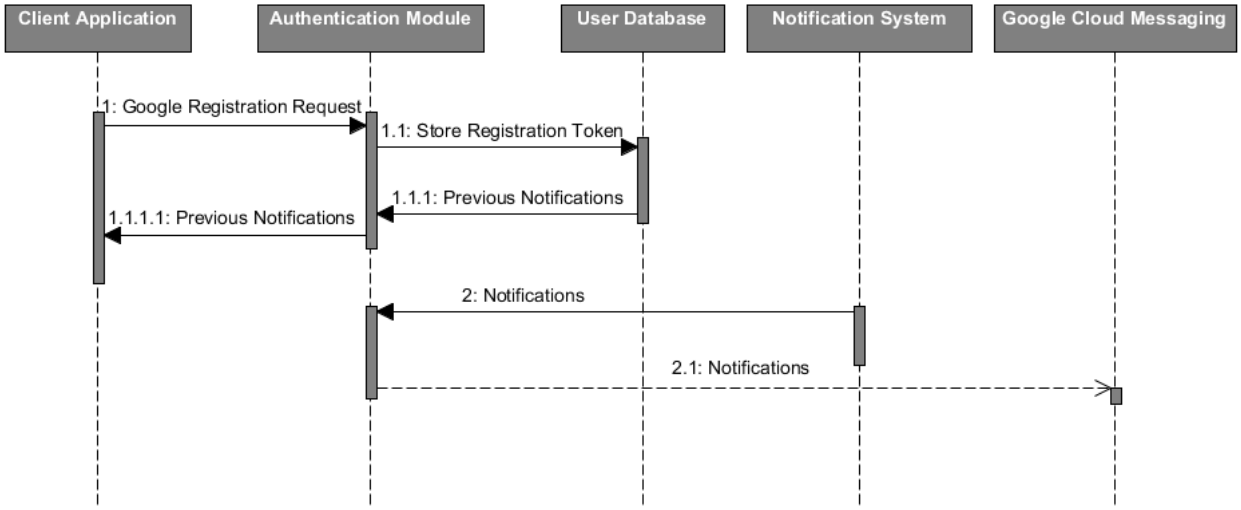

Figure 4.6: Interaction between modules after Google registration

2. Retrieves past notifications from the **User Database** and sends them to the **Client Application**
3. Forwards notifications, which are being sent by the **Notification System**, to the **Google Cloud Messaging (GCM)** service as shown in figure 4.6

– **Notification System:** Periodically checks the **System Database** for trip reminders to be sent and past trips without user feedback, and forwards the corresponding notifications to the **Authentication Module**.

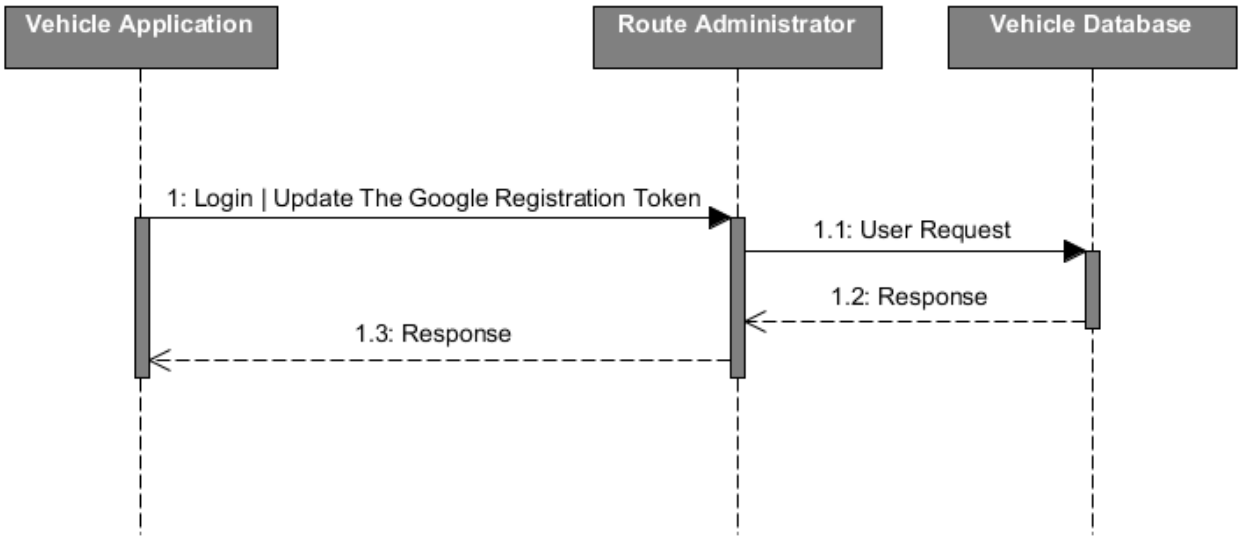

Figure 4.7: Interaction between modules after google registration

Vehicle Application

- **Case 1: Vehicle Authentication**

  Vehicle profiles containing information regarding vehicle id, current driver id, password, and current bus line are stored in the **Vehicle Database**. Drivers can log into the **Vehicle Application** by providing the corresponding credentials. In addition, a Google registration token corresponding to the Android device of the **Vehicle Application** is also stored at the **Vehicle Database**, in order to allow the application to receive notifications using the **Google Cloud Messaging (GCM)** service.

  - **Vehicle Application:**
    - Communicates with the Route Administrator in order to register a login request and update the Google registration token, stored in the Vehicle Database
  - **Route Administrator:**
    - Connects to the Vehicle Database in order to evaluate the received login credentials or update the Google registration token
    - Responds to the login request (see figure 4.7).

- **Case 2: Next Bus Trip Information**

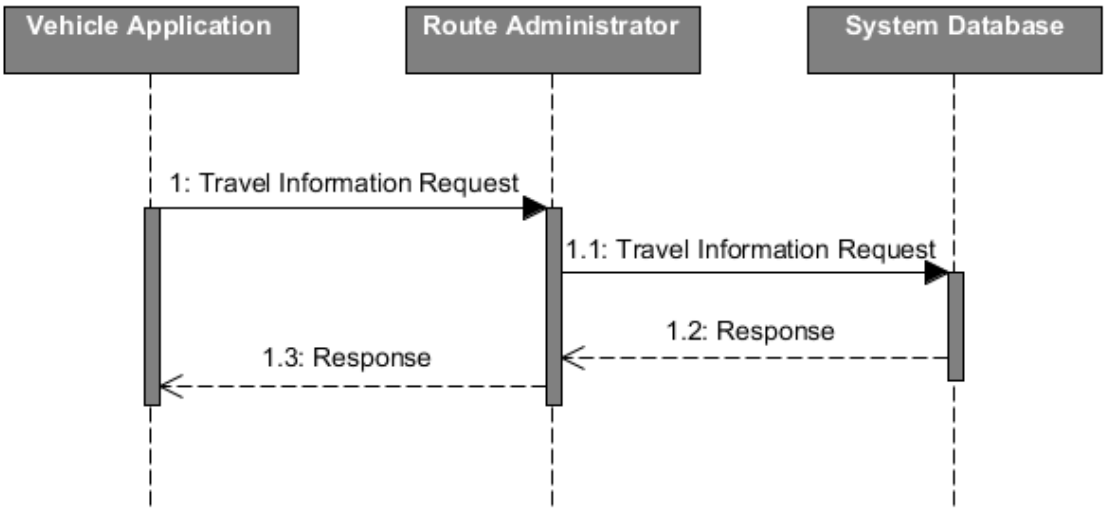

Figure 4.8: Interaction between Vehicle Application with Route Administrator in order to send travel information

  - **Vehicle Application:**

    Communicates with the **Route Administrator** in order to receive information regarding the next scheduled trip of the vehicle, such as the list of bus stops and the corresponding arrival time at each bus stop.

  - **Route Administrator:**

    Responds to the request by connecting to the **System Database** and retrieving next scheduled trip data as shown in figure 4.8.

- **Case 3: Trajectory**

  Apart from the list of bus stops and their coordinates, the trajectory of the route is also needed. For this reason, the **Vehicle Application** makes a request to the **Route Generator**, providing the list of bus stops and receives the entire trajectory as response.

- **Case 4: Traffic Information**

  - **Vehicle Application:** Makes a request to the **Route Administrator** in order to receive traffic information.

  - **Route Administrator:** Responds to the **Vehicle Application** by sending a list containing current traffic incidents of the city (see figure 4.9).

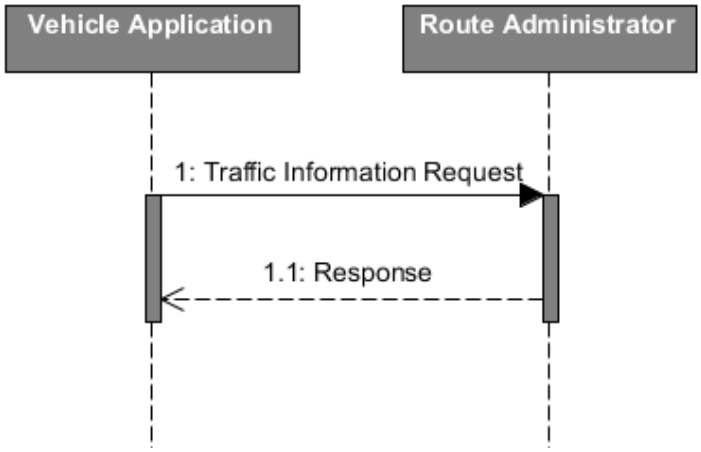

Figure 4.9: Interaction between Vehicle Application and Route Administrator for traffic information

4.2.3 Data model

System Database

Technologies

System Database stores all the information needed by MoNAD in order to operate, except for user or vehicle profiles which are stored separately for security purposes. It utilizes MongoDB which is an open-source document database. Documents in MongoDB are following the BSON serialization format, and are similar to JSON [16] Objects, consisting of key-value pairs. Values may be of types: documents, arrays, or arrays of documents. Three key features of MongoDB are high performance, high availability, and automatic scalability [17].

Collections

Within the System Database, there are 10 collections: TravelRequest, UserTrip, BookedTrip, TravelRecommendation, Route, TimeTable, BusTrip, BusStop, Geofence, and Notifications.

TravelRequest

- *_id*: ObjectID
- *userID*: ID of the user that made a request
- *startPositionLatitude*: requested starting latitude, input by user
- *startPositionLongitude*: requested starting longitude, input by user

- *endPositionLatitude*: requested ending latitude, input by user
- *endPositionLongitude*: requested ending longitude, input by user
- *startBusStop*: reference to the closest bus stop requested starting position, generated by RouteGenerator
- *endBusStop*: reference to the closest bus stop to requested ending position, generated by RouteGenerator
- *startTime*: datetime of requested start time
- *endTime*: datetime of requested end time
- *requestTime*: datetime of request

TravelRecommendation

- *_id*: ObjectID
- *userID*: ID of the user that the recommendation is for
- *userTrip*: reference to a userTrip

UserTrip

- _id: ObjectID
- *userID*: ID of the user that the to which the userTrip was assigned
- *line*: bus line of the userTrip
- *busID*: ID of the vehicle that transported the user during the userTrip
- *startBusStop*: name of bus stop where the user boarded the bus
- *endBusStop*: name of bus stop where the user departed the bus
- *startTime*: datetime of when the bus was at startBusStop
- *endTime*: datetime of when bus was at endBusStop
- *requestTime*: datetime of when request was made
- *feedback*: integer rating of userTrip;

  values: -2, -1, 1, 2, 3, 4, 5;

  -1 indicates unrated;

  -2 indicates unrated and notified;

  1-5 indicates the quality of the trip input by user

- *trajectory*: list of bus stop names in the order of route
- *requestID*: reference to the request made to initiate creation of userTrip
- *next*: reference to the userTrip that the user took next in the journey
- *busTripID*: reference to the busTrip–the full journey the bus took from start to finish
- *booked*: boolean indicating whether the user booked the userTrip

BookedTrip

- *_id*: ObjectID
- *userID*: ID of the user booked the trip
- *partialTrips*: list of references to the userTrips included in the bookedTrip; can be one or many userTrips

Route

- *_id*: ObjectID
- *line*: bus line of the route
- *duration*: total duration of the route from start to finish
- *frequency*: integer indicating how often the route occurred
- *date*: datetime of the day the route occurred
- *trajectory*: list of bus stops that the route followed. The list contains objects with key *busStop* with a reference to a bus stop within the BusStop collection and *interval* key which is an Integer indicating the number of minutes it took to travel to the next bus stop in the trajectory

TimeTable

- *_id*: ObjectID
- *line*: bus line of the timetable
- *date*: datetime of the day the timetable was active
- *timetable*: list of references to bus trips within the BusTrip collection which defined the timetable for that day

BusTrip

- *_id*: ObjectID
- *busID*: ID of the vehicle used for the busTrip
- *capacity*: number of people that could ride the bus
- *line*: bus line which the bus trip will serve
- *startTime*: datetime of when the bus was at the first bus stop in the trajectory
- *endTime*: datetime of when the bus was at the final bus stop in the trajectory
- *trajectory*: list of objects giving information about the bus stops the bus will stop at during the journey. Each object has a key: *busStop* with a reference to a bus stop within the BusStop collection, keys: *boardingPassengers* and *departingPassengers* have integers as the values, indicating how many people are boarding and departing the bus at the designated bus stop, respectively. Key: *time* is a datetime indicating the time the bus was be at the bus stop.

BusStop

- *_id*: ObjectID
- *name*: string of the bus stop name
- *latitude*: latitude position of the bus stop
- *longitude*: longitude position of the bus stop

Geofence

- *_id*: ObjectID
- *userID*: ID of the user that has entered/exited the Geofence
- *latitude*: latitude position of the Geofence
- *longitude*: longitude position of the Geofence
- *time*: datetime of when the user entered/exited the Geofence

Notifications

- *_id*: ObjectID

- *text*: string of text for user to read in the message
- *iconID*: Integer indicating the notification type
- *userID*: ID of the user
- *time*: datetime of when the notification was sent
- *partialTrips*: list of userTrip objects, each object contains the same information found in the UserTrip collection object.

4.2.4 Request Handler

The Request Handler (RH) is the main server of the system; most of the communication between the Client Application and the back end is done through it. As the name suggests, its main objective is to handle requests and provide the clients with the results of those requests. Errors and warnings are logged into files and can be viewed by the server administrator for troubleshooting purposes.

The different types of requests supported by the system are as follows:

- **Search:** The client specifies some criteria for a bus trip search (e.g. departure/destination address and time) and the RH stores that data in the System Database. The Route Generator is used to translate the given addresses into GPS coordinates. The data are then used as input by the Travel Planner to find suitable bus trips for the user. Finally, the Travel Planner returns the results to the RH and the trips are sent back to the Client Application.
- **Quick Search:** The procedure is the same as with the main Search request, except the client only specifies the destination address; the rest of the data are given default values (for example, the current GPS coordinates of the user are used as the departure address).
- **Reset Password:** An email address is given as input to the RH and an email containing a random 4 digit verification code is sent to that address from the system's email account. The code is then also sent back to the Client Application to compare it to the code entered by the user.

- **Booking:** The mongoDB ID of the specified trip is taken as input and the trip details are checked against existing bookings to see if the user has already booked it. If not, the trip is added to the user's trip list and the values for number of passengers boarding/exiting the bus are incremented.
- **Cancel Booking:** The booking is removed from the user's trip list and the values for number of passengers boarding/exiting the bus are decremented.
- **Get Bookings:** The user ID is given as input and the user's trips are returned. However, before sending the list back to the Client Application, the RH removes all trips that are older than a threshold, which is specified as a configuration parameter.
- **Update Feedback:** The Client Application sends the new feedback values for some booked trips and the values are updated in the appropriate collection of the database.
- **Store Geofence Info:** All geofence data is sent to the RH and stored in the database for future use.

4.2.5 Simulator

The simulator is designed to contain three functions: data generation, vehicle visualization, and data distribution visualization.

Purpose of the Simulator

The goal is to simulate the Client Application; create an arbitrary number of clients that send out a predefined set of traveling requests. This is also used as a way to stress-test the system.

Explanation of the Simulator

The first step is generating travel requests within the GAMA platform. Then, all requests are saved into a text file and a Java program parses the requests and sends them to the Request Handler via HTTP in a unified format. The Java program is used because the GAMA socket skill (a function whose purpose is to post requests) is not working properly and there are no other alternative options to be used based on the documentation.

Data Generation

The client simulator is designed to simulate a batch of travel requests to the server. GAMA is an agent-based modeling platform. The agent generates data by performing an experiment on a species - in this case, the client. The experiment is about sending requests using a unified format. The requests mimic rush-hour, busy bus stops, and users' travel habits to make it as realistic as possible.

Data distribution visualization

Charts are introduced to display the distribution of requests. Using charts is helpful for making sure that the data conform to the system's specification. One chart visualizes the workday requests' distribution (by time duration and bus stop), another shows the same information but is based on the weekend requests, and a third chart shows the percentage of habitual and random data.

Vehicle visualization

During the execution of the program, a bus moves through a fixed route with bus stops along it. Next to the bus stop names there are numbers representing how many passengers board and exit the bus.

Parameters of the Data Generation Process

A panel with some parameters is used to control the process of data generation. For example, the adjustable parameters are number of clients, weekday or weekend data, and time duration of the requests' starting time.

4.2.6 Feedback

The Feedback Module is a back-end component of MoNAD, it is responsible for notifying users of completed trips which have not been given feedback. Its implementation could be described

by the following steps:

Step 1: Identifying unrated trips

User trips are stored at the UserTrip collection of the System Database. Completed trips can be identified by making a comparison between the current time and the "endTime" value of the trip. Moreover, trips that have not been evaluated contain a "feedback" value of -1. The Feedback Module performs a periodic query to the aforementioned collection in order to retrieve the completed and unrated trips, as well as the IDs of the users which have made the reservations.

Step 2: Notifying clients

Corresponding notifications are generated by the Feedback Module and sent to the Authentication Module, which retrieves the Google registration tokens corresponding to the user IDs from the User Database and forwards the notifications to the Google Cloud Messaging (GCM) service. At this point, it needs to be stated that each user receives only one notification for an unrated trip. For this reason, after generating a reminding notification the Feedback Module changes the "feedback" value of the trip to -2, to avoid sending another notification for the same trip in the future.

Step 3: Receive feedback and update the corresponding entry of the System Database

After being notified, user has the option to update the "feedback" value of a past trip by visiting the "MyTrips" tab. The update request will be received by the Request Handler, which will also change the "feedback" value of the corresponding UserTrip entry at the System Database.

The procedure is visualised in figure 4.10

4.3 Implementation

This part of the report describes the details of the implementation, looking at each module separately.

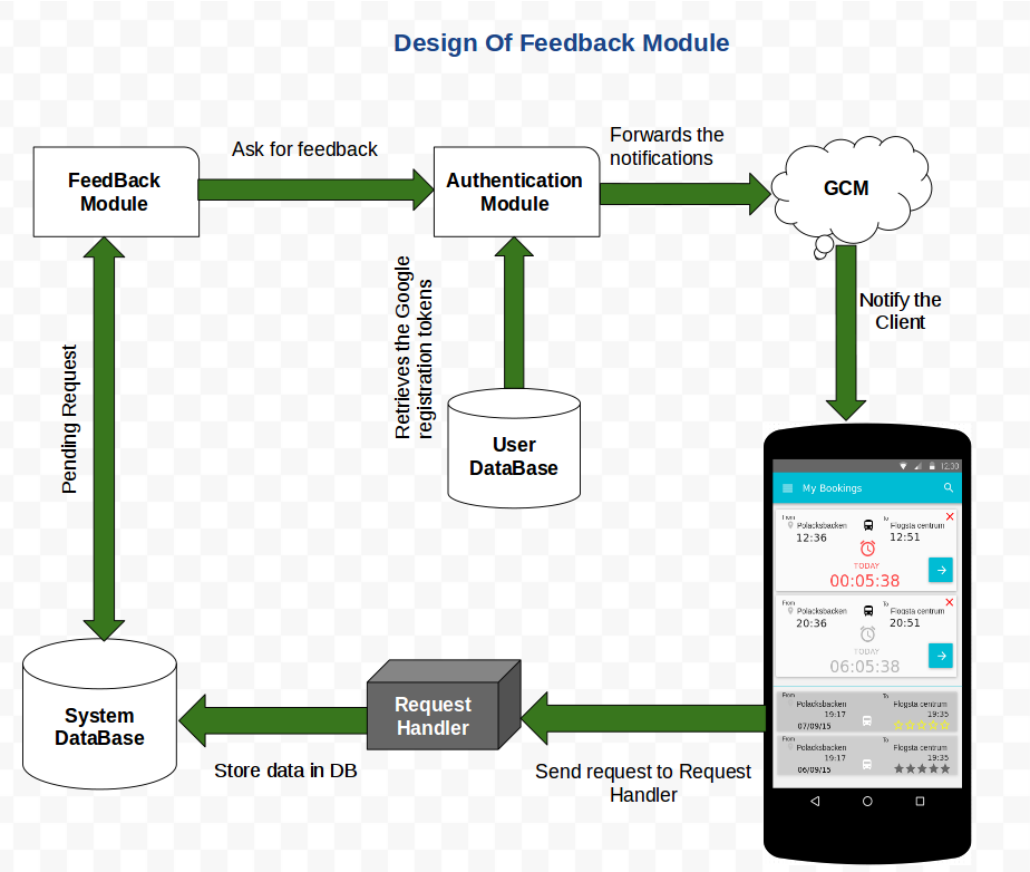

Figure 4.10: Image of the Feedback Module

4.3.1 Client Application

As mentioned in previous sections, the Client Application is an Android mobile application that communicates with the systems back end through the Request Handler and Notification System. The application is configured to support all Android versions ranging between API version 15 (i.e. Android 4.0.3 Ice Cream Sandwich), and API version 23 (i.e. Android 6.0 Marshmallow). The goal behind this decision is to ensure maximum support of active devices, which amounts to 96% in this case [8], all the while the application can support material design features, which require API 21 (i.e. Android 5.0 Lollipop) as the minimum version. Devices with lower versions of Android can still use the same interface thanks to the use of the AppCompat library.

User Interface Diagram

Figure 4.11 is a user interface diagram that describes the structure of the Client Application and the interaction between different activities.

Interaction between Activities

In Figure 4.11, the activities in the Client Application are shown as rectangles with their names inside. An arrow from an activity A to another activity B means that A can start B (usually by user interaction, such as pressing a button), and "finish();" above an arrow means that A will be destroyed by calling finish() after starting B. Destroying an activity when necessary will save significant memory on the Android device and prevent some unexpected or unwanted behavior. For example, after the user logs in successfully (not through GoogleLogIn), MainActivity will be started and LoginActivity will be destroyed, which means that the user cannot go back to LoginActivity by pressing the back button. This prevents the user from logging out by mistake.

One more optimization is that some activities are killed when the user goes to MainActivity by pressing the "Search" button on the upper right menu in any of the activities that extend MenuedActivity. Activities in Android are kept in a back stack. When a new activity is started, it is pushed to the back stack, on top of all other activities in the stack. If required, an existing activity can be resumed and all the other activities on top of it will be destroyed.

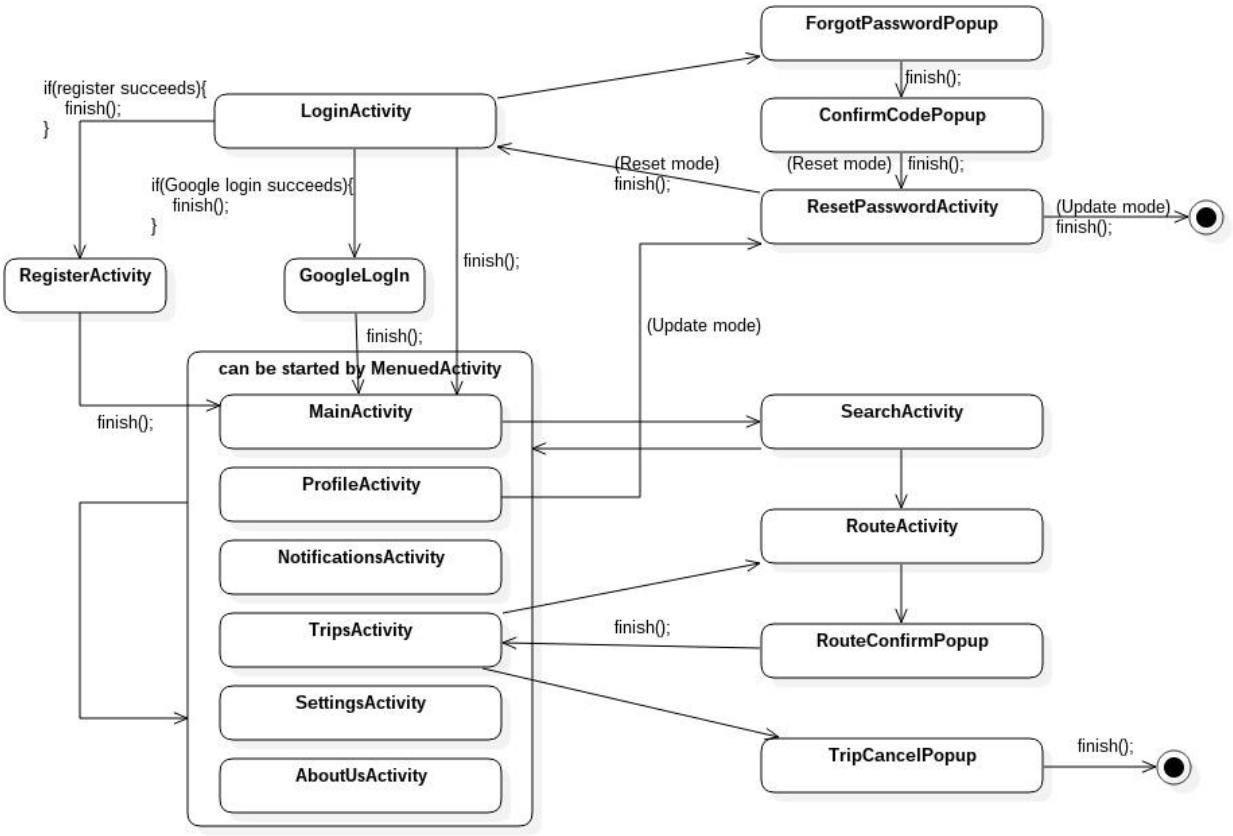

Figure 4.11: User interface diagram of the Client Application

This mechanism is used when the user presses the "Search" button. When this happens, all other activities are destroyed except for MainActivity, which will be resumed after it has been stopped.

Additionally, as indicated by Figure 4.11, MainActivity is started after a user successfully logs in, and all the other activities that started before it are destroyed (e.g. LoginActivity, RegisterActivity, etc.). Therefore, MainActivity is at the bottom of the back stack. If the user presses the back button in MainActivity, it will be destroyed and the whole application is closed. When the back button is detected to be pressed in MainActivity, a dialog is popped out for the user to confirm whether he/she wants to exit the application. This function is added to prevent the user from exiting the application by mistake.

Data Communication

In order to optimize the structure of the Android applications, data is manipulated in an MVC-like approach. In fact, several classes in the application are dedicated to reading the appropriate data from the databases, whereas others act as intermediary classes between the data and the user interface classes in order to format the data according to the needs of the Android applications. One such class is *ClientAuthentication*, which stores the information of the user after login and performs all data manipulation related to a user account using the abstract class *Authentication*. The latter class contains all basic methods that communicate with the Authentication Module and the Request Handler through HTTP requests in order to send data retrieval requests and receive the corresponding responses. The other class that falls in the category of intermediary classes is *Storage*, which relies on static attributes to store and update all information collected from the database such as the users booked trips, personal recommendations, and search results obtained through the Travel Planner.

Lists

During the design phase, it became clear that several pages in the application would contain lists with various content. Android API offers several approaches to generate lists, but the procedure is not as simple as adding a button or a text field. Consequently, two approaches are relied on for two different types of lists:

Activity	List Type	Usage
RouteActivity	static	Bus stops of the chosen trip
MainActivity	interactive	Travel recommendations
SearchActivity	interactive	Search results
TripsActivity	interactive	Users booked trips (active and past)
NotificationsActivity	interactive	Notifications
SettingsActivity	interactive	Languages

Table 4.4: Summary of the use of lists in the Client Application

- Static lists: these require no interaction and will not be updated during the lifecycle of their respective parent activities. These lists are implemented using a simple loop that creates each item separately. First of all, an XML layout file is created to represent a template for a single item, and an arraylist is created and populated with the necessary information to fill an item. Then, for each item in the list, the template layout is used and the data is assigned to the appropriate interface component in the template. Finally, that item is added to the list using an instance of LayoutInflater.

- Interactive lists: these lists either generate components that interact with the user (e.g. a button) or that can be updated (e.g. remove an item from the list). This interaction requires the application to recognize which list item is targeted and send the right information to the next activity. This is where the RecyclerView class (RV) is more convenient, as an adapter class is created and linked to the RV. The adapter class, which extends RecyclerView.Adapter, controls the whole process of adding data to an item and handling its event listeners.

Table 4.4 shows where each approach is used in the Client Application.

Date & Time Formatting

Date and time formats on Android applications are often based on the devices locale, which specifies the language and country settings of the device. However, in order to keep the application simple, date and time information in the Client Application have uniform formats thanks to the functions *formatDate* and *formatTime*, respectively. Below is an example of the output formats of the two functions.

Date: Friday, January 1st 2016 $\longrightarrow$ Fri 01 Jan.

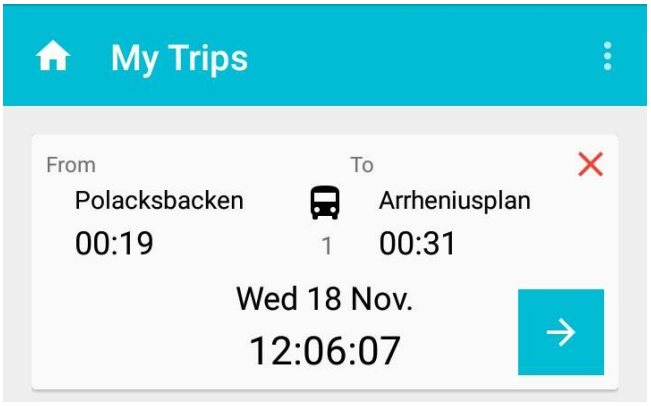

Figure 4.12: Screenshot of the countdown in a booked trip

Time: 2:15PM ⟶ 14:15

In addition to that, countdowns to upcoming trips are used in TripsActivity (see Figure 4.12). This is done thanks to the function onTick() in the class CountDownTimer, which updates the time and controls the format of the displayed text based on the countdown value.

Language Support

Thanks to the diverse origins of the team members of MoNAD, and because language support is very simple in Android Studio, the Client Application currently supports six languages. The only prerequisite to achieve multi-language support is to make sure that all the default text used in the applications source code is referenced from the file `strings.xml`. The standard procedure to add a language, for instance Finnish, is to create a new file `strings.xml` under a new folder "values-fi". The new file should contain most, if not all, translations of the strings in the original file. The default language in the Client Application is English, but the user can change it in the settings page. When another language is chosen, the applications locale is updated, and its text is updated when the user moves to a different activity.

Tab Management

Tabs are used once in SettingsActivity where Language, Alerts and Theme have separate fragments. The tabs are built using a layout that defines the sections and a pager that allows the user to navigate from one tab to another. The layout used is SlidingTabLayout, which is made publicly available by Google [3]. Its main advantage is that users can smoothly swipe

from one tab to another instead of needing to click on the tab headers. The pager uses an adapter, which is an extension of the FragmentPagerAdapter class, to return the appropriate fragment based on the given index of the tab. For example, index 0 would return the Language UI fragment. Finally, it is worth mentioning that despite the settings being updated in real time in the application, they are only truly updated in the database when the user leaves the activity.

4.3.2 Vehicle Application

Current Location of the Vehicle

According to the design of the Vehicle Application, the current location of the bus (more precisely, the location of the device with the Vehicle Application installed) is retrieved by the application at a high frequency and shown on the map. The bus driver can see from the map where the bus is and therefore get a better understanding of the on-going trip.

To keep retrieving the current location at a high frequency, LocationServices provided by Google Play Services is used. After successfully connecting to Google Play Services, a location update request is sent to MyLocationOverlay. The request also specifies the interval in which location updates are made, as well as the level of accuracy of the request. For the Vehicle Application, the accuracy is set as PRIORITY_HIGH_ACCURACY, which is a known good constant.

Everytime the interval expires, a location update is sent to MyLocationOverlay. Then MyLocationOverlay updates the location (latitude and longitude) of the marker and circle on the map and redraws the layer. The marker is used to show the current location of the bus and a circle is drawn around the marker, indicating the accuracy of the location. The bigger the circle is, the less accurate the location.

Since retrieving the location of the bus at a high frequency and accuracy is quite power-consuming, it is necessary to be able to suspend the location update when the vehicle application is not running in the foreground. To achieve this, everytime the MainActivity becomes partially visible or totally invisible, the location update will be stopped. Once the vehicle application is brought back to the foreground, a location update is requested and the location of the bus will be retrieved at a high frequency again.

Location Simulation

Although the vehicle application can show the current location of the bus, it is very hard for the development team to test the vehicle application, especially the functions that rely on the location update. For example, the functions which update the name of next bus stop, or update the number of passengers which are getting on/off the bus at next bus stop. Since it is not realistic to test the vehicle application on a running bus, a location simulation system has been implemented to facilitate the development and testing of the vehicle application. The movement of the bus is simulated manually on the vehicle application by moving a marker along the scheduled route from a starting point to destination point at a moderate pace. The marker is drawn on the map and it represents the current location of the bus.

The Location simulation system is implemented in MyLocationOverlay. Originally, when the location of the bus changes, the location of the marker and the surrounding circle will be updated with the current location of the bus and the layer will be redrawn on the map in the vehicle application. To simulate the movement of the bus along the route without moving the device along the actual route, the marker and the surrounding circle are set to the next point on the trajectory of the scheduled route instead of the current real location of the device.

To do this, MyLocationOverlay keeps a global copy of the trajectory which is passed from MainActivity, as well as a reference to the next point in the trajectory. Everytime MyLocationOverlay receives a location update, the next point of the trajectory is retrieved. The marker and circle are updated to the point retrieved. The reference to the next point of the trajectory is automatically updated when the next point has been retrieved. and the location simulation system does not need iOne special note here is that, the location update is not used, since it represents the real location of the bus but we are using a simulation system which does not need it.

The user can easily switch from simulated location to real location by changing the code in method onLocationChanged(location) in MyLocationOverlay. There are comments provided as instructions and very little needs to be done for the switch.

Name	Functionality	Comment
BRouter[4]	Bike routing and features elevation awareness, alternatives, fully configurable routing profiles and offline routing for Android.	Mainly for bike routing.
GraphHopper[5]	Routing engine for car, bike and more. Can be used as web service, android library, iOS library or Java library.	Limited free usage. Cannot draw custom route.
Mapsforge[6]	Online/offline map rendering.	Meet all the requirement and sufficient documentation.
Osmdroid[7]	Display, marker	Similar to Mapsforge but less documentation and samples.

Table 4.5: Different open-source OSM map rendering libraries on Android

OpenStreetMap integration and route/marker drawing

Since it is required to use open-source third party libraries/products in this project, OpenStreetMap(OSM) was selected as the map data for the Vehicle Application. By integrating OSM map into the Vehicle Application, the bus drivers can look at the current bus route to follow and the current location of the bus on the map.

There exist some open-source libraries on Android which render OSM map. Table 4.5 lists some of the libraries mentioned on the official wiki of OSM.

By comparison, Mapsforge seems to be the most suitable and well-documented framework for the purpose of displaying the map on the Vehicle Application. Therefore, it was selected to render the map on the Vehicle Application.

Mapsforge is a (almost) full/free replacement for Android's MapView class. It provides a free, open-source, offline vector map library for Android and Java-based applications.

Briefly, it works in a similar way as Androids MapView, and by adding different layers on Mapsforges MapView the user can add more features (e.g. routes, markers) than just rendering the OSM data. The usage of different classes and methods can be found on Mapsforges documentaion[1] and its samples which can be found on Github. For a simple offline map (with route) to work, it would suffice to add an instance of Polyline on a TileRendererLayer, which is added on MapView. In a Polyline instance, the user specifies the start and end points, as well as the

[1] `http://mapsforge.org/docs/0.5.2/`

points along the line. Polyline is used to draw the current bus route in the Vehicle Application, and Mapsforge draws a Polyline by linking all the points on it. For the TileRendererLayer, it renders map as a collection of tiles from an offline map package, which is stored locally in the mobile devices. Above Polyline, MyLocationOverlay is added to show the current location and location accuracy of the bus through marker and circle. TileRenderLayer, Polyline and MyLocationLayer are all layers and they are added one on another.

Mapsforge supports both online and offline map rendering. Online mode receives real-time tiles from a tile server. It guarantees that latest update is shown and users dont have to download any map package before using the application. In contrast, it takes time and storage to download an offline map onto a mobile device. However, offline mode has the advantage of presenting the map faster and in a more stable way. For the Vehicle Application, fast and reliable map rendering is top priority, so offline version is chosen.

The Mapsforge Download Server provides offline map packages for many countries of the world, including a map of Sweden. However, the map package of Sweden is usually quite big (the package updated on November 29, 2015 is 289 MB in size), and most of the data is never used in the Vehicle Application since currently only the map data of Uppsala is needed in the project. With OSM Osmosis Tool[2] and Mapsforge Map-Writer plug-in[3], it istext easy to transform a map file in .osm format into a map file in .map format, which is the format used by Mapsforge. The complete process is explained as follows:

1. Download Osmosis 0.4.3.1 and Mapsforge Map-Writer 0.3+. Installation instructions can be found in the Plugin Installation part in the getting started page of Map-Writer.

2. Select a specified area in OpenStreetMap, or manually enter the left/right/top/bottom boundaries in the export section of OpenStreetMap[4].

3. Export the map data. A map file in .osm format will be produced and stored locally.

4. Enter the command in terminal (suppose the current directory is the root directory of osmosis, the downloaded map uppsala.osm is stored in /tmp and the transformed map uppsala.map will also be stored in /tmp):

[2] `http://wiki.openstreetmap.org/wiki/Osmosis`
[3] `https://github.com/mapsforge/mapsforge/blob/master/docs/Getting-Started-Map-Writer.md`
[4] `https://www.openstreetmap.org/export`

```
$ bin/osmosis --rx file=/tmp/uppsala.osm --mapfile-writer
file=/tmp/uppsala.map
```

The map file of Uppsala in .map format is less than 3 MB - about 1% of the Sweden map package provided by Mapsforge Download Server. This will save a lot of storage space and memory the Vehicle application requires in a mobile device and the downloading time of the map package will also be reduced dramatically. In order for the Vehicle application to run properly, the map package should be put in the root directory of the storage in a mobile device and its name "uppsala.map" should not be changed.

Real-Time Traffic Information

After investigation, two popular and powerful APIs were identified: Google Maps and Bing Maps. Since Google Maps does not directly provide traffic information through the API, Bing Maps is employed because it can provide all kinds of traffic information directly through JSON/XML file, which are needed by the application. The mechanism behind this solution is to assign a coefficient for each kind of traffic incident based on the severity. A request contains the following data:

- Target Map Area: Two pairs of coordinates are needed here to locate the area where the traffic information needs to be provided, they are the (South Latitude, West Longitude) and (North Latitude, East Longitude).
  Considering the map shown in figure 4.13 as the target region, these coordinates will locate the borders of this area.

- JSON/XML format: Traffic information can be returned in JSON or XML format. XML format is chosen for this project because XML is easy to parse and to read by a human reader when it is necessary.

- Types of traffic incidents: There are 11 different kinds of traffic incidents that are available (e.g. accident, planned event, road hazard etc.). This API provides the opportunity to choose optional kinds of incidents by using parameters in the HTTP request. For instance, by using `t=2,9`, it will show only Congestion and Construction.

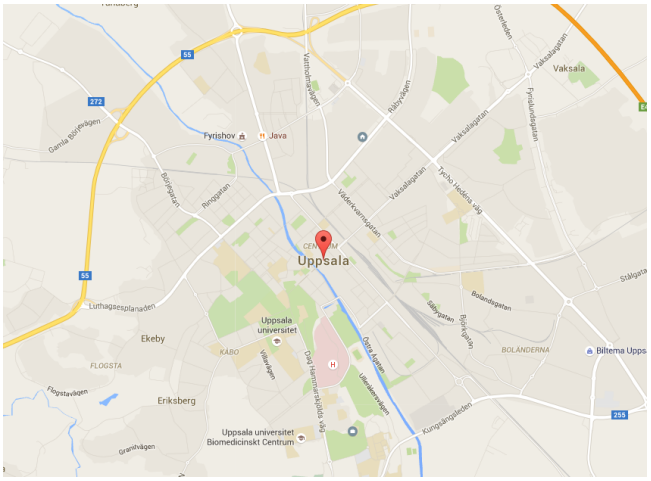

Figure 4.13: A Map of Uppsala

This project will provide several types of traffic incidents for the driver, since all of them can influence the bus traveling time.

- Severity: There are 4 degrees of severity for a traffic incident. LowImpact, Minor, Moderate and Serious. This kind of information can be used to adjust the EDT, which is helpful for both passengers (who may choose the minimum traveling time) and bus drivers (if the severity is Moderate or Serious, the driver may choose another route).

4.3.3 Authentication Module

Description

The Authentication Module is a server responsible for establishing communication between the Client Application and the back-end modules of MoNAD, by handling requests related to authentication tasks, recommendation parsing, as well as notification forwarding.

Implementation

Based on MochiWeb [21] and designed according to the principles of the Erlang Open Telecom Platform [32] and the Actor Model [33], the Authentication Module is a fault-tolerant and highly-available Web server which can serve multiple clients concurrently, offering support for the following requests:

client_sign_up

Sender: Client Application

Parameters: Username, password, email, phone, and Google registration token.

Description: A new client wants to be registered in the system. The Authentication Module connects to the User Database in order to add a new entry.

Response: Contains a confirmation code and the ID of the new client.

client_sign_in

Sender: Client Application

Parameters: Username, password, and Google registration token.

Description: An existing client wants to log into the system. The Authentication Module connects to the User Database in order to evaluate the provided credentials.

Response: Contains a confirmation code along with profile information, such as client ID, email, phone, preferred theme and language, activated notifications, recommendations, and location monitoring.

google_sign_in

Sender: Client Application

Parameters: Email and Google registration token.

Description: A client wants to login using a Google account. The Authentication Module connects to the User Database in order to retrieve an ID for this client. If the provided email already exists in the system, an already existing ID is returned. Otherwise, a new ID is registered.

Response: Similarly with the regular sign_in request response, it contains a confirmation code along with profile information, such as client ID, email, phone, preferred theme and language, activated notifications, recommendations, and location monitoring.

client_profile_update

Sender: Client Application

Parameters: Client ID, username, email, and phone.

Description: An existing client profile needs to be updated. The Authentication Module connects to the User Database in order to perform the update.

Response: Confirmation code

client_settings_update

Sender: Client Application

Parameters: Client ID, preferred theme and language, activated notifications, recommendations, and location monitoring.

Description: The settings of an existing client need to be changed. The Authentication Module connects to the User Database in order to perform the update.

Response: Confirmation code

client_existing_password_update

Sender: Client Application

Parameters: Client ID, old password, and new password.

Description: The password of an existing client needs to be changed. The Authentication Module connects to the User Database in order to perform the update.

Response: Confirmation code

client_forgotten_password_reset

Sender: Client Application

Parameters: Email and new password

Description: An existing password should be updated, since the client cannot log into the application. The Authentication Module connects to the User Database in order to reset the password. Email is used for authentication.

Response: Confirmation code

get_recommendations

Sender: Client Application

Parameters: Client ID

Description: A list of daily travel recommendations needs to be sent to the Client Application. The Authentication Module connects to the System Database in order to retrieve and parse the corresponding recommendations.

Response: List of recommendations

get_notifications

Sender: Client Application

Parameters: Client ID

Description: A list of past notifications needs to be sent to the Client Application. The Authentication Module connects to the System Database in order to retrieve and parse

the corresponding notifications.

Response: List of notifications

remove_notification

Sender: Client Application

Parameters: Notification ID

Description: Client has selected to delete a past notification and thus, the notification should also be removed from the System Database. The Authentication Module connects to the System Database in order to delete the corresponding entry.

Response: Confirmation code

generate_notification

Sender: Request Handler

Parameters: Client ID and booked trip ID.

Description: This request is sent by the Request Handler in order to notify a user about a successful booking request. The Authentication Module connects to the System Database in order to retrieve the details of the booking and generate the corresponding notification. Furthermore, it parses the Google registration token from the User Database based on the provided client ID. Finally, the notification is forwarded to the Google Cloud Messaging (GCM) service.

Response: Confirmation code

send_notification

Sender: Could be any system component.

Parameters: Client ID, notification title and message.

Description: This request could be sent by any system component which needs to send a notification to the Client Application. The Authentication Module receives the client ID, connects to the User Database in order to retrieve the Google registration token corresponding to this client, and forwards the notification to the Google Cloud Messaging (GCM) service.

Response: Confirmation code

get_bus_stops

Sender: Client Application

Parameters: None

Description: This request is related to the background monitoring service of the Client Application. It is sent after the user has successfully logged into the application, in order to receive the list of bus stop geofences. The Authentication Module connects to the System Database, retrieves the requested data and responds to the request.
Response: List of bus stop geofences

Technologies

MochiWeb is an Erlang library for building lightweight HTTP servers. Based on MochiWeb, the Authentication Module could be considered as a set of processes categorized into workers and supervisors. Workers are responsible for serving incoming requests, while supervisors supervise workers by making sure that they finish their tasks and are restarted in case of errors.

Emysql [34] is an Erlang driver towards MySQL databases. The Authentication Module utilizes Emysql while connecting to the User Database. A connection pool is used, so as to allow multiple workers to connect to the database in parallel. Moreover, prepared statements are supported in order to avoid SQL injection queries.

PyMongo [13] is a Python distribution containing tools for working with MongoDB. It is used by the Authentication Module while connecting to the System Database.

ErlPort [20] is a library which allows Erlang to connect with other programming languages. It is used by the Authentication Module while connecting with PyMongo.

4.3.4 Request Handler

The Request Handler is implemented on Gunicorn [11], a Python WSGI HTTP server. Gevent workers are used for concurrency and asynchronous communication with the MongoDB database. Gevent [12] was chosen because it is fully supported by PyMongo [13], which is a widely used and recommended Python-MongoDB driver. All these technologies together result in a lightweight server that can handle hundreds of requests per second concurrently.

Nginx [14], a reverse proxy server, complements the main server and is the entry point for all incoming communication. It redirects all the requests to the main server as it sees fit and

provides many benefits, like DDoS protection, single point of access to many servers, SSL encryption, and hiding the existence and characteristics of the origin server. Also, it provides great scalability since more origin servers can be added in the background while keeping a single IP for communication with the clients.

The server code is divided into four files: `server.py`, `serverSettings.py`, `serverConfig.py` and `logging.conf`. `Server.py` is the application code, `serverSettings.py` contains the Gunicorn configuration (e.g. number of workers, the path for the server logs etc.), `serverConfig.py` is the configuration file for the application code where passwords and IP addresses are stored and, finally, `logging.conf` contains the settings for the default logging.

4.3.5 Travel Planner

General Description

The Travel Planner (TP) is one of the key modules of the entire system. This element of MoNAD checks which routes fit the user's request and determines at which time the user can travel to reach his destination according to his preferences. The user has multiple options to influence the search result:

1. He can select a start time for his journey: The TP will then select routes starting as soon as possible after the given time.
2. He can select an end time: The results will contain routes that end as close before the chosen time as possible.
3. A third option is to select the time of the trip: In this case the results will be sorted by the time they take, with short routes being preferred.
4. The final option is the waiting time: Here the results will be sorted by the traveling time as well, but routes that start as soon as possible will be ranked higher.

The user can choose between options one and two, and three and four. The default values as shown in the client application are start time (option 1) and trip time (option 3).

Integration with Other Modules

The TP has multiple connections to other parts of the system. It is designed as a PYTHON class which is instantiated by the RequestHandler (RH). The TP provides one interface which will start the algorithm to find and evaluate possible routes. It accesses different collections in the Mongo database:

- TravelRequest: The relevant request is loaded to get the basic information the search is based upon.
- Route: The TP looks for fitting routes in this collection.
- TimeTable: From this collection all possible BusTrips are loadable.
- BusTrip: This collection provides the TP with the concrete times when buses are at certain bus stops.
- UserTrip: The TP stores the best routes in this collection.
- BusStop: Since the BusTrip collection only contains the IDs of the bus stops in the trajectory, this collection is needed to retrieve the names of the bus stops for the trajectories stored in the UserTrip collection.

After running the algorithm the TP returns a JSON-formatted object containing the five best routes to the RH. While the TP is not directly connected with the Look Ahead function, it is dependent on its work creating the routes and bus trips, and writing these into the database.

Algorithm

The work of the TP can be divided into four phases. The first part is purely for programming purposes as it contains the initialisation. The second segment of the code looks for possible connections between the start and end point of the trip. Here the database is queried in multiple steps: First the routes are checked for direct connections. Afterwards the algorithm tests if there are possible routes that contain a switch from one bus line to another. Searching for a route with an intersection works as follows: All routes going through the starting bus stop are evaluated, beginning at the bus stop after the starting one. The algorithm then tries to

find routes going from the current bus stop to the destination. If a bus line contains both the current and the end bus stop, it is stored as a possibility together with the first line leading to the current bus stop.

If there are no results after that search, the algorithm will continue and look for routes with more switches. This works in basically the same way, testing if other routes from the current bus stop have a connection to the end bus stop. After the generation of a list of possible connections to the destination is done, the third phase can begin. Here all saved routes are evaluated and ranked according to the users preferences.

The TP here differentiates between trips where the user wants to either arrive before a certain time or leave after a given point in time. If the first option is chosen, it will start looking for the last part of the trip before finding the previous ones. To achieve this the TP compares the arrival time of the bus at the destination with the time the user wants to arrive. If this is the case, the algorithm moves on the previous segment of the route and finds the bus arriving closest to the departure time at the intermediate bus stop. This continues until the first part of the trip is reached. After determining the start time of the route it is compared to the other trips found by the algorithm and ranked based on the difference between the arrival and the users given time. If it is within the five best routes, it will be stored, otherwise it is discarded as not well fitting.

In the case the users wants to leave his starting point after a certain point in time, the algorithm works very similar. Here the partial routes are evaluated in the order they will be taken. Starting with the first segment the TP checks whether the bus is at the start bus stop after the user planned to depart and, if so, continues with the second part, looking for a trip that leaves the intermediate stop after the arrival there. After reaching the last branch of the trip, the arrival and departure time are compared to the other possible routes and the trip is ranked among them or simply discarded.

After the evaluation only the five (or less) best trips are stored in the database and returned to the RH. This happens in the last phase of the TP. Here the details of each route segment are stored in the UserTrip collection of the database. To return the details of the trips to the user the RH requires the data to be formatted as a JSON-object. This is also done by the TP to reduce the number of database accesses. The TP concludes its work by returning the created JSON object to the RH.

4.3.6 Travel Recommendations

Idea

Another important feature of the system is the Travel Recommendations module, which is based on the idea that users' requests are driven by certain habits and therefore it is likely that they can be grouped together based on certain traits. More specifically, people tend to follow a number of patterns when using public transportation (e.g. going back and forth to their workplace), thus the goal is to successfully identify them and try to give recommendations in advance, right after the user logs in.

Implementation

To achieve that, a clustering algorithm is needed in order to properly separate the users' past requests into different groups of habits. K-means[5] is used, which is a simple clustering algorithm where every cluster is represented by a centroid. A number of centroids is defined in the beginning and all the objects are assigned to the group of the closest centroids. Next, the centroids are recalculated and the same process is repeated iteratively, until a point is reached where the new centroids do not significantly differentiate from the old ones.

Particularly for the trip recommendations, each request is represented by data containing the users departure and arrival location (as coordinates) and the respective times. K-means was selected for two main reasons. First, the ease of integrating the algorithm in such a way that allows high scalability through Apache Spark MLlib [6] and second, since each habit is represented by the values of the centroid, it can then be used to seek the best match from the current timetable by comparing the actual centroid against every route of the timetable. Following, there is an example of how k-means is expected to extract the users trends out of a number of requests that are shown in Table 4.6

In the example above, the first table holds the requests of the past two months for a random user. It is apparent that the two main trends are requests from Centralstation to Polacksbacken

[5] https://en.wikipedia.org/wiki/K-means_clustering
[6] http://spark.apache.org/docs/latest/mllib-guide.html

Requests	**Departure**	**Arrival**	**Dept. time**	**Arr. time**
req1	Centralstation	Polacksbacken	-	9.15
req2	Centralstation	Polacksbacken	-	9.25
req3	Centralstation	Polacksbacken	-	9.30
...	...	...	...	...
req118	Centralstation	Polacksbacken	20.40	-
req119	Centralstation	Polacksbacken	20.20	-
req120	Centralstation	Polacksbacken	20.30	-

Table 4.6: Past requests of a user

every morning, with priority to arrival time between 9.15 - 9.30 and requests from Stadshuset to Ekonomikum with priority to departure time between 20.20 - 20.40.

Centroids/Habits		
Route	Dept. time	Arr. time
Centralstation - Polacksbacken	9.00 - 9.15	9.15 - 9.30
Stadshuset - Ekonomikum	20.20 - 20.40	20.25 - 20.45

Table 4.7: Clusters extracted out of past requests of the user

After applying k-means onto this dataset of requests, the outcome that we desire is two centroids that will have the same departure and arrival locations and the respective times between the aforementioned time spans, as shown in Table 4.7.

As soon as the clusters and their respective centroids are formed, they have to be compared against every route that exists in the timetable. It is first done for the departure information, then for the arrival information and in the end a final score is calculated to indicate whether, and to what extend, the specific trip would suggest a suitable match or not. The procedure is shown in the image below.

Due to the way the algorithm works, it is expected that the more the users follow consistently specific habits, the better results they will receive. However, it is inevitable that there will be some random requests as well that will not follow the usual behavior, so it remains to be seen how these will be handled.

4.3.7 Route Generator

The RouteGenerator (RG) modules is a module that deals with geographic data e.g. bus stops information, road network and street names. It is mainly a help module for other modules that need geographical information. The RG includes functions like finding the closest bus stop, and finding the fastest path between two coordinates. The main idea is that the RG should be an interface between other modules and geographical data so that not every module needs to implement these functionality.

RG is implemented as a lightweight HTTP Erlang server using Python to parse the data and for calculation. The underlying data comes from an OSM XML file downloaded from OpenStreetMap [19]. It is basically a list of OSM element (nodes, ways, and relations).

Technologies/libraries used

- numpy
- xml.sax
- ErlPort
- MochiWeb

Bus stops

In OpenStreetMap bus stop locations are located in the way element (on a road). A bus stop name is usually associated with of more than one bus stop position, e.g. 'Centralstation' have 16 bus stops. When searching for a bus stops coordinates, the result is approximated in the way that the first match is given as answer.

Missing bus stops names and locations have been added to the OSM XML file using JOSM [22] to match ULs data.

Integration with other modules

The RG is its own web server and does not depend on other modules, but other modules use the service of RG. E.g. the Client Application use the RG to find the bus stop closest to the user's location, and the translation of strings to an address or bus stop.

Web server

When starting the server the geographic data is first parsed from the OSM XML file and then it allows for post request to be sent to the server. The RG is also used to make the bus stop graph, which is a graph that shows how the bus stops are connected to each other.

The server is implemented using the Erlang libraries MochiWeb [21] and ErlPort [20]:

- MochiWeb, a Erlang library for implementing a lightweight HTTP server
- ErlPort, a erlang library to connect Erlang to Python.

The server supports three post request:

- *get_coordinates_from_string*: This function tries to find the coordinates from a string that it suppose to be a bus stop name or an address. It prioritises to return bus stops over addresses if there are a bus stops and streets with the same name.
- *get_nearest_stops_from_coordinates*:: Finds the nearest bus stops according to the coordinates supplied and within the specified radius.
- *get_route_from_coordinates*: Finds the route from a list of destinations. The first element is the starting point and the last element is the ending point. Other coordinates in the list are intermediate point to visit in increasing order.

Algorithms

A* The path finding function uses the A* search algorithm to find the fastest path between two coordinates in the directed graph that constituting the road network of Uppsala. Deter-

mining the fastest way can be tricky, due to different conditions of the road on different times of the day.

A* uses a heuristic best-first approach, which tries to use the most promising node at a given point to find the fastest way. When evaluating alternatives a heuristic function is used, in this case the estimated time to the goal. An expected shorter distance to the goal is more likely to be the faster alternative.

Distance The haversine formula is used to calculate the distance between two coordinates. For the simplicity, the earth is in this case approximated to be a sphere with a radius of 6,371 kilometers (the mean radius of the earth). In reality the earth is more of an oblate spheroid and therefore an small error could occur in the calculation. This error is usually less than 0.03%[23], 30 meters per 10 kilometers.

4.3.8 Route Administrator

Description

The Route Administrator is a server responsible for establishing communication between the Vehicle Application and the backend modules of MoNAD, by handling requests related to authentication tasks, trip and passenger information, traffic data, as well as notifications.

Implementation

Based on MochiWeb [21] and designed according to the principles of the Erlang Open Telecom Platform [32] and the Actor Model [33], the Route Administrator is a fault-tolerant and highly-available Web server which can serve multiple clients concurrently, offering support for the following requests:

vehicle_sign_in

Sender: Vehicle Application

Parameters: Driver ID, password, and number of bus line.

Description: A new driver wants to log into the Vehicle Application. A request will be

received by the Route Administrator, which will connect to the Vehicle Database in order to verify the provided credentials.

Response: Contains a confirmation code along with the ID of the vehicle.

vehicle_get_next_trip

Sender: Vehicle Application

Parameters: Vehicle ID

Description: The Vehicle Application should receive information regarding the details of the next trip. The Route Administrator connects to the System Database, retrieves the details of the next trip and responds to the Vehicle Application.

Response: Contains a confirmation code along with the list containing the bus stops of the trip and the corresponding arrival times.

send_notification

Sender: Could be any system component.

Parameters: Vehicle ID, notification title and message.

Description: This request could be sent by any system component which needs to send a notification to the Vehicle Application. The Route Administrator receives the vehicle ID, connects to the Vehicle Database in order to retrieve the Google registration token corresponding to this vehicle, and forwards the notification to the Google Cloud Messaging (GCM) service.

Response: Contains a confirmation code.

get_passengers

Sender: Vehicle Application

Parameters: BusTrip ID, current bus stop, and next bus stop.

Description: The number of boarding and departing passengers should be sent to the Vehicle Application. The Route Administrator connects to the System Database and retrieves the corresponding passenger numbers, based on the current and upcoming bus stop.

Response: Contains a confirmation code along with the requested numbers of passengers.

get_traffic_information

Sender: Vehicle Application

Parameters: None

Description: Traffic information should be sent to the Vehicle Application. The Route Administrator retrieves the traffic incidents which are related to the city of Uppsala and forwards them to the Vehicle Application.

Response: Contains a confirmation code along with the requested traffic data.

set_google_registration_token

Sender: Vehicle Application

Parameters: Vehicle ID and Google registration token.

Description: The Route Administrator stores the Google registration token, which corresponds to the mobile device of the Vehicle Application, to the Vehicle Database.

Response: Contains a confirmation code.

Technologies

MochiWeb is an Erlang library for building lightweight HTTP servers. Based on MochiWeb, the Route Administrator could be considered as a set of processes categorized into workers and supervisors. Workers are responsible for serving incoming requests, while supervisors supervise workers by making sure that they finish their tasks and are restarted in case of errors.

Emysql [34] is an Erlang driver towards MySQL databases. The Route Administrator utilizes Emysql while connecting to the Vehicle Database. A connection pool is used, so as to allow multiple workers to connect to the database in parallel. Moreover, prepared statements are supported in order to avoid SQL injection queries.

PyMongo [13] is a Python distribution containing tools for working with MongoDB. It is used by the Route Administrator while connecting to the System Database.

ErlPort [20] is a library which allows Erlang to connect with other programming languages. It is used by the Route Administrator while connecting with PyMongo.

4.3.9 Look Ahead

General Description

The Look Ahead (LA) module is a genetic algorithm (GA) implementation and is an indispensible function of MoNAD. The objective of the LA module was to generate a timetable for a specific day by taking into account the trip requests from the same day of the previous week. The module's main assumption was that the requests' distribution on the same days of the week will be similar.

A model was designed to minimize the passengers' average waiting time and number of trips the bus company allocated for a particular line per day. The scheduling of buses was approached as an optimization problem.

As a GA implementation, the LA module consists of three phases:

1. Initial population: An initial population of possible solutions (individuals), generated based on data stored in the database.
2. Selection: A fitness function evaluated each individual; some of the same individuals could still be available in future generations depending on their fitness and the selection strategy.
3. Genetic operators: Crossover and mutation operators were used to improve the search. These operators could provide better performance than a random search.

Additionally, the LA module was designed to incorporate a function which inserts the resulting timetable into the System Database and run an independent process which modifies the timetable based on weather conditions.

Integration with other modules

The Look Ahead module was connected to the System Database, which stored the optimized generated timetable. However, it was indirectly connected to other modules of the system as well, since other modules used the timetable to successfully operate. One such module was the Travel Planner, which queried the generated timetable to search for suitable bus trips.

Tools

The Look Ahead module was developed using Python 2.7, MongoDB, and Python libraries DEAP and SCOOP.

- **DEAP (Distributed Evolutionary Algorithms in Python)**
  DEAP is an evolutionary algorithm framework that can be used to develop customized evolutionary algorithms. DEAP is distinct from other frameworks because is not limited by predefined types; therefore, custom types can be created to represent the individuals of a population [29].

- **SCOOP (Scalable Concurrent Operations in Python)**
  SCOOP is a Python module that allows concurrent parallel programming on several environments, from heterogeneous grids to supercomputers. The philosophy behind this module is focused on performing parallel tasks in a simple way [30].

Implementation Versions

1. No frequency
2. Time slices

Implementation Version 1: No Frequency

The objective of the first implementation was to comprehend the libraries to be used during development. The initial implementation did not consider many technicalities, but defined a flow of the process from gathering the requests to inserting the timetable into the System Database. The main issues during the implementation of version 1 included constant changes in the System Database design and generating reliable test data.

- **Initial Population**
  During implementation, the process was split into different tasks, such as initial population, selection, and genetic operators; however, many dependencies were found during the experimental design phase. For example, chromosome encoding must have occurred prior to initial population generation, which must have occurred before individual evaluation.

As part of the fitness function during individual evaluation, genotype-to-phenotype mapping was performed. If the chromosome encoding was not of great quality, the quality of the genotype-to-phenotype mapping would not be either, since genotype-to-phenotype mapping had a direct dependency on chromosome encoding.

Encoding a number of parameters on the chromosome defines the search space as well. Each parameter could be considered to be a dimension on which to search. The concept was to randomly propagate individuals all around the search space, so the initial population generation had to be carefully planned to avoid getting stuck at local minima.

The chromosome, an array of arrays, was modeled as follows:

[bus line, capacity, random starting trip time]

Bus line was an integer. It represented the bus line id, as it was stored in the database. At the time, experiments were being conducted with only 6 bus lines.

Capacity was an integer. It represented the maximum number of passengers that a bus could carry. Its value was randomly chosen from an array that had the following values: 20, 60 and 120.

Random starting trip time was a time structure. Its format was hh:mm. It represented a random generated time over the day. Its value was randomly generated from a function that randomly selected an integer between 0 and 23 for the hours and an integer between 0 and 59 for the minutes.

DEAP's toolbox was used as a class to register data structures. For instance, it allowed for registering chromosome attributes and initializing a population by easily creating multiple individuals with similar attributes.

- **Selection**

  The selection phase focused on individual evaluation. The fitness function evaluated each individual of the population at a given generation. An individual consisted of a set of chromosomes, also referred to as genotypes.

  The first step was to convert the genotype into a phenotype. The phenotype was evaluated by the fitness function since it contained the whole representation of the problem's solution. The phenotype represented a whole trip on a line starting at a random starting time, to be carried out by a bus with a particular capacity. The genotype was an array containing the name of the bus stop and the time the bus would be at that bus stop.

Eventually, integration with the database became possible, and generated data was used. The first computational challenge was querying user requests stored in the System Database. During this stage, the focus was on limiting processing cycles and network overhead when connecting to the System Database. The solutions implemented to handle these two issues were query aggregation, decorator functions, and on-memory indexing.

Query aggregation had a great, positive impact on performance since several calls to the database were replaced by only two request aggregate queries for the whole day. The difference between them was one grouped requests by starting bus stop, and the other grouped requests by ending bus stop. Nonetheless, both used the request time as a grouping condition.

Python decorator functions work as a wrapper which ensure a function is executed only once. So, functions which executed request queries were placed at the very beginning, defined with a decorator. Thus, the whole day's worth of requests to be processed were available on memory after the function was called, which reduced disk reads.

However, a linear search was still being performed on both query results, so performance was not as good as expected. By using on-memory indexing, requests were queried in a smarter way. The main idea behind the index was to store the array position for each time of the day. This way, it was possible to directly access requests that were going to be evaluated.

The next step was to calculate the average waiting time based on bus capacity. Since the request data had already been queried, a counter that kept track of the current amount of passengers was updated at each bus stop. Every time the counter was higher than the bus capacity, the difference between the two values was stored in an array, as well as the time until the next trips arrived. By calculating the product of these two factors, the total waiting time was calculated for these passengers.

Next, the waiting time for passenger requests going into each bus stop on that line, at that particular time was obtained. All time differences were accumulated for all individuals at every bus stop. Then, the accumulated time difference was stored in minutes. Thus, the total waiting time was defined as the summation of the total waiting time based on bus capacity and the total waiting time for passenger requests.

Next, a function assigned a cost to each bus trip depending on the bus capacity and

mapping the total waiting time into cost; the function returned the fitness value for an individual, expressed in Swedish Krona.

Finally, the SCOOP library was used when executing the fitness function since it is a process that can be done independently for each individual; there was no need to process each individual of the population in an ordered sequence. This way, each thread took care of a number of individuals in parallel. Adding the SCOOP library was the latest performance improvement made during this stage, and it provided scalability as well.

- **Genetic Operators**

  In this section, genetic operators such as crossover and mutation are explained. The LA module used DEAP's single point crossover implementation and a custom implementation for mutation. Both operations were performed when a random generated number was greater than a predefined threshold.

  1. **DEAP's Single Point Crossover**

     Crossover was used in order to generate a new pair of individuals from two individuals that already had good fitness values. The assumption here was that swapping information between fit individuals would generate individuals that would have an even better fitness value and would guide us to the global minima. From a natural evolution approach, this resembles inheritance. Single point means that both original individuals were divided at the same position and only at that point. This could change in the situation where individuals have different lengths; that was not the case for us however.

     This function accepted two individuals as input. First, it checked the individual's lengths and chose the shortest one. Then, it selected a random integer from position one to the previously defined length minus one. Next, it swapped a subset of individuals "a" with the corresponding subset of individuals "b" and vice versa. Finally, it returned both modified individuals.

  2. **MoNAD's Custom Mutation**

     Mutation, on the other hand, introduced the concept of injecting random information. The main reason for doing this during the search was to allow for exploration. In that sense, mutation was a useful tool when the search may have gotten stuck on a local minima. However, it could have also destroyed fit individuals by injecting

information that was not meaningful or it could even create unfeasible individuals. Taking all these considerations into account, it was decided to implement a mutation function from scratch.

The function received an individual. Next, it selected a random integer from one to the length of the individual minus one, which was where the mutation would take place. Next, it generated new random starting trip times and capacities by calling functions used on the initial population generation. Then, it assigned the new values on the specified position of the individual. Finally, the modified individual was returned.

Implementation Version 2: Time Slices Approach

- **Initial Population**

  The starting process of the genetic algorithm was to initialize and create a population. This meant that several individuals, which represented a possible solution to the problem, were created. The format of an individual for this particular problem (i.e. the encoding of the individual) was an array of arrays. An individual in the population consisted of several genes; however, in this approach the length of the individual was fixed. The size of the individual was based on the number of bus lines multiplied by the number of time slices. At the time, the individual size was 42 because only 6 bus lines were considered along with 7 time slices.

  ```
  Individual Size = Number of Bus Lines x Number of Time Slices
  ```

  ```
  Individual Size = 6·7 = 42
  ```

  The chosen time slices were 3 hour slices starting from 3am until midnight. This left out a time slice as can be seen in table 4.8, from midnight until 3am, because it was assumed that the bus service would not operate within those hours, but the time slices could easily be changed.

  [bus line, capacity, frequency, a random starting trip time(from each time slice)]

  Each gene within an individual consisted of:

  1. Bus line number
  2. Bus capacity, randomly chosen number between 20, 60, or 120.

Time Slice 1: 3:00 - 5:59
Time Slice 2: 6:00 - 8:59
Time Slice 3: 9:00 - 11:59
Time Slice 4: 12:00 - 14:59
Time Slice 5: 15:00 - 17:59
Time Slice 6: 18:00 - 20:59
Time Slice 7: 21:00 - 00:00
Non-Working Hours: 00:00 - 02:59

Table 4.8: Time Slices

3. Frequency of a trip (in minutes), randomly chosen number between 5-30 minutes.
4. A starting time for a trip, randomly chosen within each time slice.

Initially, when the experiment was conducted, only a single bus line was considered: UL's bus line 5, running from Sävja to Stenhagen. However, multiple lines were quickly considered and the experiment was carried out with six bus lines, which can now be extended to as many bus lines as the number preferred by the bus company. Such a change would only result in the size of the individuals to be larger than it is currently.

The following is an example of the encoding of an individual for finding an optimised timetable for bus line 2 and 5:

[[2,60,10,03:15], [2,120,3,06:15], [2,60,5,09:15], [2,60,20,13:15], [2,120,10,17:15], [2,60,8,20:15], [2,20,30,23:15], [5,60,10,03:15], [5,120,3,06:15], [5,60,5,09:15], [5,60,20,13:15], [5,120,10,17:15], [5,60,8,20:15], [5,20,30,23:15]]

- **Selection Operation**

  The chosen selection operation for this GA was Tournament Selection. In Tournament Selection, K number of individuals were chosen from the initial population using K number of tournaments of N number of individuals participating in the tournament. The winner of each tournament, i.e. the individual with the best fitness score, was selected for crossover.

- **Genetic Operators**

  1. **One-point Crossover**

     The next step of the Look Ahead function was to perform a one point crossover on the individuals that were chosen by the selection operation. The one point crossover randomly chose a point on two individuals and swapped the data of each individual from that point on. This produced two new offspring of the same size as the parent.

2. **Mutation**

   The mutation function was responsible for applying modifications to a particular gene of an individual. It randomly chose a gene and changed the bus capacity value or starting trip time. The mutation function nudged the starting trip time in a particular gene by increasing or decreasing the value slightly. To prevent the starting trip time of the particular gene from exceeding the time slice it was in, the amount of change was kept within the size of the time slice.

- **Fitness Function**

  The objective of the fitness function was to evaluate individuals in an population; the more fit an individual, the higher its chances of survival to the next generation. The individuals of the Look Ahead function were evaluated based on the bus capacity, the starting trip time for each time slice, and the frequency of a trip within a time slice. All of these factors affected the average waiting time for a user in some way, which will shortly be discussed.

  An individual was a solution to the problem, hence it was a timetable for the entire day for all bus lines. The goal was to find the individual with the lowest average waiting time, which would be the best individual found. As previously mentioned, 7 time slices existed, therefore a starting trip time had to be randomly selected and allocated in the gene from each time slice for each bus line. Since the individuals were sorted by time slice and bus line, each gene within an individual contained encoded information about a particular bus line, starting with a random time selected from the first time slice until the last time slice.

- **Algorithm**

  Initially, a query to the database was executed to retrieve and count all requests made for each time slice for the required date. As the function iterated over each gene in an individual, the total average waiting time was calculated in the following way.

  Starting with the bus capacity, the total number of requests made for that particular time slice was compared and regarded as the initial crew of passengers. This value was the total number of passengers interested in travelling within that particular time slice. A random bus capacity was allocated to the specific gene; to ensure that it was a suitable capacity for the particular time slice, the total number of trips made within the time slice

was calculated.

The value representing the total number of trips was calculated by using the frequency encoded within the gene and the random starting time allocated in the gene for a trip as follows:

- If a particular gene had a randomly selected starting time of 3:29am and a frequency of 30 minutes, it would indicate that there would be an available trip every 30 minutes for the first time slice for the particular bus line
- Going back 30 minutes from this random starting time would imply that there would be a trip at 2:59am, occuring during an invalid time slice since the first time slice begins at 3:00am. Therefore, the actual starting time for the particular bus line for this individual would be 3:29am.
- Since no trips could be scheduled before 3:29am, the next step in the algorithm was to calculate the number of trips that could be made until the last hour of the time slice, which in this case was 6:00am. This was done by subtracting the random starting time from the last hour of the time slice and converting the value to minutes.
- This value was then divided by the frequency number and the rounded value of the result was considered. The calculation would then indicate that 5 additional trips with the same route would be made, in total 6 trips for the particular bus line during time slice 1.

The next step was to calculate the total capacity of the particular bus line in the time slices. This was done by simply multiplying the encoded bus capacity in the gene by the total number of the trips, which was found using the aforementioned calculation. Assuming that the particular gene had a bus capacity of 60 encoded in it and knowing that the total number of trips was 6, the total number of bus capacity would be 360 for the particular time slice.

To evaluate whether the total capacity was a valid number or not, the total bus capacity for a time slice was compared to the total number of requests made within the same time slice. If the total number of requests, i.e. the total number of users who wanted to travel within this time slice, were more than the total bus capacity, there would be leftover passengers. The fitness function would punish this by extending or increasing the average waiting time, assuming that all leftovers would have to wait for the next trip in

the same time slice thus increasing their waiting time. This would result in the particular individual being less fit to the problem, which would indicate that the particular bus capacity encoded in the gene was not well suited for the particular time slice.

The second phase of the fitness function was to evaluate whether or not the randomly allocated trip's starting time for a particular time slice would meet the user's requirements. The objective here was to find a trip's starting time for each time slice and bus line that would result in the average waiting time for all users that wanted to travel within this time slice to be as short as possible.

Each gene in an individual has, among other information as previously stated, a randomly allocated starting trip time. This starting time was be the actual time the bus left the initial bus stop for a particular bus line. To evaluate this, a phenotype of the gene was made; this means that the database was queried to retrieve all the necessary information for the particular bus line, such as the name of all the bus stops to pass through, as well as the time taken from one bus stop to the next. This would result in acquiring all starting times for each bus stop that the bus would serve for that particular bus line in the gene.

As the fitness function iterated through each of these bus stops, it compared each request to the bus stop and calculated the waiting time to evaluate the quality of starting time in the gene, whether or not it was good for the specific time slice.

When evaluating the randomly allocated trip starting time, two scenarios could occur. In the first scenario, a particular user request time for the bus stop was made after the allocated starting time for the bus stop in the gene, i.e. the user would miss the trip. In the second scenario, the request would be made before the starting time for the bus stop, i.e. the user would not miss the trip, and instead could ride the bus.

1. **First Scenario:** Users who would miss the trip (user request time was after the trip starting time in phenotype)

   Using the frequency encoded information in the gene, the fitness function was used to calculate the next estimated starting time for a particular bus stop in a bus line. Assuming that the starting time in the gene was 3:40am, this would imply that the bus would start its trip at 3:40am, i.e. the departure time for bus stop A for the bus line would be 3:40am. From the database, the fitness function would retrieve the time it would take for the bus to reach the next bus stop, all the way until the last

bus stop of a line. If it would take the bus 5 minutes to arrive to the next bus stop B from the initial bus stop A, the fitness function would estimate that the starting time for bus stop B would then be 3:45am.

To calculate the waiting time, the difference between each request and the request's particular bus stop start time was calculated and added to an array called `totalWaitingMinutes`. In order to estimate a more accurate time difference, the fitness function would find the correct trip the user would be able to take. Assuming a user request was made at 5:30pm at bus stop B, instead of having a waiting time of 2 hours and 15 minutes, the fitness function, with the help of the frequency, would calculate the actual trip the user could take which in this case will be at 5:45pm, assuming the frequency encoded in the gene was 30 minutes. The user's waiting time would be reduced because the user would then have a waiting time of 15 minutes instead of 2 hours and 15 minutes. The waiting time would then be appended in the `totalWaitingMinutes` array and the same process would be repeated for each user request.

2. **Second Scenario:** Users who will make the bus (user request time was before trip starting time in phenotype)

   This scenario is similar to the first scenario; however, instead of allowing the fitness function to calculate the upcoming starting time of the bus stop, the frequency was used to find the appropriate time the user can take the bus; it calculated the previous appropriate departure time from the encoded departure time of a bus stop. Assuming that the user request time was 3:00pm and the encoded departure time for the bus stop was 3:20pm with a frequency of 10 minutes. Instead of having a waiting time of 20 minutes, the fitness function would calculate the appropriate time the user could actually take the bus which, in this case, would be 3:00pm, making the waiting time zero because it would know that there would be a bus in that bus stop every 10 minutes with the help of the frequency value.

   Interestingly, in this version of the fitness function, the appropriate departure time was simply found by applying calculations in the following way allowing the genetic algorithm to be executed much faster:

   (a) Found the difference between the user request and departure time.

   (b) Converted the difference to minutes.

(c) Found the number of times to go forward or backwards on the clock by dividing the difference in minutes by the frequency value and taking either the floor or ceiling value, depending on whether the user request was after or before the encoded departure time.

(d) Found the total minutes to move forward or backwards on the clock by multiplying the total number of times moving forward or backwards by the frequency value, also in minutes. This would result in having the total amount of minutes to either subtract or add to the encoded departure time to find the actual appropriate departure time.

(e) Calculated the accurate waiting time for the user by subtracting the user request time from the new appropriate starting time and appended this value to the `totalWaitingMinutes` array.

Finally, the fitness function calculated the average waiting time. To do this, it needed the total number of requests made and the total waiting time for all users including the passengers that were regarded as leftovers. The total waiting time was then simply divided by the total number of requests made and the particular individual that was being evaluated was assigned a fitness value.

Weather conditions

As part of the research, weather conditions were identified as a feature that can have an impact on the bus scheduling. Especially during winter season, when Uppsala faces snowy days and it gets very cold outdoors. So the weather conditions that were taken into consideration were snow and rain. The idea was to reduce the waiting time for users as much as possible when these particular weather conditions occur.

It was concluded that including weather conditions as part of the GA experiment was not beneficial. Encoding them would have affected the experiment in two ways:

- It would increase the search space.
- It would increase the computational work of the fitness function.

Instead, it was decided that a better approach would be to execute an independent process after the timetable has been generated. This process would get weather conditions from an external API. Then, based on this information, it would modify the timetable for the next day only during the times when special weather conditions are found. For example, if it is snowing between 5:00 pm and 7:00 pm, the timetable would only be affected during those hours.

Given this approach, it might be quite trivial to measure the adaptability of the timetable to weather conditions. This process was not part of the GA experiment, so it did not have a fitness value that could be evaluated. Moreover, the process itself was quite straightforward, and it did not affect the whole timetable. However, it could be interesting to observe how it affected the relationship on the cost since the initial premise was to allocate more buses in order to reduce the waiting time.

Implementation details

In this section, the implementation of the weather plug-in will be explained in more detail. As previously mentioned, an API was used to gather the weather conditions. Forecastio[7] was the chosen API since it provides information detailed by hour and it is free to use for the first thousand API calls a day. Currently, MoNAD only supports one city (Uppsala); so a single API call is needed per day.

The API returned an object that contained the weather information for the next day because it was assumed that the weather prediction would be more accurate if it was computed just one day earlier. Since the aim was to modify the timetable for the following day; it was acceptable to run the process a couple of hours before midnight. Next, the weather information was stored into the database to avoid extra API calls (if needed).

The document returned by the API call was read to find the defined weather conditions (snow and rain). If snow and rain were included, the timetable would be queried in order to extract all affected trips.

Then, the frequency of all the affected trips on a line were calculated. This was done by calculating the time in minutes between the earliest and latest trip, divided by the number of trips. After that, the frequency was reduced by 50%. Given this new value for the frequency,

[7] `https://developer.forecast.io/docs/v2`

new trips were generated for that time span. The final step was to update the timetable on the database with the set of new trips.

Finally, affected users had to be notified of the changes. The last step was not implemented due to time restrictions. The idea was to create an array of users' ID from the Booked Trips collection. Then, this array would be sent as a parameter to the Notification system, which would notify users about the bus rescheduling.

4.3.10 Dynamic Routes

Why Dynamic Routes?

Currently, public transportation systems with fixed lines cause companies to pay large sums of money on fuel and equipment maintenance on a daily basis. However, companies are not only concerned about the transportation cost but also about customer service. If the transportation system is able to satisfy its customers, the company will experience an increase in customer loyalty and company profit.

Cost and time management play a crucial role in the success of any business process. In order to fulfill these requirements, it is necessary to find the optimal routes based on user demand.

Explanation of Dynamic Routes

In order to ensure the decrease in cost for transportation companies and customers' travel time, the shortest possible routes are generated based on user demand and buses are scheduled according to capacity. If a large bus is to be used for a certain route, then many requests can be satisfied by a single bus. This is an attempt to consolidate the requests and decrease the number of overfull and empty buses.

In Dynamic Routes there is a set of customer requests with demands to be served. The buses are often assumed to have a common home base called the depot. For MoNAD, depot is the Central Station. The ultimate goal is to find a route which starts and ends at the depot, such that the maximum number of user requests are served in the shortest path.

Solution for Implementing Dynamic Routes

Several restrictions must be taken into account such as bus capacity, bus drivers' working hours, and that each bus route starts or ends at the depot, also known as the Central Station.

Graph

All possible public connections between bus stops must be represented in a graph before routes can be generated. The graph will contain the distance between each possible connection, allowing the cost of each connection or route to be determined and eventually minimized. The graph will help ensure that valid routes are generated between possible connections. Due to the existence of one-way and two-way roads, the graph must be a mixed graph.

Let `G (VG, EG)` be a graph in which: vertex `vi` $\in$ `VG` represents a bus stop in the city and |`VG`| = `n`; `n` : number of bus stops in the city. Edge `e` $\in$ `EG` joining two vertices `vi` and `vj`, represents the existence of a public route between bus stop `vi` and bus stop `vj`; two weights are introduced here to show the cost of traversing this edge `e`. `cij` represents the distance and time cost between the two bus stops `vi` and `vj` respectively. Networkx was used for the creation and manipulation of the graph.

Algorithm

As shown in Figure 4.14, a 2-dimensional matrix with size `N` is created, where `N` is the number of nodes (bus stops) in the city graph. This matrix is used to describe the transitions of users' requests. Each of its entries is a non-negative integer number representing a number of requests in which the users want to move from bus stop `i` to bus stop `j`. `M[i][j]` = `c`, where `i` = start bus stop, `j` = end bus stop, and `c` = number of requests.

Algorithm:

1. Get the users' requests from the database between two dates: start and end.
2. Check the requests' source and destination and increment the corresponding row and column entry in the Matrix.
3. While Max (Matrix) > 0:

   3.1 Initialize available bus seats.

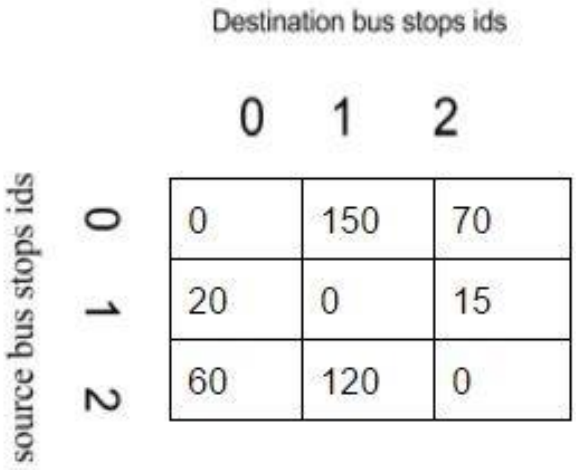

Figure 4.14: Matrix Diagram

3.2 Get the names of start and end bus stops based on the column and row of the max element in the Matrix.

3.3 Generate the shortest route from {central station, start bus stop, end bus stop, central Station} using A* algorithm.

3.4 Add routes to the routes array.

3.5 For each bus stop along the generated route:

3.5.1 Get the number of passengers that are getting off the bus.

3.5.2 Available bus seats += number of passengers getting off.

3.5.3 If (available seats > 0):

3.5.3.3 Get number of boarding passengers.

3.5.3.3 Available bus seats -= number of passengers boarding the bus.

3.5.3.3 Matrix at current bus stop -= number of boarding passengers at the current bus stop.

The route generation process is prioritized by the number of requests demanding the routes, so the maximum entry in the matrix is selected. The routes are generated using A* algorithm, which is widely used because of its performance and accuracy. The bus capacity is checked at each bus stop along the generated route by calculating the number of passengers that enter and exit the bus to ensure that all requests will be served by at least one route and the number of bus changes will be minimized. Also, bus capacity controls the number of trips needed to serve all requests between start to end bus stop.

4.3.11 Notification System

Description

The Notification System is a backend system which is used by other modules in order to notify the Client Application at appropriate times. The particular occasions for sending notifications include: reminders for upcoming trips or feedback ratings, and alerts for Travel Recommendations. Moreover, the Notification System can also be used for communication with the Application. Driver could be notified about road accidents, altered routes, or inaccessible bus stops. In addition, notifications about emergencies can be sent from the Vehicle Application; depending on the type of emergency, an appropriate vehicle could be sent.

Upcoming Trips

Notifications for upcoming trips are received 30 minutes before bus arrival. The system ensures that checks are performed every 60 seconds and reminders are sent only once, by keeping a boolean variable at the BookedTrip collection, which is true in the case that the user has already been notified.

Feedback

A similar procedure is followed for sending notifications regarding feedback. A search through the UserTrip collection is performed in order to find past trips which have not been rated (i.e., "Feedback" attribute has a value of -1). In this case, the corresponding users will be notified once, in order to provide their ratings.

Travel Recommendations

Notifications regarding Travel Recommendations are handled within the Client Application. After the list of Travel Recommendations is loaded onto the Main Activity, a check is made to see if the notification for each item has already been added. If not, an alarm is created for each recommendation; the alarms are scheduled for 30 minutes prior each bus' arrival. After the alarms are created, the boolean which indicates that the notifications have been sent will

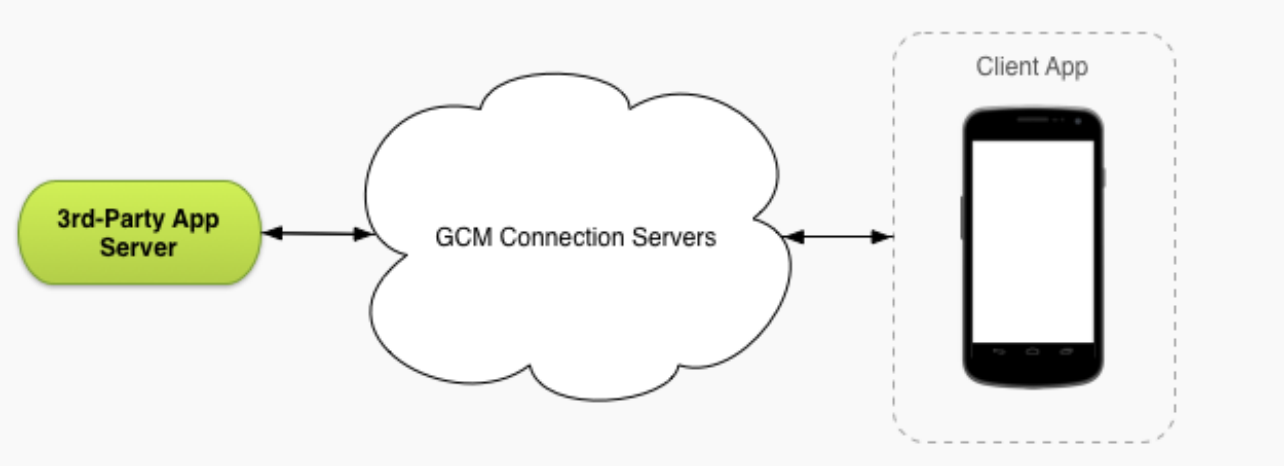

Figure 4.15: GCM Visualization [9]

be set to “True”, preventing notifications from being sent multiple times. When it is time for the notification to be sent, a check is performed in order to ensure the application is running and the user is currently logged into the application. If both conditions are met, a notification will be created and appear at the top of the user’s screen. The notification directs the user to the booking view of the recommended trip if the user press it, making bookings faster and more convenient.

Google Cloud Messaging Service

One part of this module is ”Google Cloud Messaging” service, which enables developers to send data from servers to both Android applications or chrome apps and extensions [18]. This platform is used in order to send notifications to our Client and Vehicle Applications. It is a way of extra communication when the applications are connected to the internet, even when the application is not running. The architecture is shown in figure 4.15.

Specifics

Both Vehicle and Client Applications use the Instance API from Google in order to get the REGISTRATION_TOKEN, which uniquely identifies the device; the token establishes a relationship with GCM. The token is initially stored to the User or Vehicle Database when the user registers to the application, depending on the type of user. The token value in the User Database is updated each time the user logs into the application in order to send notifications

to the correct device.

The Notification System uses either the Authentication Module or the Route Administration in order to retrieve the token. When a notification should be sent to the Client Application, the Authentication Module retrieves the token from the User Database using the User ID. The Authentication then forwards the notification to the GCM and the user receives the appropriate message. When a notification should be sent to the Vehicle Application, the Route Administrator retrieves the token from the Vehicle Database using the Vehicle ID; the Route Administrator then forwards the notification to to the GCM and the driver will receive the notification.

The Android application contacts the Google servers. It uses the specified API and subscribes a Google "topic" (/topics/). This is an organization technique for Google to choose to message all the devices in a specific topic. Within the MoNAD applications notifications are sent to users only based on the token. After registering on a topic and getting the token, the application enters a waiting state by employing a listener. If a notification arrives then the listener is able to pick it up and display an appropriate message at the top of the user's screen.

4.3.12 Simulator

Introduction

In a GAMA model, there are three sections: Global, Species and Experiment. In the global section, all the variables and actions that have a global affection are defined here, such as the simulators initialization. Attributes and actions of species will be defined in Species part which will guide species behaviors[10]. Simulators execution is also defined here along side with visualization parameters. Based on GAMA, one model that integrates data generation, data characteristics visualization and vehicle visualization is implemented. In this case, some basic information like bus stop name, request number need to be created for only one time then could be shared among different features.

Data Generation

The generated data are used to test the usability and/or performance of some backend modules. In order to have a better testing result, the generated data are tailored to mimic the real life situation.

Data sent to Request Handler should be using a unified format described as following:

`user_Id&start_time&end_time&request_time&start_position&end_position&priority&start_latitude&end_latitude`

In this project, within GAMA, global variables and parameters are defined in Global, except that, the connection to database is initiated here. When parsing requests, bus stop names and their coordinates are needed which are stored in the databases.

In Species part, one species called client is used to generate and output travel requests. There are two kinds of requests: one is requests from random passengers, which means they are generated along with random userid, the other one is requests from regular users. For the second type, we define users travel habits which has been stored in a csv file.When agent (agent is client in the context of GAMA) starts working, it looks into these habits first and generate new requests of where the attributes are almost the same as the older ones, only the date will be different. It means the new request will follow users departure and destination bus stops, only its start time and end time date will be updated. Agent identify users habits by userId and date.

Four different kinds of actions are used to define species behaviour.

1. workday
2. weekend
3. random request
4. habits request

The reason of dealing workday and weekend differently is that they have different data distribution within daily time series. A Gaussian function is introduced here to make the data fit for distribution as expected. Data distribution follows rules described below:

- workday rush hour: 6am - 9am, 3pm - 6pm
- weekend rush hour: 9am - 3pm
- bus stop distribution: weight from 1 - 10, weight of 5 is on the peak of distribution.

Bus stops are valued by a weight from 1 to 10, regarding of Gaussian distribution, the peak is in center of the range which means we use 5 for hottest bus stops and 1 for minimal ones. Some other constraints for requests follow system design, which are:

- requests only have either start time or end time according its priority
- start and end positions are mandatory as well as their coordinates
- start time should be earlier than request time

Random function is employed here to generate parts of data which should meet all above rules. All generated data will be tailored into a string and written into a csv file.

Experiment for Client Species is to define experiments parameters, initiate status and display output. It is used for implementing the species and its action.

Data characteristics' Visualisation

During the process of data generation, the statistic information about the data will be collected such as Number of passengers who want to get on/get off at a specific bus stop and the time when passengers would like to take a bus. There is also a bar chart which shows the data distribution by time or each bus stop, and the detailed information like how many passengers will take a bus at each time duration, will also be provided. While the data is being collected, the chart will be automatically updated.

The visualization is implemented to help product owner to understand better how the performance testing is done for Look Ahead module. For example, when the generated data is used to evaluate the current timetable (which is used by UL) and the Look Ahead generated time table, the employees from UL can judge if the test is valid based on the visualization of data.

Vehicle Visualisation

A pixelated map which was obtained along with the GAMA, was used to show an approximated geography of Uppsala. The bus stop was drawn along with the road on the map, and a text with the information about number of passengers was drawn next to a bus stop name. A bus icon was initially located at the beginning bus stop. Along the bus route, a set of coordinates were predetermined to be followed by the bus.

When the program is running, the bus will move along the route (coordinates), and the passengers' information text will be updated while the data is being generated.

Data Sending

The purpose of using simulator is to generate and send requests to Request Handler. Passing data to Request Handler by GAMA will be the best way to achieve this goal. Within GAMA 1.6.1, socket is supposed to be a built-in function of GAMA for sending data, but it does not work properly. After reviewing the documentation [10], this is the only option within GAMA.

Therefore, a java script is implemented here to help send the generated data to request handler. GAMA is still used without socket function is because of its good capability of visualization and data statistic. Secondly, the project time is not enough to study and use another simulation platform.

4.3.13 Background Monitoring

MoNAD is heavily dependent on the activity of users. If there are many individuals who are interested in taking the bus, it should be reflected in the number of travel requests made during a day; however, if individuals start boarding buses that conveniently suit their traveling needs without sending a travel request for those trips, future timetables will not be optimal.

One solution to this problem is to set up geofences to monitor the user's movement through certain areas. By setting up geofences at the bus stops, it is possible to detect if the user is riding the bus without making a request. The goal is not to penalize the passengers who do

not make requests to ride the bus, but to collect information in order to create pseudo-requests to ensure that the system continues to work properly.

The Background Monitoring procedure is as follows:

1. Geofences are set up. Each geofence is in the shape of a circle, with a 50 meter radius surrounding each bus stop location. In MoNAD, the bus stop ID and coordinates are uploaded when then the user logs in to the application, allowing geofence centers to be created by the Google Geofence API.

2. When a user enters or exits a geofence, the transition information is stored in a list within the Storage class. This list contains information about the time the transition occurred and which bus stop the user entered or exited. When the user logs out or closes the application, the data is stored in the System Database Geofence collection. Currently, the information is collected into the database and stored for a period of time; this version of MoNAD does not support any other features other than storing the data.

   The current Background Monitoring system is not complete. See Appendix B for more information about future work and developments.

4.3.14 User Database

Description

User Database is a relational database used for storing data related to the users of the Client Application. More specifically, it contains information concerning user profiles and settings, as well as functions for handling the requests of the Authentication Module.

Tables

User Database contains one table (i.e., **client_profile**) which is used in order to store information regrarding user profiles and settings:

Variable	**Type**	**Dscription**
id	Int	Primary key - Not null

username	Varchar	Self explanatory
pass	Char	Hashed password
email	Varchar	Self explanatory
phone	Varchar	Self explanatory
language	Varchar	User preference for MoNAD language
store_location	Varchar	User preference for location monitoring - Possible values: 0 (disabled), 1 (enabled)
notification_alert	Varchar	User preference for notifications - Possible values: 0 (disabled), 1 (enabled)
recommendations_alert	Varchar	User preference for recommendations - Possible values: 0 (disabled), 1 (enabled)
theme	Varchar	User preference for MoNAD theme
registration_date	Timestamp	Self explanatory
google_registration_token	Varchar	Used by the Google Cloud Messaging service in order to identify the mobile device of the user

Functions

User Database is consisted of the following functions, which ensure communication with the Authentication Module and serve some of the requests of the Client Application:

client_sign_up

Parameters: Username, password, email, phone, and Google registration token.

Description: Registers a new user, if there is no other user with the same username, email, or phone.

Response: A string corresponding to the following table of status codes

Response	**Case**
"01"	Username already exists - User cannot be registered
"02"	Email already exists - User cannot be registered
"03"	Phone already exists - User cannot be registered

“0”	User could not be registered due to a database exception
“1”	User was successfully registered

client_sign_in

Parameters: Username, password, and Google registration token.

Description: Verifies user registration, based on the provided username and password. Moreover, in case of success, updates the Google registration token value.

Response: A string described by the following table

Case	**Response**
Existing User	“1|id|username|email|phone|language| store_location|notifications_alert| recommendations_alert|theme”
Non-Existing User	“0”

google_sign_in

Parameters: Google email

Description: Used in order to sign in a user with a Google account. If the email already exists in the database, which indicates that the user has signed in again, then profile data and settings are returned. Otherwise, a new account is registered and a new user id is generated (google_sign_up function).

Response: A string described by the following table

Case	**Response**
Existing User Email	“1|id|username|phone|language| store_location|notifications_alert| recommendations_alert|theme”
Non-Existing User Email	“2|id”
Database Exception	“0”

client_profile_update

Parameters: Username, email, and phone.

Description: Used in order to update user profile.

Response: A string described by the following table

Response	Case
"01"	Username corresponds to another profile
"02"	Email corresponds to another profile
"03"	Phone corresponds to another profile
"0"	Database exception - Profile was not updated
"1"	Profile was successfully updated

client_settings_update

Parameters: User id, language, store_location, notifications_alert, recommendations_alert, and theme.

Description: Used in order to update user settings.

Response: A string described by the following table

Response	Case
"0"	User settings have not been updated - Possible reasons: User id does not correspond to any registered user, or a database exception occurred
"1"	User settings have successfully been updated

client_existing_password_update

Parameters: User id, old password, and new password.

Description: Used in order to update the password of a registered client.

Response: A string described by the following table

Response	Case
"0"	User password has not been updated - Possible reasons: User id does not correspond to any registered user, or a database exception occurred
"1"	User password has successfully been updated

client_forgotten_password_reset

Parameters: User id and new password.

Description: Used in order to reset the password of an existing user account.

Response: A string described by the following table

Response	Case
"0"	User password has not been updated - Possible reasons: User id does not correspond to any registered user, or a database exception occurred
"1"	User password has successfully been updated

get_google_registration_token

Parameters: User id

Description: Used in order to retrieve the Google registration token which corresponds to the mobile device of a user.

Response: A string containing the requested token.

4.3.15 Vehicle Database

Description

Vehicle Database is a relational database used for storing data related to the Vehicle Application. More specifically, it contains information concerning the vehicles of the system, as well as functions for handling requests coming from the Route Administrator.

Tables

Vehicle Database contains one table (i.e., **vehicle_profile**) which is used for storing vehicle data:

Variable	Type	Dscription
vehicle_id	Int	Primary key - Not null
driver_id	Varchar	Driver identification
pass	Char	Hashed password
bus_line	Varchar	Number of the bus line which is followed by the vehicle

google_registration_token	Varchar	Used by the Google Cloud Messaging service in order to identify the mobile device of the vehicle

Functions

Vehicle Database is consisted of the following functions, which ensure communication with the Route Administrator and serve some of the requests of the Vehicle Application:

vehicle_sign_up

Parameters: Driver id, password, and bus line.

Description: Used in order to register a new vehicle to the system.

Response: A string corresponding to the following table of status codes

Response	Case
"0"	Vehicle could not be registered due to a database exception
"1"	Vehicle was successfully registered

vehicle_sign_in

Parameters: Driver id, password, and bus line.

Description: Used in order to verify the provided credentials of a sign in request.

Response: A string corresponding to the following table of status codes

Response	Case
"0"	Invalid credentials or database exception
"1|vehicle_id"	Driver can successfully sign in, since the provided credentials are valid

get_google_registration_token

Parameters: Vehicle id

Description: Used in order to retrieve the Google registration token which corresponds to the mobile device of a vehicle.

Response: Contains the requested token.

set_google_registration_token

Parameters: Vehicle id and Google registration token

Description: Used in order to update the Google registration token which corresponds to the mobile device of a vehicle.

Response: A string corresponding to the following table of status codes

Response	**Case**
"0"	Token could not be updated - Possible reasons: Vehicle id does not correspond to any registered vehicle, or a database exception occurred
"1"	Token was successfully updated

Chapter 5

Evaluation

5.1 Current Performance

5.1.1 Look Ahead

Frequency to generate schedules

As the name of the module suggests, the main objective of the Look Ahead is to look-ahead and predict future timetables for a transportation network, assuming that similar days will have similar user requests distribution. In other words, the Look Ahead function will assume that user requests on a future Monday will somehow be similar to a previous Monday. Of course, there are other factors to consider such as weather conditions, but this will be discussed in the upcoming section.

Deciding on the frequency at which the Look Ahead algorithm will be executed and generate schedules is tricky. This is because there should always be a timetable that is available to the user in order to search and find results.

During the testing phase, a small experiment was carried out by asking 12 students whether they would make a request for a bus to take it as soon as possible, in a couple of hours, the following week, or the following day, 5 of them answered that they would want to take it as soon as possible. 6 of them preferred to take it in a couple of hours, and one person said the following day. However, none of them wanted to make a request for the following week.

Typically, users tend to make a request for the same day, usually for the next available trip, but often some other time during the day. Very few, if any, want to book a local bus for the following week.

The Look Ahead was executed with different time frequencies in order to figure out the most appropriate time frequency of generating schedules.

Initially, it was generated on the go during the day every hour or so. If the users requested departure time was for a near future time of the day, where there was yet no new generated timetable, previously generated timetable would be used as a temporary reference to find a match for the user request. As the algorithm was executed and a new timetable was generated, if there were any changes in the departure time, all users that are booked for a particular trip would be notified. This seems rather infeasible because this would have interfered with the user's plans.

As a result of this testing and evaluation, it was decided that the Look Ahead would run everyday after midnight when the bus service has finished all trips. A timetable would be generated for the same day of the week, but for the following week. On a Monday, for instance, once the day's service has been completed, the Look Ahead will look at all requests made during the day and generate a timetable for the following Monday.

Time Needed To Get "Good" Results

Experiment 1 - Cross-Over Only All experiments was carried out on requests that were generated by the simulator for the 12th of November 2015. The results come from the second implementation version of the Look Ahead called Time Slice Approach which can be found in the LookAheadVersion2 folder on Github. The following table 5.1 summarizes the results.

By increasing the number of individuals, the execution time of the genetic algorithm increases. This is because each individual is populated with encoded information to represent a solution to our problem. The advantage of having a large number of individuals is that, more solutions can be chosen for each bus line. Since during the initial population stage each bus line is allocated a time from each time slice, having more individuals implies that there are more random times allocated from a time slice to a bus line in the population.

Generations	No.Of Individuals	Time	Best Fitness Value
10	10	2 min	8 min 40 sec
50 (stuck from generation 32)	100	27 min	5 min 34 sec
50 (stuck from generation 29)	200	54 min	5 min 42 sec
50 (stuck from generation 46)	400	1 hr 49 min	3 min 51 sec
50	800	3 hr 41 min	3 min 56 sec

Table 5.1: Genetic Algorithm Experiment 1

It can also be noted that, increasing the number of individuals sufficiently, might prevent the genetic algorithm from converging to a local minimum or global minimum too fast and getting stuck as seen in table 5.1 when the experiment was carried out with 800 individuals. This could be due to the fact that more individuals implies more random solutions. However, this particular experiment was carried out without mutation; having mutation could prevent the genetic algorithm from getting stuck, which can be seen in the second experiment in table 5.2.

Finding an individual with a lower fitness value is faster with more individuals. An individual with a fitness value of 5.57 was found in the 32th generation with 100 individuals. However, it took 11 generations to find an individual with a fitness value of 5.33 with 800 individuals.

It is fair to say that having a lower limit of 50 generations is enough if the number of individuals is not sufficiently large when running the genetic algorithm with cross over only. To allow the genetic algorithm to execute and terminate with reasonable time and result, a individual size of 200 - 350 should be enough.

Experiment 2 - Combination of Mutation and Cross-Over The next experiment carried out was a combination of cross over and mutation. The results are summerized in table 5.2.

The significant fact that should be noted in experiment 2 is that it did not get stuck in a local minimum or global minimum for too long. It was seen from the experiment that there were generations where the algorithm did converge to a local minimum or global minimum and got stuck for 2 or 3 generations but due to the mutation function this was resolved.

For instance, when the genetic algorithm was executed for 10 generation with 10 individuals,

Generations	No.Of Individuals	Time	Best Fitness Value
10	10	2 min	8 min 40 sec
50	100	52 min	4 min 16 sec
50	200	1hr 45 min	3 min 29 sec
50	400	3 hr 25 min	2 min 41 sec
50	800	6 hr 30 min	2 min 36 sec

Table 5.2: Genetic Algorithm Experiment 2

in generation 7 and 8 the best fitness value that was found was 8.79 but in generation 9 due to mutation this was changed and an individual with a fitness value of 8.66 was found. The genetic algorithm continued to look for better individuals.

The genetic algorithm takes more time to terminate with the combination of the mutation and crossover operations. It can be seen in table 5.2 that it took 3 hr and 25 minutes for the algorithm to terminate with 400 individuals for 50 generations with both crossover and mutation while it took 1 hr and 49 minutes with crossover only 5.1.

Finally, the results achieved in experiment 2 were much better than in experiment 1. The genetic algorithm managed to find an individual with a fitness value of 2.60 in experiment 2 whereas the best fitness value found in experiment 1 was 3.85.

Experiment 3 - Comparison with Mutation Alone, Crossover Alone and Combining Both The final experiment that was carried out was to compare the result of the genetic algorithm when it was executed with 100 individuals for 50 generations with mutation only, crossover only and combining them.

Mutation only

Running mutation alone, results in a similar fashion to a random search because the mutation function chooses a random gene in the chromosome and inserts a random time and/or random bus capacity to it. This is why in the first generation the average fitness value started with 12.1 and decreased very slowly in the upcoming generations, reaching 8.5 as it can be seen in figure 5.1 in the last generation. The worst result is found when only mutation is used.

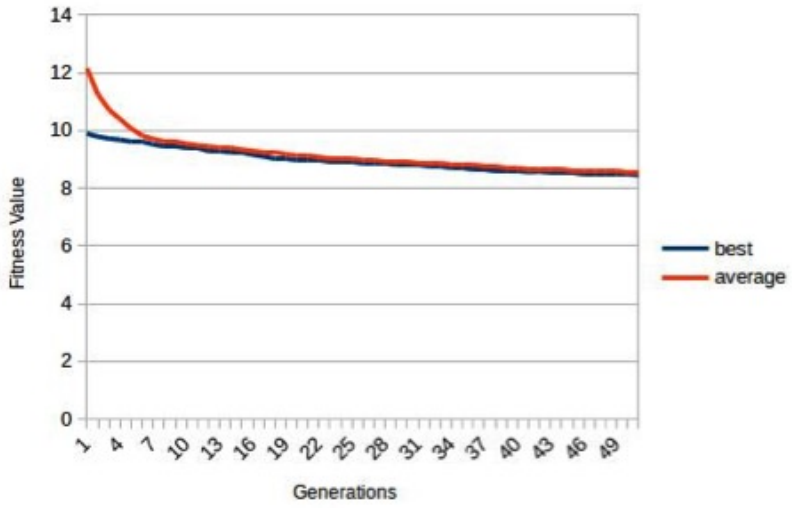

Figure 5.1: Mutation Only

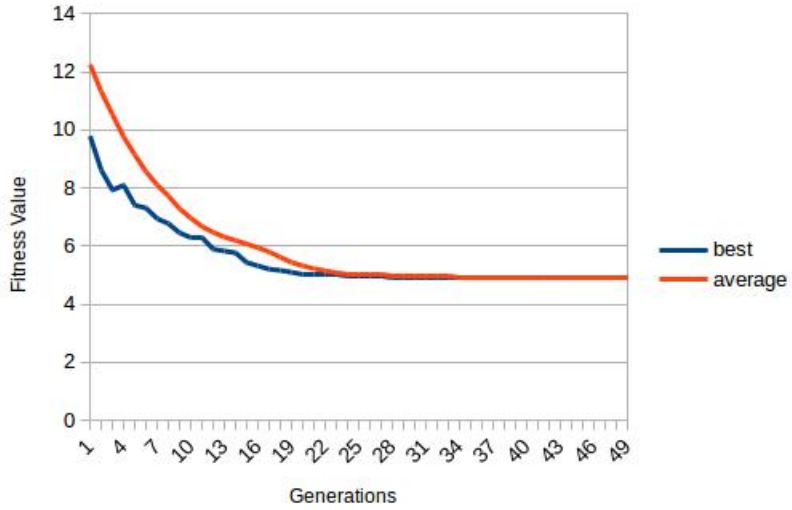

Figure 5.2: Crossover Only

Crossover only

Crossover alone performed much better than mutation alone since in crossover there is less randomness and instead of completely changing the starting times and inserting a totally random time in the genes, only the tails are swapped. The starting average fitness value was similar to mutation's but it changed rapidly in the upcoming generations and finally converged to a final average fitness value of 4.9.

The disadvantage of using only crossover is that the likelihood of the genetic algorithm to converge to a local minimum or global minimum is high. Once it converges to a local or global minimum it gets stuck there for the rest of the generations as no other better individual is

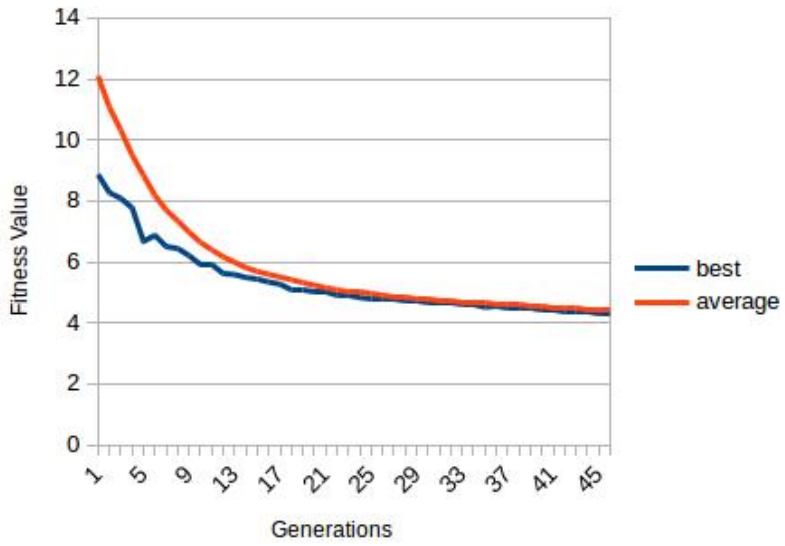

Figure 5.3: Combination of Crossover and Mutation

found. In this particular experiment the genetic algorithm got stuck from generation 38 to generation 50 since it only found an individual with an average fitness value of 4.9 as seen in figure 5.2

Combination of Crossover and Mutation

Having a combination of both the mutation function and crossover operation resulted in giving the best performance. Although it took more time for the genetic algorithm to terminate, the result that was found was much better than any of the other two approaches. It could converge to a local or global minimum, however it only got stuck there for at most 2 or 3 generations. The mutation function resolved this issue as it modified the chromosome randomly and the genetic algorithm managed to continue to find better individuals.

This approach gave the best result, a fitness value of 4.25 as seen in figure 5.3, and is the recommended approach to use for this genetic algorithm.

Finally, it should also be noted that since the algorithm is being deployed on 4 workers over 1 host, the performance is rather slow but by having a better machine, the genetic algorithm will definitely speed up and the time will decrease for the genetic algorithm to terminate.

5.1.2 Travel Recommendations

Accuracy

It is important for the reliability of this module that the recommendations closely match what the user usually requests for. For this reason, the Trip Recommendation (TR) module handles the data that represent the requests properly, by performing the correct transformations so as to normalize them and bring them under the same scale. Another parameter that is taken into consideration to ensure that the recommendations fit the users habits accurately, is a suitability score for each potential recommendation. Therefore, it will be given back to the user as a suggestion only if it is below a certain limit that is defined by the module.

An example is shown below. The user in this example has followed four habits during the last month:

- Centralstation to Polacksbacken between 9.15 - 9.30, every day
- Polacksbacken to Centralstation between 18.30 - 18.45, every day
- Stadshuset to Ekonomikum between 20.00 - 21.00, once per three days
- Ekonomikum to stadshuset between 22.00 - 23.00, once per three days

Based on these past requests the TR module, after identifying these habits, will give back to the user the routes of the current timetable that will be considered as best matches, as it is shown in Figure 5.4.

Performance

Implementing a module that will be able to perform and provide results in a reasonable amount of time is also essential for the whole functionality of the system. The proposed design of the Trip Recommendation was tested using Apache Spark 1.4.1[1] on a standalone cluster mode, using a single 2.6 GHz Quadcore, 4GB RAM machine, which is connected locally with the

[1] `https://spark.apache.org/releases/spark-release-1-4-1.html`

Figure 5.4: Recommendations' accuracy

Number of users	1	10	20	30	100	200	500
Time (sec.)	36	84	141	200	583	1005	2455

Table 5.3: Performance of module for different numbers of users

database. The test involved 6 bus lines in total and the variable was the total number of users/clients to be served.

The results for different numbers of users are shown in table 5.3.

By plotting these results, it is apparent that the performance time increases linearly as more users are served by the module. This plot is shown in Figure 5.5.

To conclude, the TR module fulfills its goal of recognizing and extracting habits out of users requests if they exist. This is done with proven accuracy, especially when users follow certain patterns every day or week. In terms of performance, an adequate number of users can be used by only one machine, although there is clearly room for improvement as the tests were held for a limited number of lines per day. Adding more machines to the local Spark cluster could decrease the time needed by the system to perform.

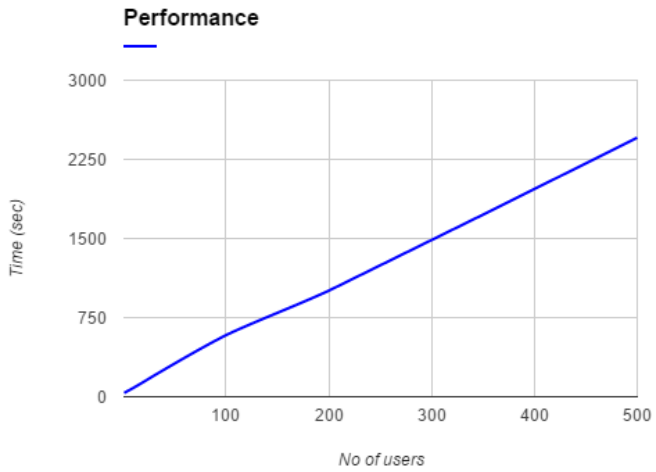

Figure 5.5: Recommendations' performance

5.1.3 Request Handler

Load testing was performed on the Request Handler before integration with the system to measure how many requests the server can handle simultaneously. Apache JMeter [15] was used on four different nodes to simulate HTTP POST requests and every response from the server with a code other than `200 OK` or taking longer than two seconds was considered a failure.

The stress test showed that the server can handle about 500 requests per second with a 0% error rate using the available hardware provided by the university. It should be noted that these results do not represent the system's performance as a whole, since the various calls of the server to different modules/servers after integration limited its performance.

5.1.4 Travel Planner

Evaluation

As one of the most important parts of the system the performance of the Travel Planner (TP) is essential to not just the other modules, but also it is directly linked with the customer satisfaction. For example, if the TP would take a very long time to calculate the best routes for a request, users would get annoyed and probably either complain about it to the company or just refuse to use the application and maybe even use a competitor's system. To avoid such

an unwanted event, the performance of the TP was closely monitored. The idea was to make the user feel that the system responds fast enough that he does not have time to switch to another application on his phone during the wait for the response. Furthermore, he should not feel like he is waiting for the system to conclude its work. Based on these rough guidelines the maximum round trip time starting at the point the user sends his request and ending at the moment he sees the results was determined to be around two seconds. Since the network delays cannot be predicted there must be enough of a time buffer to cover for this.

Testing

Besides the basic UnitTests to ensure the functionality, the key testing factor of the TP was the runtime of the algorithm. However, there are a few aspects reducing the significance of the here shown values:

- Number of bus lines: Due to the low count of bus lines in the test system it is not possible to say how well the current algorithm of the TP will perform in a complete system with maybe 40 or more lines.
- Availability of computational power: All time values given here are based on a single machine with a 2.6 GHz Quadcore and a local database. It might vary on different hardware sets.
- Python version: While the TP is compatible with both Python versions 2 and 3, due to the improved performance, the TP executes much faster in Python 3.

The goal for the required time to complete the entire algorithm on the given hardware was set to 500 ms, creating a large enough buffer for network delays. While not completely reaching this goal for Python 2, the final test results as shown in tables 5.4 and 5.5 for Python 2 and 3, respectively, are close enough to consider the TP to be in a working condition. The tables show two values for each kind of route, pointing out the influence various starting and ending locations have on the algorithm. The missing time between the parts of the algorithm and the total time contains the not mentioned parts of initialisation, processing the results and storing them in the database.

Algorithm part	Single route		Double route		Triple route		Quadruple route	
Find connection	11	41	65	130	210	218	232	269
Evaluate routes	292	106	357	324	406	400	537	327
Total time	306	174	432	462	628	628	782	611

Table 5.4: Evaluation results of the Travel Planner for Python 2. All values are given in milliseconds.

Algorithm part	Single route		Double route		Triple route		Quadruple route	
Find connection	4	28	42	89	146	143	158	199
Evaluate routes	45	16	50	54	72	61	69	52
Total time	51	47	96	147	246	209	232	258

Table 5.5: Evaluation results of the Travel Planner for Python 3. All values are given in milliseconds.

These are the example routes used to get a realistic view on the required time (lines according to the current UL system, with the modification that line 5 runs only between Stenhagenskolan and Flogsta centrum):

- Single route 1: Centralstation to Akademiska sjukhuset (either line 1, 3 or 14)
- Single route 2: Centralstation to Garnisonen (line 14)
- Double route 1: Topeliusgatan to Rosendals skola via Väderkvarnsgatan (lines 2 and 3) or via Stadshuset (lines 2 and 14)
- Double route 2: Centralstation to Flogsta centrum via Stadshuset (lines 3/14 and 2)
- Triple route 1: Studentstaden to Djursjukhuset via Skolgatan and Centralstation (lines 2, 14 and 1) or via Stadshuset and Centralstation (lines 2, 3 and 1)
- Triple route 2: Arrheniusplan to Flogsta centrum via Grindstugan and Stadshuset (lines 1, 3/14 and 2)
- Quadruple route 1: Stenhagenskolan to Polacksbaken via Flogsta centrum, Stadshuset and Centralstation (lines 5, 2, 3 and 1) or via Flogsta centrum, Skolgatan and Centralstationen (lines 5, 2, 14 and 1)
- Quadruple route 2: Arrheniusplan to Stenhagenskolan via Grindstugan, Stadshuset and Flogsta centrum (lines 1, 3/14, 2 and 5)

The different times for similar routes come from the varying number of lines intersecting at different bus stops. Due to the available test data there was no possibility of testing routes with more than two intersections and getting meaningful results. A test script to reprodue these results is included in the github repository.

To conclude, the test results show that the TP meets the requirements given by the specification and does so in a quite fast time, depending on the details of the request. However, it also displays room for improvement as detailed in section B.5.

5.1.5 Route Generator

When using the OpenStreetMap data, it first has to be parsed by the Route Generator. To load the map data it takes approximately 14 seconds. The map contains 373,640 nodes, of these are 39,497 nodes used in the road network that is used in the application. There are 685 bus stops locations, where most bus stops consist of two bus stop locations. Centralstationen has 16 bus stop locations.

Erlang server

Test for the server's part of the Route Generator was done using a 2 Quad CPU 2.66 Ghz with 3,7 GB of RAM running Ubuntu 14.04 LTS. The following three functions were tested in different ways.

- **get_coordinates_from_string** Three types of tests were done: finding coordinates from an address; finding bus stop from bus stop name; and supplying a string that is neither an address nor a bus stop. The result can be found in Table 5.6.

Test	Average time (seconds)
Finding an address	0.003
Finding a bus stop	0.007
Neither address nor bus stop	0.014

Table 5.6: Route Generator Performance test 1

- **get_routes_from_coordinates** Three tests were done: finding the route between two bus stops; finding the route between two coordinates near bus stops; finding the route

from an address, via a bus stop, to a bus stop. The following notation is used in the table: P, Polacksbacken; F, Flogsta centrum; S, Studentvägen; and ˜, coordinate close to. The result can be found in Table 5.7.

Test	Average time (seconds)
P to F	0.049
$\tilde{\text{P}}$ to $\tilde{\text{F}}$	0.370
S to F to P	0.075

Table 5.7: Route Generator Performance test 2

- **get_nearest_stops_from_coordinates** The function was tested with three different radiuses ranging from 400 meter to 4000 meters. The result can be found in Table 5.8.

Test	Average time (seconds)
400	0.016
1000	0.020
4000	0.033

Table 5.8: Route Generator Performance test 3

Bus stop graph

The test was done on a computer with 2 GHz Intel Core i7, with 16 GB of RAM, running Mac OS X El Capitan. Creating the bus stop graph takes on average 340 seconds.

5.1.6 Comparison between genetic algorithms and static timetables

The primary aim here is to see how well the genetic algorithm performs in comparison with static timetables as currently used by UL. The experiments were done with bus lines 1, 2 and 14. For the purpose of this comparison, results for bus line 2, which starts from Kungshögarna and ends at Säves väg, was used. The genetic algorithm was run for an initial population of 200 individuals for 20 generations. The results obtained using Genetic Algorithms is presented in Table 5.9 and the UL timetable in Table 5.10 for the onward journey starting from Kungshögarna.

The objective in the fitness function is to minimize the waiting time for passengers for the corresponding bus lines. Additionally, a timetable's fitness is also defined to incorporate the ability

Capacity	Headway	Departure from Kungshögarna
60	11	2015-12-10 03:16:18
120	11	2015-12-10 07:38:43
120	15	2015-12-10 09:15:14
20	20	2015-12-10 12:52:38
60	13	2015-12-10 16:24:08
60	5	2015-12-10 18:44:17
20	7	2015-12-10 21:33:38

Table 5.9: Timetable generated by GA for line 2 - onward journey

Time slice	Headway(s)
03 - 06	30, 15
06 09	10, 12, 15, 13
09 12	12
12 15	12
15 18	12
18 21	21, 22, 20
21 00	20

Table 5.10: UL timetable for line 2 onward journey

of a candidate solution to handle all the travel requests for a given time window (encoded as the bus capacity in individual). To see how well the produced timetables are, we make a comparison against current UL timetables for line 2 (`http://www.ul.se/Global/Resa/Tidtabeller_pdf/StadH2015//H2_web.pdf`). The comparison is done against weekday timetables.

The travel request distribution injected by the simulator assumed a uniform hourly distribution, while accounting for morning and evening peak hours falling at around 7am-9am and 5pm-7pm respectively. A total of 10,000 requests were used for each line, for each day. We note here that while this request distribution likely tracks the distribution as experienced by UL, the total number of requests are not expected to be similar. In comparing the timetables generated against the static ones, a deliberate choice is made for general parameters that constitute a good timetable. In particular, it would not be very helpful making the comparison based on the average waiting time as the profile of the travel request distribution chosen for the simulations may not closely track actual bus demand and hence any kind of statistical inference made would certainly be uninformative. This is due in part to the fact that we were not able to obtain data pertaining to travel requests for the city of Uppsala.

Note: the time slice 00 03 was left out of the experiments as it correlates with few/no travel requests. UL timetables typically also do not have buses at these times.

Capacity	Headway	Departure from Säves väg
60	18	2015-12-10 05:14:24
120	13	2015-12-10 08:41:52
120	6	2015-12-10 09:39:08
20	6	2015-12-10 14:52:36
60	11	2015-12-10 16:18:13
60	24	2015-12-10 18:11:45
20	6	2015-12-10 23:29:33

Table 5.11: Timetable generated by GA for line 2 - return journey

Time slice	Headway(s)
03 - 06	15
06 09	15, 12
09 12	12
12 15	12
15 18	12
18 21	21, 22, 20
21 00	20

Table 5.12: UL timetable for line 2 return journey

The results for the return journey are shown in Table 5.11, with the corresponding UL timetable in Table 5.12.

Waiting time

The generated timetables have the passenger waiting time dictated by the headway(frequency of buses in minutes). Within a time slice, the headway remains constant, but may vary in the static timetables. The passenger waiting time is thus predictable. For instance in the time slice 06 - 09 in Table 5.9, the GA timetable has a headway of 11 minutes implying that passengers will wait a maximum of 11 minutes before the next bus arrives within this time window. In contrast, the corresponding static timetable has a variable headway i.e. 10, 12, 15, 13 minutes, for a maximum waiting time of 15 minutes. From the experiments done, it was observed that GA produces a smaller maximum waiting time in some cases, but this does not carry over to all the time slices. The main difference is that the dynamic timetables assure a constant bus frequency within a time slice as opposed to the static timetables that may have buses arriving with varying frequencies. The deterministic nature of the headway produced by GA can be of benefit to the bus company as the buses will be sent with a predetermined frequency(such as within a time period of 3 hours as used in our case).

Headway and bus capacity

The headways in UL timetables change to reflect demand for buses buses leave the holding bus station less frequently when the number of passengers is smaller. The solution presented also does the same, but additionally encodes the bus capacity in every candidate solution to ensure that irrespective of that the size of bus sent, it should be able to handle most, if not all, the travel requests. The optimization seeks to fit both the waiting time and bus capacity in a an individual solution. The headways for the static timetables are not determined by the number of requests as they are constant for the same days. The GA based approach is dynamic in the sense that the solution produced will reflect variations in the number of requests as shown in Table 5.10 and Table 5.12.

Special days/events

The static timetables make no difference between days in which a sudden surge or drop in bus requests occurs and thus is unable to anticipate and gracefully handle such special days. The approach developed offers the possibility to take as input any special event/day predictions for instance from the weather API, it is possible to predict snowfall and hence adjust the timetable to account for this. This dynamic is the key difference between the timetables generated by genetic algorithms and the existing static ones.

Number of bus trips

Based on tables 5.11 and 5.12, Genetic Algorithms (GA) based search produces 131 bus trips while UL timetable results in 81 bus trips. The return journey has 139 trips and 75 trips for GA and UL timetables respectively. The observation here is that while the Genetic Algorithm approach generally results in a larger number of bus trips for a particular bus line, the numbers are comparable and the differences can be attributed to the disparity between the number of requests used in the simulations (10, 000 requests per day) and the actual travel demands experienced by UL per day.

Bus changes

A single bus change was used for all bus lines. The bus network was modelled in such a way that every bus line connected to Central station. Further, the solution developed assumes that any bus changes for passengers would take place at the single changover point, even though synchronization of bus arrivals was not part of our objective function. As currently implemented, the approach works best for passengers requiring a single bus change.

5.2 Technologies found that were better than initial suggestions

5.2.1 Travel Recommendations

For the Travel Recommendations three technologies came up during the development of the project. The first one is PyPy. It is an alternative implementation of Python and has several advantages: It runs faster than normal python programs and uses less memory than CPython. The Travel Recommendations module takes time to generate the recommendations, thus PyPy was thought as a really good alternative to speed up the generation of recommendations. PyPy is compatible with famous Python libraries like Django or Twisted, but unfortunately not with Numpy, which is a fundamental library for data processing. Thus the change to PyPy was canceled.

The second possible change that came up was about the function calculating the traveling time between two locations. There were two ways to calculate this. Initially the library geoPy was used, but later the Route Generator team created a custom function based on Open Street Maps that was calculating the time more accurately. One downside of geoPy is the restriction for using it, the library can be called only a limited number of times. Thus the Route Generators function has been tried in order to replace geoPy, however, the server providing the Route Generators functionality had severe stability issues when the function was used for a high number of times as 100. Eventually, it was decided to keep the geoPy library.

The third choice that had to be done was about the implementation of the KMeans algorithm.

There are two libraries that offer a function for KMeans. Scikit-learn, the open source machine learning python library and MLlib, the distributed machine learning framework on top of Spark Core.
MoNAD generates recommendations based on the users past requests, thus a large set of data is expected to be dealt with; this explains the distributed architecture of MoNAD. Based on this, the Spark MLlib KMeans implementation has been chosen, as it is much faster and scales better than Scikit-learn especially as the volume of requests data grows.

5.2.2 Travel Planner

The Travel Planner had just a single change, which was to make it compatible with Python 3. In the beginning of the project the decision was made to write the code for Python 2.7. During the testing phase of the project it was discovered that there was a significant difference in the performance using the respective Python versions as detailed in section 5.1.4. The code was then adjusted to work in both versions.

5.3 Comparison with existing platforms

In this section MoNAD is compared with the similar existing platforms described previously: Uber, Kutsuplus and SL.

5.3.1 UBER

Uber solves the problem of supply and demand by adjusting the price because the driver partners are independent of each other which leads to greater flexibility. However Uber system is not based on the riders habits. Their goal is to pick up a “single” rider as fast as possible. While MoNAD’s goal is to pick up as much people as possible, as fast as possible. MoNAD is a system for busses, thus the conditions and the objectives are not the same as Uber. Thats why MoNAD solves the problem of supply and demand in another way: by analyzing users habits in order to generate timetables aiming at reducing the waiting time and insuring the

availability of busses. Besides, despite the less of flexibility you have with the bus system, you get the guarantee than the fare remains the same.

5.3.2 KUTSUPLUS

To be efficient, the system needs to get as close as possible to this scenario: have enough people going to the same place at approximately the same time in order to fill the 9 bus seats and get the cheapest fare for passengers. This system is really dynamic and considers the real-time demands of users, but looking at the real life, this ideal scenario is hard to get, not because this scenario never happens but because people going to the same place at the same time do not all use the Kutsuplus mobile application. Besides when this scenario does not happen you need to wait for more time in order to the system to catch more people going to the same destination in order to fill the bus. Otherwise Kutsuplus has to practice surge prices in order to stay profitable which is not advantageous for the rider. This can explain why Kutsuplus stopped the service on the 31th of December 2015.

MoNAD is very close to this concept but does not currently enable users to get a bus on demand because it is too soon for doing it, the Kutsuplus concept is not enough mature enough. Indeed bus riders want the system to be working perfectly at all times and sporadically. That is why the MoNAD system considers the users' demands by creating new timetables everyday in order to reduce more and more the waiting time. Finally, MoNAD adds mUTSUPLUSore customization by generating Travel Recommendations based on users habits.

5.3.3 SL

The MoNAD system does not propose such a service, at least not to that extent. Indeed, MoNAD generates a new timetable everyday based on the requests made during the previous days. But "The Commuter Prognosis" algorithm would enable, if integrated in MoNAD, to create timetables every two hours which would be even more efficient than once per day but on the other hand more processing machines will be required because this system will be more expensive time-wise compared to what MoNAD currently does. Customers would thus benefit from MoNAD services with a higher accuracy. However, creating timetables every two hours

is a problem for people booking a trip three hours or more in advance, because he/she will have to take the bus in a time different that the one booked. Still, this algorithm is a really important point that needs to be reflected in the future.

Chapter 6

Conclusion

Project MoNAD is a serious effort to improve and, perhaps, revolutionize public bus transportation systems. Making the system revolve around the behaviors of its respective users is a challenging, yet very rewarding experiment that provides some interesting results for future research in the field. The prototype applications and the backend system that have been implemented over the course of the semester meet the entire list of basic requirements of the system that was originally drafted. Nevertheless, more work and efforts can and should be devoted to this project in order to transform the fully-operating prototype into a fully-deployed system that is used with real-life data and interaction. A significant part of these efforts could focus on improving the compatibility of the Android applications that are to be made available for the users and bus drivers, which, in turn, would result in a larger reach in terms of application use and data reliability. Several components of the system backend, while already operating effectively, could still be altered and optimized in order to increase their efficiency and scalability. For example, modules like Travel Recommendation and Look Ahead might need to consider more criteria in order to define the values for a few threshold values, whereas the simulator could be improved in order to overcome the limitations that were met because of bugs related to Gamma. A more detailed and technical description of possible improvements can be consulted in Appendix B.

The fact that all of the original requirements have been met does not mean that more modules cannot be considered, though. In addition to the aforementioned improvements, the system can be complemented by many features that would drastically improve the user experience, some

of which have been proposed but could not be designed or implemented due to time constraints or lack of necessary resources. Perhaps the most critical of these features is the creation of timetables using dynamic routes, which would be linked to the Look Ahead module. This approach would eliminate the need for bus stops in the city and, instead, offer ad-hoc bus stops based on the route that the bus has to take as well as on the requests made by the passengers. Another addition to the system would be the ability to adapt the timetables based on special events that occur in the city. In fact, while the Look Ahead can already use weather forecast to its advantage, it still does not take any other events into account. Consequently, it would be helpful to have the Look Ahead adapt its timetable generation process to other criteria such as holidays and cultural or sports events. These possible features and others are explained in more details in Appendix B.

Appendix A

Installation

A.1 Software/platforms required and deployment instructions.

A.1.1 Android applications

Requirements

The Client Application requires a mobile phone with Android 4.0.3 (API 15) or higher and Google Play Services 7 or higher. Besides, it should have GPS inside and able to access the Internet when using the application. While the CA can run on all devices supporting Android API 15 or higher, the optimal display of the UI would be on an Android 5 (API 21 or higher) phone, with a resolution of 1080x1920, or on xxhdpi in a more general way. The reason is that the prototype has been thoroughly tested on a Nexus 5 simulator and cell phone.

As for Google Play Services, version 7 is the minimum requirement to use features such as location, notifications, and Google login.

For the Vehicle Application, the only difference from CA is that the mobile device should be an xhdpi tablet, ideally with a resolution of 2048x1536 for optimal rendering. The prototype of the VA has only been tested on a Nexus 9 tablet, which has the aforementioned specifications.

Deployment of Android Application

Install the Client/Vehicle application: Install the appropriate apk provided.

For the Vehicle application, the file "uppsala.map" must be available under the root directory of the internal storage of the tablet.

A.1.2 Request Handler / Travel Planner

- **Install python:** `http://docs.python-guide.org/en/latest/starting/install/linux/`
- **Install pip:** `https://pip.pypa.io/en/latest/installing/`
- **Install gunicorn:** `http://docs.gunicorn.org/en/latest/install.html` (make sure to also install greenlet and gevent per the instructions there)
- **Install nginx:** `http://nginx.org/en/docs/install.html`

After installing the necessary components, configuration of gunicorn and nginx is needed.

- **Gunicorn (origin server):** Edit `serverSettings.py` and `serverConfig.py` with your desired settings (IP to bind, path of the logs, passwords, etc.).
- **Nginx (reverse proxy server):** Find the folder it was installed in (usually `/etc/nginx`) and edit `nginx.conf` according to your needs (or copy-paste the file made for the project). Then create a new `.conf` file in the `sites-enabled` folder where the settings for the forwarding of the requests will be placed (or just copy-paste the file that was configured for the project, `proxyServerSettings.conf`, there). The maximum number of open files for the operating system also needs to be raised depending on your needs: `http://www.cyberciti.biz/faq/linux-increase-the-maximum-number-of-open-files/` or use the `worker rlimit nofile` setting of nginx (`http://nginx.org/en/docs/ngx_core_module.html#worker_rlimit_nofile`).

Afterwards, all the necessary files for the origin server and the modules that are called by it need to be placed in the same folder (you can create the folder wherever you want): `logging.conf`,

`planner.py`, `routeGenerator.py`, `server.py`, `serverConfig.py` and `serverSettings.py`. Placing these files in the folder is enough to install the Travel Planner as well (`planner.py`).

Everything is set and the server is ready to be deployed. Open a console and type the following:

- **sudo service mongod start**
  - Start the mongoDB service (depending on your settings this may be done automatically)
- **cd ...**
  - Change the directory to the server folder
- **gunicorn -c serverSettings.py server**
  - Start the origin server
- **sudo service nginx start**
  - Start the reverse proxy server

Make sure that your mongoDB settings match the ones in the `serverConfig.py` file and edit the `se/uu/csproject/monadclient/serverinteractions/ConnectToRequestHandler.java` file of the Client Application, and more specifically the `SERVER` variable, with your IP settings.

A.1.3 Travel Recommendation

The TR module uses mainly the Apache Spark package[35], MongoDB[17], which is a document based database and Python[36]. In the Apache Spark, the TR module uses different libraries such as MLlib for clustering the users' history using the KMeans algorithm. It also uses Geopy library for calculating the distance between two locations. Pyspark, a Spark Python API and Pymongo, a python distribution tool for mongodb are among other tools used in the Travel Recommendation module.

The steps to install and use the technologies used in the Travel Recommendation:

1. **Download and install Apache Spark 1.4.1:**
   - install latest java version - minimum req 1.8
   - download and unzip spark 1.4.1.tgz: `http://spark.apache.org/downloads.html`
   - build spark: `https://spark.apache.org/docs/1.4.1/building-spark.html`, ready to use with pyspark API
2. **Download and install Python 2.7.6:** `https://www.python.org/download/releases/2.7.6/`,`http://docs.python-guide.org/en/latest/starting/install/linux/`
3. **Install MongoDB shell version 3.0.7:** `https://docs.mongodb.org/v3.0/tutorial/install-mongodb-on-ubuntu/`
4. **Install pip:** `https://pip.pypa.io/en/stable/installing/`
5. Install numpy - **pip install numpy**
6. **Install geopy library using pip:** `https://pypi.python.org/pypi/geopy`
7. in order to use python with MongoDB,
   - **Install PyMongo:** `https://docs.mongodb.org/getting-started/python/client/`

Once the required modules and components are installed, use the following commands:

- **sudo service mongod start**
  - start the mongodb service
- **YOUR_SPARK_HOME/bin/spark-submit Path_To_Your_Folder/script**
  - change the directory to the Spark Home folder
  - locate the folder where the python script file is found

A.1.4 User Database / Vehicle Database

Both components are implemented in SQL thus, a MySQL distribution is a prerequisite.

A.1.5 Authentication Module / Route Administrator

Prerequisites

1. **Install Erlang:** `http://www.erlang.org/doc/installation_guide/INSTALL.html`
2. **Install Python:** `https://www.python.org/downloads/`
3. **Install pip (optional):** `https://pip.pypa.io/en/stable/installing/`
4. **Install PyMongo:** `https://docs.mongodb.org/getting-started/python/client/`

Configuration

- **Server Port:** By default, the Authentication Module uses port **9999** and the Route Administrator the port **9997**. Different ports could be selected by modifying the ***init*** function of the ***/authentication/src/authentication_sup.erl*** and ***/routesAdministrator/src/routesAdministrator_sup.erl*** files respectively.
- **Emysql:** In order to establish connection with the MySQL databases (i.e., **User Database** or **Vehicle Database**) the following parameters should be selected: **connection pool size, user, password, host, port, database**, and **encoding**. The corresponding changes could be made by modifying the ***start_emysql*** function of the ***/authentication/src/authentication_web.erl*** and ***/routesAdministrator/src/routesAdministrator_web.erl*** files respectively.
- **PyMongo:** The **host** and the **port** of the System Database should be selected before establishing the connection with MongoDB database. The corresponding changes could be made by modifying the ***start_python*** function of the ***/authentication/src/authentication_web.erl*** and ***/routesAdministrator/src/routesAdministrator_web.erl*** files respectively.

Compiling and Running

- **Compiling:** Both servers contain a Makefile thus, ***make*** command should be used in order to compile. Note that the latest version of Emysql is designed for the Erlang

R16B03. For this reason, there will be compiling errors in case a later Erlang version is used, such as OTP 18. In this case, Emysql should be added manually to the ***/deps*** directory and the following changes should be made to the source code: `https://github.com/deadtrickster/Emysql/commit/52b802098322aad372198b9f5fa9ae9a4c758ad1`

- **Running:** The ***start-dev.sh*** file contains commands in order to start each server thus, the ***./start-dev.sh*** command should be used.

A.1.6 Look Ahead Module

Prerequisites

1. **Install Python:** `https://www.python.org/downloads/`
2. **Install pip (optional):** `https://pip.pypa.io/en/stable/installing/`
3. **Install PyMongo:** `https://docs.mongodb.org/getting-started/python/client/`
4. **Install DEAP:** `https://github.com/deap/deap`
5. **Install SCOOP:** `https://pypi.python.org/pypi/scoop`
6. **Install Numpy** `https://pypi.python.org/pypi/numpy`

Running with SCOOP

Run the main.py file with the following command in order to execute the genetic algorithm in multi-threads.

python -m scoop main.py

A.1.7 Route Generator

Prerequisites

1. **Install Erlang:** `http://www.erlang.org/doc/installation_guide/INSTALL.html`
2. **Install Python:** `https://www.python.org/downloads/`

3. **Install Numpy** `https://pypi.python.org/pypi/numpy`
4. **Install PIL (optional)** `http://www.pythonware.com/products/pil/`
5. **OpenSteetMap data** `https://www.openstreetmap.org/export`

Configuration

- **Server Port:** By default, the Route Generator Module use the port **9998**. A different port could be selected by modifying the ***init*** function of the ***/routesGenerator/src/routesGenerator_sup.erl***.
- **Map (server):** By default, the map used is **UppsalaTest.osm**. Another map can be selected by changing ***the_map*** variable in ***/routesGenerator/src/python/rooter.py***.

Compiling and Running

- **Compiling:** The server contain a Makefile thus, ***make*** command should be used in order to compile.
- **Running:** The ***start-dev.sh*** file contains commands in order to start the server thus, the ***./start-dev.sh*** command should be used.

Appendix B

Future Work

The following appendix regroups a complete list of improvements and new features that can be the focus of any eventual future work on MoNAD. It contains a thorough description and explanation of each of these suggestions.

B.1 Database

Database design of the system ensures there is high consistency and size optimization. However, one crucial element that acts as a tradeoff with this design choice is performance. Keeping a complex collection database consistent usually impacts the speed of the response, so it is strongly suggested that the collections be redesigned while keeping performance as the priority. For example, the collection TravelRequest does not necessarily need high consistency since its use in the Travel Recommendations and Look Ahead modules considers past requests in a broad way. Therefore, it would make sense to make the collection embedded inside the collection UserTrip. Some other collections will be duplicated, which will increase the size of the database significantly. However, working towards a faster response time should be more primordial than saving on data storage costs.

B.2 Android applications

The Client and Vehicle Applications are currently fully working prototypes where all the important features are implemented. However, some fine tuning needs to be done before deploying them.

B.2.1 Language support (CA)

It was mentioned in Section 4.3.1 that 6 languages are already supported by the Client Application: English, Swedish, Chinese, French, Greek, and Norwegian. However, the application should also support languages that are more appropriate to the region where it is deployed (i.e. Sweden). Therefore, languages such as Danish, Finnish and German should be considered for geographic reasons. Additionally, more languages could be supported based on Swedens immigrant population. The procedure to add the language is already described in Section 4.3.1.
The Settings Activity has a third tab, the Theme tab, which will allow the user to change the color of the application. Currently, changing the settings in the Theme tab does not make any difference in the application appearance but it could be implemented in the future.
Another issue that should be handled is a bug that has to do with the title of the activities on the CA; when the user changes the language in the *Settings* activity and navigates to a different activity, the title of that activity might not be translated like the rest of the text. This only happens in the case where the activity has been started before the user changes the languages, therefore restoring its state. After trying to determine the origin of the bug, it seems that the function `onResume()` in all activities (or the parent class *MenuedActivity*) should be altered to update the title field before displaying its content again.

B.2.2 Theme change support (CA)

Since user experience is not the prime focus of this project, the Client Application comes with a single default theme with a light background and with light blue as its main color. In order to appeal to the majority of the potential users of the application, at least one more theme should be available (i.e. dark theme) where the background color would be black or dark gray. The theme(s) can be added to the file `styles.xml` under the folder `values`, in a format similar

to the code below. The theme changes can then be handled in `SettingsActivity.java` by assigning the value selected by the user to the active theme.

```
<style name="DarkAppTheme.Base" parent="Theme.AppCompat.NoActionBar">
    <item name="colorPrimary">@color/primaryColor</item>
    <item name="colorPrimaryDark">@color/primaryColorDark</item>
    <item name="colorAccent">@color/accentColor</item>
</style>
```

B.2.3 Alerts & notifications

Currently, the alert settings provided to the user only allow them to choose whether they would like to receive recommendations and reminders about upcoming trips. This is very limited and means that the CA might send automated reminders at inappropriate times for the user. One possible improvement would be to list a few suggestions that the user could check or uncheck. The Vehicle Application also needs more work to make the emergency notification system work. While the UI is available, submitting the emergency report to the server is currently not handled.

B.2.4 Anonymous users (CA)

The original requirements of the project requested to provide the option to the users of using the CA without logging in. This feature has not been implemented due to the system security challenges this would imply, but a tentative design has been made. If the user chooses to proceed anonymously, the CA should allow them to perform all of the important actions such as searching for a trip, which in turn would provide the Look Ahead with more requests to consider.

On the other hand, any feature that is tailored to a users profile will be under default configuration, and the Background Monitoring module shall be completely disabled. The application language, for instance, will be the same as the system language if supported, whereas the theme and alerts will always revert to their default state. Finally, the Travel Recommendations module could be used to generate a default list of recommendations based on recent trends. This

list would be displayed to all anonymous users as a replacement to the usual customized list of recommendations.

B.2.5 Improve Quick Search and Advanced Search display (CA)

Part of the considerations to improve the overall user experience in the CA is to minimize the number of actions a user has to perform in order to reach a goal. With that in mind, the Quick Search and Advanced Search could be merged in the same activity or redesigned in a more optimal way. For example, the user could access the Advanced Search panel by swiping it down from the Quick Search, whereas clicking on the search button would hide the list of recommendations in order to display the result list.

B.2.6 Popup dialogs (CA)

While they are generally cumbersome and affect the user experience, some popup dialogs are used in the application in order to improve safety with a few actions (e.g. confirmation, deletion). These popup dialogs have all been implemented as separate activities, which is not necessarily the most efficient approach. A quick analysis should be conducted in order to determine which ones should be changed to extend DialogFragment instead [27].

B.2.7 Real-time data map (VA)

Currently, no real-time traffic or accident information is shown on the map of the Vehicle Application. If this kind of information is added on the map, the bus driver can see if the current route is affected by traffic jams or accidents and, perhaps, slightly change the route on the fly in order to avoid trip delay.

B.2.8 Login persistence

Currently, if the application is closed unexpectedly (e.g. by the Android system to save memory, or some error occurs), the user has to login again after restarting the application. It would

save a lot of trouble for the user if the application automatically logged in again. One possible solution is to store the login information in the storage of the phone, and when the application gets started, it checks the login information and automatically logs in if the information is found. Otherwise, the user is required to login. When the user logs out manually, the login information is wiped out.

B.2.9 Security revision

Since the efforts dedicated to the CA and VA focused more on the features, it is essential to have a full revision of the security aspect. For example, the hash function that is implemented for the login function uses the obsolete MD5 [28].

B.2.10 Compatibility revision

While the Android applications currently support Android API 15 and higher, some of the UI components that were used require API 19 or 21 as the minimum version. Because the team did not have the necessary resources and time to test the applications on all versions of Android, the applications have to be revised in order to ensure full compatibility. For example, the date and time picker dialogs are only supported by API 21 and higher, so the users might witness some unexpected behavior or a crash because of such components.

B.3 Simulator

Since the simulator currently relies on an external script to send trip requests, the major improvement that can be made is to be able to send those requests directly from the Gama simulator. However, bugs and limitations in Gama make it hard to implement such a feature since it cannot convert the current system time from milliseconds to a date format correctly. One more change that could improve the simulator would be to make its execution automated based on specific times using a script.

B.4 Request Handler

The Request Handler is quite scalable; adding more origin and/or reverse proxy servers with the same application code would make it able to handle many more simultaneous connections. The reverse proxy servers would be responsible for load balancing and forwarding the requests to the appropriate origin server. Additionally, the reverse proxy server(s) should be tested in order to assess their resistance against DDoS attacks.

B.5 Travel Planner

As described in section 4.3.5, the Travel Planner can return search results that involve multiple bus changes. However, it has to consider every intermediate bus stop on each route during its search for the optimal trip. This results in a vast search space which can be reduced by marking some bus stops as intersections and only searching for further routes from these stops. Similarly to many other modules, the Travel Planner also needs further evaluation in terms of scalability under a real-life scenario. The reason is that the tests that were conducted involved 8 bus lines only, whereas a bus company like UL has over one hundred lines that should be taken into account.

Another important feature which should be implemented is the act of walking to different bus stop. Currently it is only possible to get from one bus stop to the next one by bus, even if they are just fifty meters apart. Including a walking option could improve some of the routes where the user has to wait some time for one bus to get to the next one while they are very close to each other.

B.6 Notification Module

The GCM notification system uses a very simple and efficient logic; it identifies a device by a unique ID and all communication is done in one-by-one basis between a device and the Request Handler. This presents an unnecessary load for our servers especially in the case where you need to send the same message to many recipients (e.g. in case of an accident). An improvement to

this system is what Google has identified as topics.
In our implementation, all the devices are registered under a single topic. However, Google also provides the ability to message a topic instead of GCM-code, which means that a single device can register in more than one topic and receive its respective updates. This implies that the server only sends one http-post message per topic message. For example, assuming that each bus line has its own topic, a user will automatically be registered in the topic Bus3 if they request a trip using that bus. Therefore, if bus 3 has a significant delay, the server will send one http-post message to the topic Bus3, which will be forwarded to all the users registered on that topic. This approach will move the workload to the more powerful Google servers and away from MoNADs premises.

B.7 Look Ahead

The generation of initial population has been found to be challenging for several reasons. The lack of data about Uppsala traffic is one of them. Because of this, many parameters are defined by the LA team just by trial and error. Another reason is the use of the DEAP library. The composition of the class provided by DEAP proved to be tricky sometimes because, at some point, the reference to the fitness value was lost. In addition, as a design issue, the generation of the initial population should be more flexible. For instance, the size of the individual should be modified on the fly when a new bus line is added to the database or if more time slices are defined. Also, the boundary values of the range from which the bus line frequency is generated should be generated according to the line.
The design of the Look Ahead module suffers from a small drawback when generating timetables for a certain day. In fact, the requests that are taken into account are always in the time range between 00:00AM and 23:59PM of that day. As a result, all the last trips of the day which end on the next day are not accurately evaluated because requests after midnight for those trips are not included when evaluating the last trips. This certainly impacts the accuracy of the generated timetables, but it can be solved by making the static time limit a value based on the existing timetable. For example, if the latest end time for trips scheduled on Monday is on Tuesday at 00:15AM, then the time limit will be that same value, and all of the overlapping trips will be counted as Monday trips.

B.8 Route Generator

B.8.1 Map data

Due to time constraints and prioritization, MoNAD currently uses the map of the city of Uppsala as it is in OpenStreetMap. This leads to some anomalies during the route generation process due to missing map data or duplicate names for bus stops that are across each other in the same avenue. Therefore, some efforts need to be allocated to come up with a naming convention for bus stops and make sure that all bus stops are specified as nodes in the map metadata. Another option would be to generate a customized map for MoNAD that only has information about roads and bus stops, which would increase the performance of the module when loading the map.

B.8.2 Time approximation

The calculation for the time spent on the road can be improved. Currently, the distance, speed limit and type of road are used for time approximation. However, getting a more accurate time approximation requires more parameters, and among the ones that could be considered are: weather, traffic data, time of the day, traffic lights, and the behavior of a bus (e.g. acceleration). These criteria would give a better value when calculating the time between two bus stops and therefore be able to help in generating a better timetable.

B.8.3 Search string matching

In the current implementation, names of bus stops and addresses have to be spelled correctly in order to return any results. The application should instead try to find strings that match what the users has supplied without the need of being the exact same value. For example, searching for “Akademiska sjukhuset ingng 60” should return the possible bus stop “Akademiska ingng 60-70”.

B.9 Travel Recommendations

The following improvements should be considered for the Travel Recommendations. The aim behind these changes is to increase the efficiency of the recommendations that are sent to the user as well as the reliability and accuracy of the algorithm.

B.9.1 Position-based recommendations

The main idea of the TR is to be able to identify each users habits based on departure and arrival location and time, which makes the application smarter and more tailored to the users needs, yet there is room for further improvement in this context. In particular, recommendations close in time (i.e. trips happening soon after being displayed) can be modified based on the user's current position.
The application could check whether the departure location of an upcoming recommendation is close enough to the users current location (based on GPS signal). While it is expected that users will follow their habits, there might be exceptions as well. Having this extra check will allow the TR to readjust the departure location to the closest bus stop to the users current location and, eventually, offer a new recommendation. This way the system can always offer valid suggestions and even handle cases when users break their habits.

B.9.2 Recommendations with bus changes

The limited number of lines that were used in the current prototype did not give the possibility of properly handling and testing the cases where the trip that a user has to follow needs more than one bus. Ideally, the TR should be able to check some combinations of lines that have common bus stops in order to provide the best recommendations regardless of the number of bus switches. Thus, an important consideration for future work would be to perform this check to better respond to the users demands.

B.9.3 Testing with local Apache Spark cluster

As mentioned in the evaluation part, the performance results come from a standalone cluster built in a single machine that communicates locally with the database. However, the algorithm is built in a way that allows scalability by distributing the calculations that find the score for each recommendation on one hand, and apply the k-means algorithm on past requests. Therefore, an improvement in performance is expected by adding more machines to this cluster. This way, not only more computational power can be exploited, but a lot of time could also be saved from communication with the database.

B.9.4 Modifying the score threshold based on user feedback

In the TR module, a number of factors such as weights for time and location, score threshold, the number of clusters, and the number of repetitions for k-means algorithm are used, therefore affecting both the calculation and the filtering of recommendations. These factors have fixed values that were assigned based on how satisfactory the test results of a limited number of requests and users were (due to the lack of data).
Nevertheless, there is no option for potential optimization of these values, be it for the ideal case (i.e. for each user) or at least for all the users. For this reason, a useful addition that would improve the functionality would be to apply a multinomial logistic regression model where the final outcome is the feedback that each user provides in a scale of 1-5. Based on these factors, it can predict which combination of values would achieve the highest feedback.

B.10 Background Monitoring

The next steps describe the Background Monitoring tasks that can be implemented in the future.

1. The Geofence collection data will be imported into a Comma Separated Values (CSV) file. This file, along with a JSON file containing points which define the Geofence borders, will be given as input to an external module. The output of the module will contain a list of individuals who rode a specific bus from point A to point B at a certain time.

2. The information from the module will be used to create travel requests. Information will be collected from anyone who uses the application and has enabled location services. To avoid redundancy, a search through the Travel Request collection will be made to find the passengers who have made a real travel request for the trip; a pseudo-request will not be created for them. Travel requests will only be created for the module results that did not match up with a request in the database.

As a result, the request information is provided to the system whether the user manually searches for the trip or not; however, the system only supports this feature if the user has logged in to the application and has the location services enabled. Currently, the background monitoring feature does not ensure that every person riding the bus is accounted for, since passengers may still ride the bus without having downloaded the application, logged into the application, or given location permission. These issues can be solved in other ways in the future.

Appendix C

Acronyms and Abbreviations

The acronyms and abbreviations presented in table C.1 are used in this project report.

Acronym	Description
BSON	Binary JSON
DDoS	Distributed Denial of Service
EC	Evolutionary Computation
EDT	Estimated Driving Time
GA	Genetic Algorithms
GCM	Google Cloud Messaging
GPS	Global Positioning System
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
LA	Look Ahead
MoNAD	Mobile Network Assisted Driving
MVC	Model View Controller
OSM	OpenStreetMap
RG	Route Generator
RH	Request Handler
RV	RecyclerView
SSL	Secure Socket Layer
TP	Travel Planner
TR	Travel Recommendation
UL	Upplands Lokaltrafik
WSGI	Web Server Gateway Interface
XML	Extensible Markup Language

Table C.1: Acronyms used throughout this report

Bibliography

[1] *UL website*. [Online]. Available: `https://www.ul.se/en/About-UL/About-our-public-function/` (Date last accessed Jan 7, 2016).

[2] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison Wesley, Massachusetts, 2nd edition, 1994.

[3] *Google Sliding Android Tabs*. [Online]. Available: `https://github.com/google/iosched/tree/master/android/src/main/java/com/google/samples/apps/iosched/ui/widget` (Date last accessed Jan 7, 2016).

[4] *BRouter*. [Online]. Available: `http://brouter.de/brouter/` (Date last accessed Jan 7, 2016).

[5] *GraphHopper*. [Online]. Available: `https://graphhopper.com/` (Date last accessed Jan 7, 2016).

[6] *Mapsforge*. [Online]. Available: `https://github.com/mapsforge/mapsforge` (Date last accessed Jan 7, 2016).

[7] *Osmdroid*. [Online]. Available: `https://github.com/osmdroid/osmdroid` (Date last accessed Jan 7, 2016).

[8] *Android Version Distribution*. [Online]. Available: `http://developer.android.com/about/dashboards/index.html` (Date last accessed Jan 7, 2016).

[9] *Google Cloud Messaging Architecture*. [Online]. Available: `http://www.programming-techniques.com/2014/01/google-cloud-messaging-gcm-in-android.html` (Date last accessed Jan 7, 2016).

[10] *Gama Platform*. [Online]. Available: https://code.google.com/p/gama-platform/wiki/G__Downloads?tm=2 (Date last accessed Jan 7, 2016).

[11] *Gunicorn*. [Online]. Available: http://gunicorn.org/ (Date last accessed Jan 7, 2016).

[12] *Gevent*. [Online]. Available: http://www.gevent.org/ (Date last accessed Jan 7, 2016).

[13] *PyMongo*. [Online]. Available: https://api.mongodb.org/python/current/ (Date last accessed Jan 7, 2016).

[14] *Nginx*. [Online]. Available: http://nginx.org/ (Date last accessed Jan 7, 2016).

[15] *Apache JMeter*. [Online]. Available: http://jmeter.apache.org/ (Date last accessed Jan 7, 2016).

[16] *JSON and BSON*. [Online]. Available: https://www.mongodb.com/json-and-bson (Date last accessed Jan 7, 2016).

[17] *Introduction to MongoDB*. [Online]. Available: https://docs.mongodb.org/manual/core/introduction/ (Date last accessed Jan 7, 2016).

[18] *Introduction to Google Cloud Messaging Service*. [Online]. Available: https://developers.google.com/cloud-messaging/ (Date last accessed Jan 7, 2016).

[19] *OpenStreetMap*. [Online]. Available: https://www.openstreetmap.org (Date last accessed Jan 7, 2016).

[20] *ErlPort*. [Online]. Available: http://www.erlport.org (Date last accessed Jan 7, 2016).

[21] *MochiWeb*. [Online]. Available: https://github.com/mochi/mochiweb (Date last accessed Jan 7, 2016).

[22] *JOSM*. [Online]. Available: https://josm.openstreetmap.de/ (Date last accessed Jan 7, 2016).

[23] *Movable Type Scripts – Distance Calculation*. [Online]. Available: http://www.movable-type.co.uk/scripts/latlong.html (Date last accessed Jan 7, 2016).

[24] Charles Robert Darwin. *The Origin of Species*. Vol. XI. The Harvard Classics. New York: P.F. Collier & Son, 190914; Bartleby.com, 2001.

[25] Thomas Weise. *Global Optimization Algorithms – Theory and Application*. (self-published), 2009. [Online]. Available: http://www.it-weise.de/

[26] Andries P. Engelbrecht. *Computational Intelligence: An Introduction*. 2nd Edition. Wiley. 2007.

[27] *Android Developers*. [Online]. Available: http://developer.android.com/guide/topics/ui/dialogs.html (Date last accessed Jan 7, 2016).

[28] Alexey Melnikov. *Moving DIGEST-MD5 to Historic*. [Online]. Available: https://tools.ietf.org/html/rfc6331 (Date last accessed Jan 7, 2016).

[29] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. *DEAP: Evolutionary Algorithms Made Easy*. Journal of Machine Learning Research. volume: 13. pages: 2171 – 2175. Jul, 2012.

[30] Yannick Hold-Geoffroy, Olivier Gagnon, and Marc Parizeau. *Once you SCOOP, no need to fork*. Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment. pages: 60. ACM. 2014.

[31] Ian Sommerville. *Software Engineering*. pages: 84 – 87. 9th edition. 2011.

[32] *Erlang OTP*. [Online]. Available: https://github.com/erlang/otp (Date last accessed Jan 7, 2016).

[33] Carl Hewitt. *Actor Model for Discretionary, Adaptive Concurrency*. [Online]. Available: http://arxiv.org/abs/1008.1459. CoRR, abs/1008.1459. 2010.

[34] *Emysql*. [Online]. Available: https://github.com/Eonblast/Emysql (Date last accessed Jan 7, 2016).

[35] *Apache Spark*. [Online]. Available: http://spark.apache.org/docs/1.4.1/ (Date last accessed Jan 7, 2016).

[36] *Python*. [Online]. Available: https://www.python.org/doc/ (Date last accessed Jan 7, 2016).

[37] *A* Search*. [Online]. Available: https://en.wikipedia.org/wiki/A*_search_algorithm (Date last accessed Jan 11, 2016).