Rideshare Automatic Matching System

A Ridematching Android App utilizing the aSTEP System

Project Report

SW605F16

Aalborg University
Department of Computer Science

Department of Computer Science
Aalborg University
http://www.cs.aau.dk

AALBORG UNIVERSITY

STUDENT REPORT

Title:
Rideshare Automatic Matching System

Theme:
Developing Complex Software Systems

Project Period:
Spring Semester 2016

Project Group:
SW605F16

Participant(s):
Mathias C. Mikkelsen
Bjørn E. Opstad
Morten Pedersen
Claus W. Wiingreen

Supervisor(s):
Davide Frazzetto

Copies: 1

Page Numbers: 71

Date of Completion:
May 25, 2016

Abstract:

This project was developed as part of a multi-group project with the goal to develop a platform for spatial and temporal data management (aSTEP) with accompanying apps.
Our task was to develop an application (RideShare) which utilizes aSTEP to suggest rideshare partners to users by storing gathered location data and analyzing users routes. For this, we developed an algorithm that compares two routes and returns a score based on our estimation of the potential for ridesharing.
The solution is in an acceptable state but needs a user interface refinement and a large-scale test to confirm the matching algorithm is working as intended for a bigger user basis.

Summary

During this semester, we have collaborated with nine other groups on the startup of a new multi-group project with the goal of developing a platform for spatial and temporal data management called aSTEP. The system was also required to be developed with accompanying apps. Our task in the multi-project was to develop an application that would utilize the aSTEP system API and database.

We designed and developed an application with the goal of automatically finding ridesharing candidates based on tracking of the application users commutes. The solution was also designed to automatically detect which routes a user drove was reoccurring in a weekly basis. If a stable route is found it is then compared with every other stable route in the aSTEP system to find a match. If a match between two users were found, the matches would then be displayed in the application with a score from 0 to 100.

The developed solution is fulfilling our highest prioritized requirements, but still has potential for improvement. There are requirements we have written that are not yet fulfilled during the development process and therefore not implemented. The app does, however, suffice as a proof of concept for the aSTEP system and as a foundation for further development.

The aSTEP system was developed to a stage where the API and database had enough functionality to support applications that would store locations, routes, and compute ride matches. There exists a satisfying user management system, where users can be created and managed. These users can be utilized in the applications to ensure correct authentication for the different API calls.

We have been collaborating with the other aSTEP groups in multiple degrees and with different motivations. The collaboration with the FriendFinder app group was superficial, and only to vaguely agree on the visual UI design scheme. However, we mostly collaborated with the OD group, as they were responsible for handling the location input and implementation of the route matching algorithm on our behalf in the aSTEP system. The user management group was also collaborated with, as they were responsible for the user management functionality in the aSTEP system. We discussed multiple types of functionality and privacy concerns with them, and the other app groups were also involved, as the aSTEP system should be modular and usable for multiple app types.

For this project to work, an algorithm that solved our problem was necessary. The report cover our logic behind the algorithm and how we worked together with the group that implemented it to test it.

Preface

The purpose of this report is to document the development process of the semester's project performed in the spring of 2016 by our group, SW605F16. This project was developed as part of the multi-group project AAU Spatio-Temporal data management Platform (aSTEP). Throughout the project, we gained experience working with Android development, multi-group project collaboration, and much more. The developed solution consists of three parts: an app, a server, and an aSTEP integration. These cooperate and serve to provide the app users with automatic ride sharing suggestions for their reoccurring driven routes, as a mean to reduce traffic, emissions and travel time.

We would like to thank Davide, our supervisor, for his work and support throughout the project, and Bin Yang for the multi-group project suggestion of aSTEP.

Aalborg University, May 25, 2016

Bjørn E.Opstad
<bopsta13@student.aau.dk>

Claus W. Wiingreen
<cwiing13@student.aau.dk>

Mathias C. Mikkelsen
<mcmi13@student.aau.dk>

Morten Pedersen
<morped13@student.aau.dk>

Contents

Chapter 1 Introduction

People traveling the same route in different vehicles induce slower traffic and a heavier strain on the environment [1][2]. This project embarks on the task of informing the individual commuters of other commuters traveling a similar path, so that they can share a vehicle. If this project is successful, the reduced number of cars could reduce the CO_2 emissions caused by traffic and commuting.

1.1 Project Environment

This project is part of the collaboration project, aSTEP, in the SW6F16 semester at Aalborg University. The goal of the project is to develop a location-based service with accompanying applications.

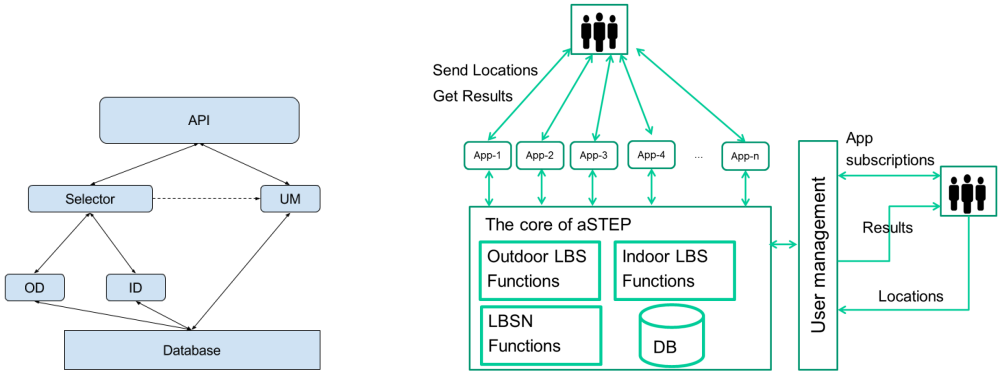
The aSTEP project was developed in ten teams with four members each over the span of four months. The project was furthermore divided into sprints with each sprint spanning over four weeks, resulting in a total of four sprints. The sprints were synchronization points of the project groups, and this report will contain documentation of each sprint chronologically.

The first sprint is intended for analysis and collaboration development, the second and third are development sprints, and the fourth is intended for testing and finalizing the development. The ten project groups are assigned to their respective tasks, with seven project groups developing the core aSTEP service, while the remaining three groups are developing applications utilizing the system.

1.2 Group Role

This section describes the multi-group project setting and defines the role of our group's work in the multi-group project.

The multi-group project consists of several components, including user management, outdoor location based services, and database groups as seen in Figure 1.1b. OD and ID in the figure are abbreviations of respectively outdoor and indoor location based services. Each group is assigned to one component and some components have multiple groups assigned to them The intended component architecture can be seen in Figure 1.1a.



(a) The aSTEP core architecture [3].

(b) The aSTEP group distribution according to the project guidelines.

Figure 1.1: aSTEP architectural design

Our project group is responsible for developing an app that utilizes the aSTEP core through the API, and contribute to the development of the interface of the core. As the project focus is on commuting, the solution will depend upon outdoor location based services. Any user registration, login or other user related tasks should be handled by the user management group. According to our project focus and the component architecture, represented in Figure 1.1a, we will mainly cooperate with the OD and user management groups.

1.3 Problem Domain

aSTEP's planned functionality gave the idea of a large set of location data which a system could use to identify and help drivers and passengers with arranging ridesharing. The goal of the solution is to provide ridesharing suggestions to the users of the app. When referring to the term ridesharing in this report, it is referred to as the action of private persons sharing a car for the whole or a part of a route. This definition is covered by the terms carpool and ad-hoc ridesharing by Chan and Shaheen [4].

The intention of this project is to develop an Android app that utilizes the aSTEP system for historical location data and arranging carpool style ridesharing. The extent of such a project can be wide and therefore we early choose that the main focus will be to design and implement a basic system, which could be expanded upon in a potential future project. The purpose of the solution is to propose appropriate candidates for ridesharing to the users of the app. The project will include developing and implementing of an algorithm that assesses if two users of the

system should be suggested for ridesharing. Other parts of the application, such as graphical user interface, are not prioritized but are developed to be sufficiently functional for practical purposes and testing.

The potential benefits are social, environmental and economical, as described in [4]. The app allows people to meet each other and thus reduce the use of cars, fuel and the load on the road infrastructure. The app could also decrease problems such as traffic jam and thereby also be time-saving, and causing less frustration for the drivers.

The future aspect of this solution seems bright since it easily should be modifiable for future transportation needs, i.e. autonomous vehicles.

To formally define the problem, a problem statement will be constructed.

1.4 Problem Statement

Sharing rides is advantageous in an economical and environmental sense, but it is made tedious and difficult due to the lack of knowledge of other people sharing the same or similar commute routes as explained in [4]. This leads to the following problem statement.

How can one design and develop an app that automatically suggests ride sharing companions, based on common locations in origin and destination, utilizing the aSTEP platform?

To successfully solve the problem statement, Sprint 1 will be initialized with analyzing the problem domain to form a list of requirements.

Chapter 2 Sprint 1

The first sprint is concerned with getting the semester project started, including both the aSTEP project as a whole and the RideShare project. Thus, the first sprint will be spent on organizing the collaboration with the other project groups and setting the direction for the project. This means that the contributing parts of sprint 1 will be the initial analysis, requirement specification and overall design, with the documentation in this chapter in the same order.

2.1 Analysis

This analysis section will contain descriptions of the material analyzed in sprint 1. The first topic is to analyze the description and definition of ridesharing. The second is to investigate the state of the art, and finding current solutions of the problem. The analysis will provide sufficient information and details to establish a requirement specification, defining the subproblems of the problem statement.

2.1.1 Ridesharing Definition

Ridesharing is an activity which has had phases of popularity in recent history according to Chan and Shaheen [4]. They describe the current phase as the technology-enabled ridematching phase which started back in 2004. In this phase, the integration of the internet, mobile phones, and social media into ridesharing services reduces the barrier to entry for new potential passengers and drivers. Chan and Shaheen [4] also lists incentives for ridesharing in the current phase, which are summarized in the following points:

- "Focus on reducing climate change, growing dependence on foreign oil, and traffic congestion
- Partnerships between ridematching software companies and regions and large employers
- Financial incentives for green trips through sponsors
- Social networking platforms that target youth
- Real-time ridesharing services"

This presents a broad overview of why ridesharing became popular again after 2004. Because technologies play a vital role in the current phase, a selection of

popular commercial services will be examined along with some scientific papers to get a comprehension of the state of the art in the field. The main focus of this project will be on real-time and dynamic ridesharing which Amey, Attanucci, and Mishalani [5] defined as "rideshare service relying heavily on mobile phone technologies".

2.1.2 Scientific Papers

The problems of automating ridesharing through modern technology has already been studied with different approaches. The following sections account for some of which the project group found the most interesting and relevant to the problem statement.

Shuo Ma et al. [6] developed an algorithm for taxi services and improved throughput of passengers by 25% and reduced the distance a single taxi had to drive by 13%, at six requests for rides per taxi. Especially interesting in their paper, is the approach of a grid representation of a road network, to avoid shortest path calculations between points on a map, and instead approximate distance based on cells in the grid. The paper states that the solution also utilizes the user's smartphone to collect location data and act as a user interface.

Ghoseiri et al. [7] researched which properties might matter when matching driver and passengers. They focused on preferences which influence whether you want to drive with a person or not. The preferences can be properties such as gender or pet friendliness. They developed functions that can assess if a passenger and driver match, based on location, preferences, passenger and/or driver detour as the most important factors [7]. This algorithm might be useful in this project solution.

Actual choices and influences regarding algorithm design for ridematching will be addressed later in the design phase, in Section 2.3.

2.1.3 Commercial Solutions

As corporate and community solutions are more economically motivated instead of scientifically, the solutions differs from the academic field solutions. The main focus seems to be centered more around taxi alternatives. The are multiple services available around the world, with the two biggest international services in this field as of 2015 seems to be Uber and Lyft[8]. There are also several services that operates in Denmark, among those are Norwegian-based Haxi and Danish-based Drivr. The two services both work in a similar fashion:

1. A customer requests a ride through a smartphone app or a web interface

2. The backend of the service sends the request to one or more appropriate drivers
3. A driver accepts the request, and dispatches

Since the mentioned services are commercial and closed-source, actual information about the design or architecture of the service is not available. Drivr has a a web interface for fleet management and business which provides administration and expense control of employee taxi travels.

Besides the taxi-like services, there are services that focus on traditional ridesharing between private people, where money is not earned. These services exist both locally and internationally and provides the opportunity to either offer a lift, request a ride and the possibility to connect drivers and passengers.

The technology used in ridesharing services, especially smartphone technology, has been evolving at a fast pace.[9] We believe that advancement in smartphone sensors is an opportunity that is still not fully utilizing its potential.

Based of the information gathered in this analysis, a list of requirements for the solution can be established and formally written.

2.2 Requirement Specification

The requirements are divided into two main categories, functional and nonfunctional requirements, and are furthermore sorted in the MoSCoW [10] structure. The structure assists in solving the core purpose of the system first, as these are the highest prioritized requirements, and later adding additional functionality.

2.2.1 Functional Requirements

The functional requirements are directly related to the tasks and operations of the developed application.

Must-have requirements
The 'Must have' requirements must be fulfilled for the solution to be acceptable.

Graphical user interface
The application must have a graphical user interface (GUI) as users must be able to use the application themselves, and to structurally and aesthetically display user matches. The GUI must be intuitive as users may have different levels of experience with using mobile applications.

User accounts, including login and registration.
Unique user accounts are required as it serves as an identifier for each user in the system. With user accounts, store personal information, and compare user based on the data. It will also provide functionality for displaying user to other users.

Communication with the aSTEP system.
As the project is a part of the bigger system aSTEP, there must be a relation to the developed platform. The communication would be storing data in the aSTEP database and using user management also implemented on the platform.

User location tracking and storage.
The application must track users as they move, to be able to determine the users travel routes. The application should store the commuted routes and the data should be stored in the aSTEP database.

Automatically determine regular routes.
The solution must automatically determine if a route is regular. Regular routes should be used for route comparison and match generating.

Automatic user ridesharing suggestions.
Automatic matching with other users routes must be provided to the the individual app users, and the matches must be computed by the solution. When two regular routes are found similar, the users must receive an indication of the match in the app. If the match computation shows to be a excessive process it is deemed to be done in the aSTEP core.

Should-have requirements
The requirements in this subsection are important, but are not regarded as critical for the functionality of the solution.

Give the user option to specify whether they have a car or not.
Because some users might not have a car, they should be considered differently than the ones with, as the users without vehicle cannot give a ride to a potential match.

Enable users to blacklist other users.
Users should be able to filter other users from the matches to prevent further possible bad experiences with either a driver or passenger.

Suggest rides with users who only drives a subset of the way from A to B.
Giving users the option to match with people who do not have the same source and destination, will increase the number of suggested matches, as a user can join a part of the matches' route.

Could-have requirements
The 'Could have' requirements are the lowest realistically fulfillable requirements,

and are implemented if the higher priority requirements are fulfilled.

Ride reservation or request from, to, time.
This requirement enables the user to reserve rides with other users.

Would-have requirements
The following requirements are only considered when all other requirements are satisfied but initially regarded as tasks to be solved in future projects.

Inform users of their environmental and economic savings due to their use of the solution.
Provide users with detailed information on how much fuel the user has saved, how much money saved based on fuel prices, and how much CO2 that is not released into the environment.

2.2.2 Non-functional Requirements

The nonfunctional requirements for the solution are stated in the following paragraphs.

Must-have requirements
These are the requirements the solution must fulfill to be acceptable.

Development cooperation with the other aSTEP project groups.
The development must be done in cooperation with the other aSTEP groups.

User privacy
The app must respect user privacy, especially in regards to a user location data and personal user data. The solution should consider securing elements such as database storage, user management, and developers.

Should-have requirements
These requirements are important, but are not regarded as highly critical.

Aesthetics matching other aSTEP project applications
The app should share design and guidelines as the other apps developed for aSTEP.

The first design phase can be initiated as the solution requirements now are specified.

2.3 Design

This section contains the overall app design, which is based on requirements with inspiration taken from the different commercial products and the papers read in

the analysis. The section will outline the main features of the design concerning system structure, components, and user interface.

2.3.1 System design

To exploit the availability of sensors on the mobile phone and computational power of the server, the system is developed in two parts: the phone application and an extension of the route system in the aSTEP system which handles the analysis and comparison of routes. The two parts will have their own delegated responsibilities, and will perform the necessary tasks, enabling the system to deliver the service.

The RideShare App
The application, hereafter referenced as the app, is the program executed on a mobile device. Its responsibilities include the gathering of location data and providing the user interface.

The task of comparing routes is assumed to require much processing power, and also be able to access location data to get the aforementioned routes. Performing the route comparisons in the app would require that each user have a copy of every other user's routes. This is a serious privacy concern as this can enable other applications to leach on to the aSTEP system a gather data about when people are home and where they live, and is an aspect we would like to avoid. As mentioned before, calculating the best route matches would most likely require a lot of processing power, which would drain the battery and produce heat.

Assuming that a central server can handle the analysis of the routes, the application is only required to supply the server with location data. This is a much better solution as the impact on the battery is reduced, and private information can be contained within the aSTEP system.

A service running in the background of the app can handle the location data sampling and transfer to a server. This service would be kept alive when the application is closed as its own thread, thus being able to record routes even when the app is not in focus on the device. Because the main objective of the background service is to observe the current activity and transfer the route to aSTEP, we call it Observer. The structure of the class can be seen in Figure 2.1. The Observer's main goal is to observe what activity the user is currently performing and receive and push locations accordingly.

The data representing the routes needs to be sent to aSTEP, the aSTEP component is documented in the following section.

The aSTEP Component
On the aSTEP server, the route is stored in a database and accessed by the RouteSta-

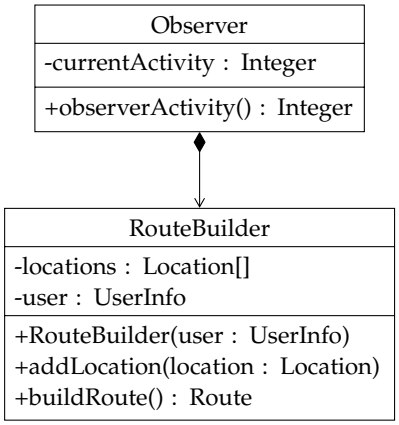


Figure 2.1: The small process which should run at all time to build the routes.

bilityAnalyser. This analyzer takes the route and compares it to other routes by the same user, to determine whether this route is a regular occurrence, and hence a stable route.

If the RouteStabilityAnalyser determines that the route is stable, the route should be marked with a tag indicating that the route is stable in the database together with values that indicate when this route is expected to occur next. The server then uses the RouteSimilarityAnalyser to find matches for the new StableRoute. The matches are stored and new matches are transmitted to the appropriate devices when possible to inform the users of new matches.

This is the first semester working with the aSTEP project which will be further developed upon and expanded in the future, emphasizing the importance of reuse in the server, to reduce the number of specialized server resources. By isolating parts that are specific to the rideshare app on the server, most of this system would be reusable. An example of how this is achieved through object-oriented design, is the RouteAnalyser seen in Figure 2.2. The RouteAnalyzer filter routes according to its concrete implementation. After the filtering, the RouteAnalyser compare the routes. This enables other projects to develop their own filters and comparison functions for later projects.

Because commuters were our target and commutes are most likely to occur on a weekly basis, a weekly stability was decided upon. This means that the RouteStabilityAnalyser can be limited to analyze only two months of data, as the route can be stopped driven.

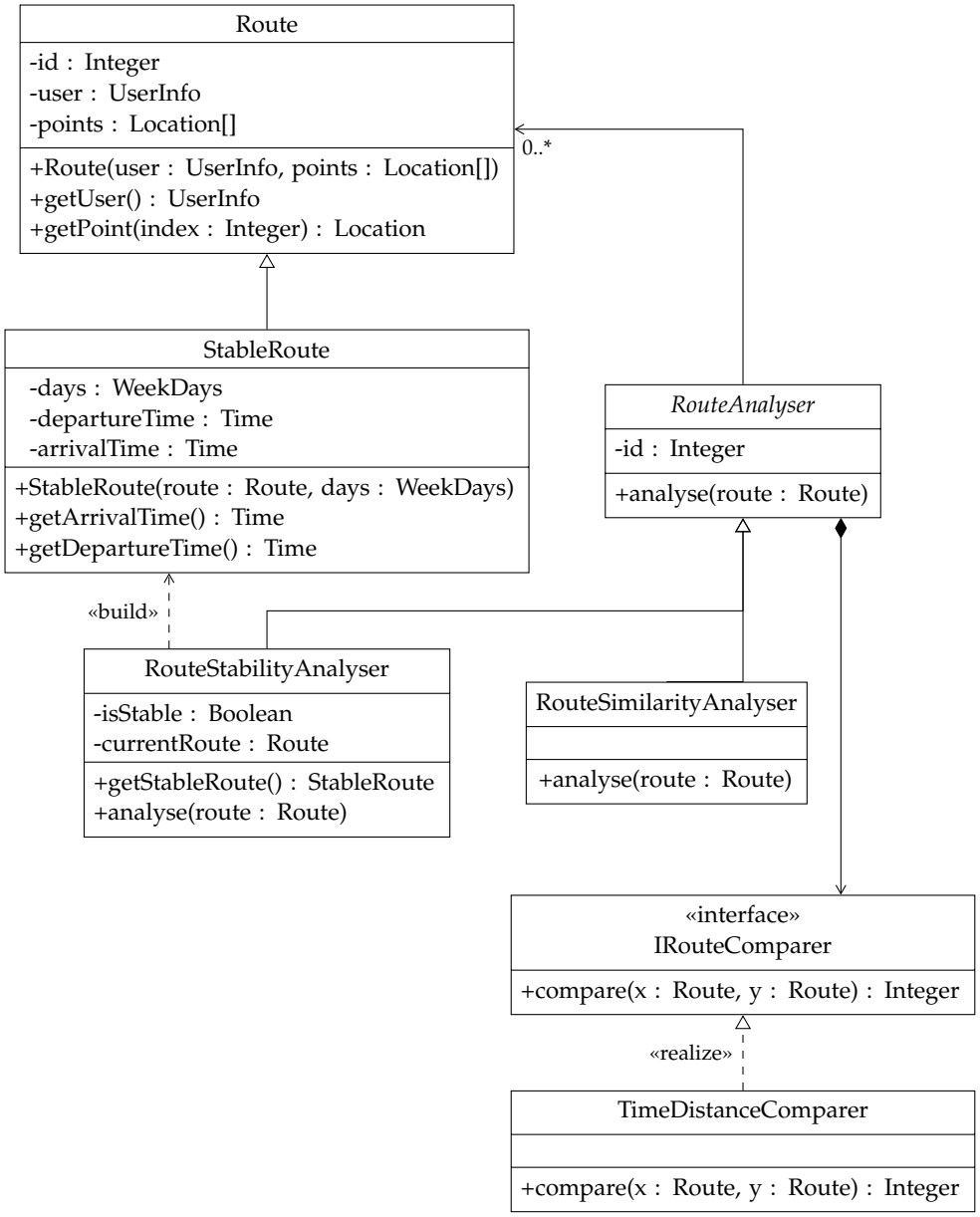


Figure 2.2: The structure of the system inside the server.

2.3.2 User Interface Design

The user interface (UI) provides interaction methods between the user and the app. As the priority of this project is functionality with regards to data collection and route generation and comparison, the UI will not be developed together with users, nor allocated excessive resources. The UI is developed for practical purposes, such

as testing, and to lay the basis for further development, but is still held to a usable standard which will be documented in the following sections.

Design language

We consider the design language as the visual look and appearance of the app. This is the foundation for the user impressions and affects the user experience. The design language is selected in collaboration with another aSTEP project group, SW604F16.

The user interface is designed to achieve the following usability characteristics described by Benyon [11]:

- Learnability
- Utility
- Safety
- Effectiveness

The UI is designed to comply with the Google's Material design guidelines [12], being "*bold, graphic, intentional*". We consider Material Design to be a nice modern design framework, and it is the recommended design language to use in Android applications, by Google[13].

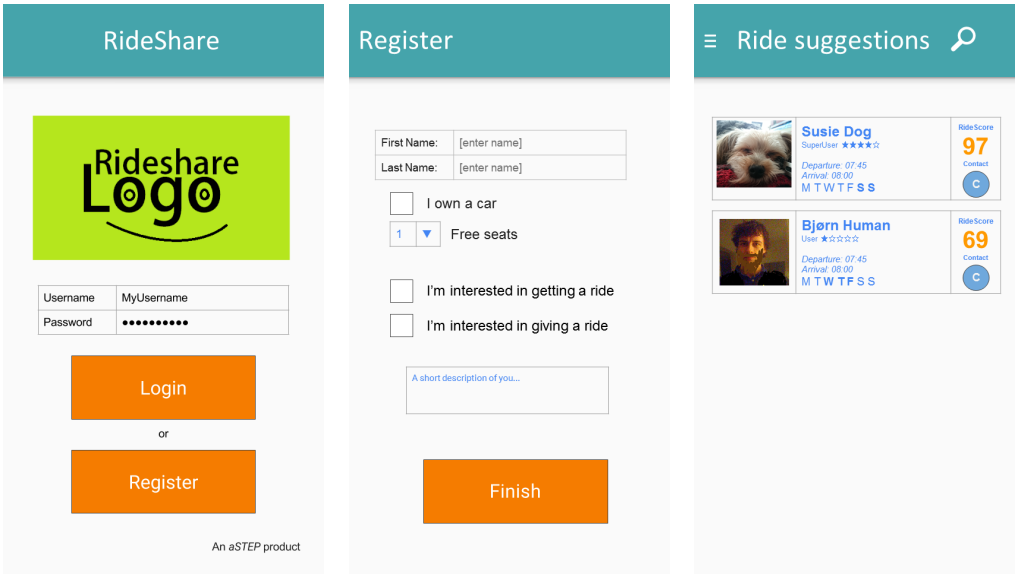
Adhering the material design guidelines makes the app achieve a similar aesthetic and usage method as other apps in the Android eco-system. The design language aims to make the interface clean and simple, regarding colors and input methods.

User interface

This section contains design drafts of the user interface style and functionality. The design drafts in Figure 2.3 reflect the necessary functions and the previously described design language.

As the user needs a user profile in the aSTEP system to use the app, the users must be able to log in if they have a user profile, and create a new user profile if they are not already registered.

Several views are necessary to support the different parts of the app. The app needs views for login, registering, and presentation of matches as can be seen in the figures in Figure 2.3.



(a) Login page **(b)** Register page **(c)** Main page

Figure 2.3: Draft of the views in the app.

The application must be able to show matches in a separate view. In this view, the user will be presented a list of other users which most likely are good matches. The list of user matches should be in a descending order, with the highest rated match at the top. A short description of the users and match should be displayed, such as the user's full name, phone number, the match score, and what day(s) of the week the user drove the route and is expected to drive again.

This concludes the design of the app. As earlier mentioned, the UI is not the main focus of the project and will therefore not be further developed for other than functional purposes.

Chapter 3 Sprint 2

Whereas the main topic of sprint 1 was an overall analysis and collaboration agreement, the topic of sprint 2 is design and development based on the results achieved in sprint 1. The sprint intentions are to develop a route comparison algorithm that can be implemented in the aSTEP system and to develop an app that has basic functionality, to be advanced upon in further sprints.

This chapter contains documentation of the analysis, design, implementation and test phases of the algorithm and app development performed in sprint 2.

3.1 Analysis

This section contains the analysis of app functionality, Android app development, RideShare algorithms and communication with the aSTEP system.

3.1.1 App Functionality

The app user interface is as earlier stated a low priority task during this project. However, the app needs basic functionality, according to the requirements.

To narrow the app focus and to ensure that users matched by the algorithm actually can transport each other, it is decided to only track users when they are traveling in a vehicle. This excludes consideration of routes that are using bicycling or walking as means of transportation.

To be able to differentiate the stored routes, and to assign them to their respective users, there has to be a user management system, so that each user has its own unique id. The user should also be informed with relevant information of ride matches, so that the user can make a decision of sharing a ride or not.

The users of the app should also be able to adjust their settings. The settings are preferences regarding different properties of ride sharing. The user should be able to decide if they wants to get a ride, give a ride or both. There should also also be properties related to the user them self, like a description and a picture, that other users can access to decide if they want to share a ride with the user. These settings are of 'Should have' priority, and not considered further until the app is of acceptable functionality otherwise.

3.1.2 Android Version Selection

When developing Android apps, one of the first choices to consider is which API levels to target [14]. The "*API Level is an integer value that uniquely identifies the framework API revision offered by a version of the Android platform.*", according to Android Developers [14].

An Android application must specify the API level for three definitions: minimum, target and maximum. The target and max versions requires few consideration to determine. The "targetSdkVersion" reflects the version to which the app is developed and tested against, without enabling compatibility behaviors. The "maxSdkVersion" reflects the highest API level that an app is designed to run. Both of these are set to the highest API level available during the start of the development, which is API version 23.

The last API level that needs to be specified is the "minSdkVersion". This entry specifies the lowest API level that the app supports. A consequence of this is that the app will not be compatible with any device with an API level lower than the specified "minSdkVersion". Supporting older API versions requires development efforts, typically implementing functionality though the Android Support Library [15].

For this project we choose to target a higher API level to simplify the development process. At the time of writing, the platform version distribution sum of API levels greater than or equal to 21 (Android Lollipop) is 35.3%. The distribution of the Android version levels are shown in 3.1.

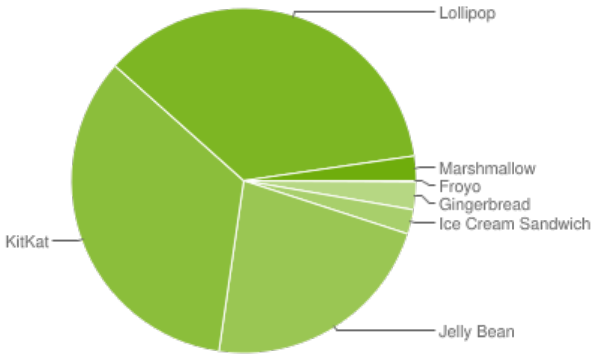


Figure 3.1: Android version distribution, March 2016 [16]

API version 21 is chosen as the "minSdkVerison". The main reason for this choice is to shorten development time as API level 21 added support for the material design style, among other improvements [17]. Refraining from further backwards compatibility allows the focus of the project to be on the functionality of the app.

3.1.3 Periodic Background Tasks

When the app is installed on the device, the app needs to run a task to periodically collect location data. Friesen and Smith [18] writes how this can be done in the Android API by utilizing the AlarmManager and the JobSchedular.

AlarmManager
The AlarmManager was created to handle alarms, and implemented as a general class that can call any task after one or multiple given time periods. The alarm can be set to be reoccurring, thus gather a location with the given interval.

JobScheduler
The JobScheduler was implemented in Android 5.0 (API 21) as an alternative to the AlarmManager. The scheduler behaves in a similar way as the AlarmManager, but tries to batch the jobs together and execute them in bundles. This means that the device can save power by avoiding going to sleep just to wake a short moment later, on the cost of time precision. While the AlarmManager has the option to occur at a exact time, the JobSchedular does not.

The JobScheduler is chosen as the basis for the background tasks, because the timing accuracy of the location data is not sensitive to minor deviations, in addition to the advantages of saving power features.

3.1.4 Location tracking

In order to be able to locate a user and to construct routes, it is necessary to collect location data. Such collection of locations can be done by using already existing services, such as the Google Play Services.

Location data in RideShare is formatted in decimal coordinates in an array that represents a route from A to B. There are several ways to collect location data. One of them is to manually develop a component to do so on the Android device, but the chosen solution is to utilize the already existing Google Play Services.

To use Google Play Services, a client library must be included in the app, and will communicate via inter-process communication to the Google Play Services.

An advantage of the Google Play Services [19], is that it will automatically receive silent updates regularly, to acquire new features and bug fixes to the used services, developed by Google, and this is illustrated in Figure 3.2. The Google Play Services are restricted, and are not supporting devices with Android versions lower than 2.3. This limits the backwards compatibility, but the app is already restricted to Android 5 and newer, and has no influence on the target audience.

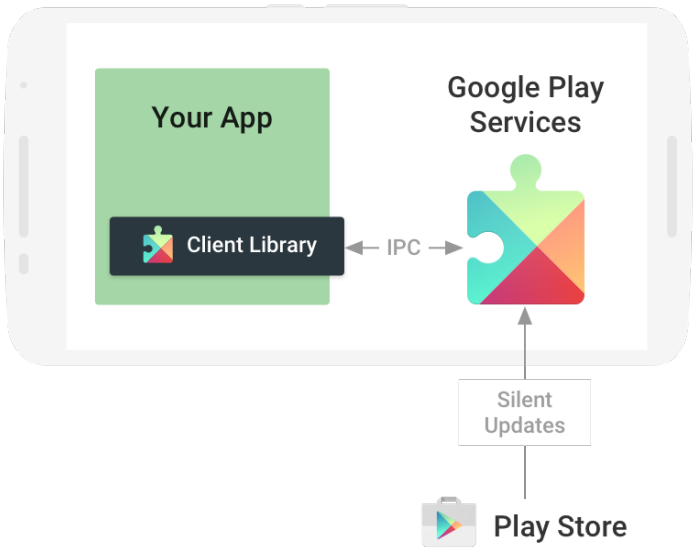


Figure 3.2: Google Play Services communication[20]

The Google Play Services will allow the app to collect location data, but it is not doing so solely by using the GPS in the device. The location service utilizes both network location and GPS to estimate a position as precise as possible [21]. To ensure the collected location data is relevant, locations are only collected when the user is traveling by vehicle.

The Google Play Services provides a service to determine the user's activity, called Activity recognition. Activity recognition is a service that utilizes several sensors on the device to determine what kind of activity the user is currently performing, therein driving, walking, etc. [22]. The service will return a probability level from 0 to 100, where 100 is certain that a user is performing the activity. The activity recognition will also prevent unnecessary data will be stored on the database.

The practical design of the location tracking and activity recognition will be further explored in the following design section. The aSTEP system is examined as an external tool, and its functionalities are explored.

3.1.5 Communication with aSTEP

To be able to utilize the aSTEP system, the communication protocol and its functions must be analyzed. The analysis is based on the current development and plan of the API and other functions.

The intended functions are established in collaboration with other project groups

responsible for the aSTEP system side of the development. The design of the aSTEP system is currently consisting of an API that apps and services can communicate with, and a backend with user management and location based services, which is stored in a database system.

The aSTEP system will store information regarding location data, and basic user information. The data stored in the aSTEP system, relevant to this project solution, is:

- Location data consisting of userID, routeID, GPS coordinates and timestamp
- Username
- Password

The aSTEP user management system does not provide storage of data regarding contact information for aSTEP users. The only information stored is a username and password to keep the aSTEP core as simple as possible. Additional information that is required to make the app work as intended is each of the app groups own responsibility. The aSTEP users are made to ensure the correct permissions are given to the correct user and so that the appropriate data is returned to each user. An API call cannot be done without the user first being authenticated with a valid login.

The communication with aSTEP is done through a REST API over HTTP, decided in agreement between the aSTEP project groups. REST is an abbreviation of Representational State Transfer, and is a communication design often used in for HTTP-communication [23]. Accordingly, the communication is performed by making queries to the aSTEP system. All communication must be done as a request from the device, to which the aSTEP server will respond.

The currently available API functions at this stage of the development of aSTEP from user management and location services can be seen in Table 3.1. The API is providing a POST-request under the name “PostLocationDat”, as listed in 3.1, which is the current method to use when sending location data to the aSTEP system. The call will accept location data as a coordinate consisting of longitude and latitude, a precision value, and a value representing time of day in milliseconds.

As both the aSTEP service and Android system have been analyzed it is possible to establish a set of requirements for the second sprint.

3.1.6 Requirements for the Second Sprint

Sprint two is the first of the two middle sprints that have implementation as the primary focus. In this section, the main issues to be solved in the current iteration

LBS	UM
GetAllEntitiesInArea	Create user
GetAllEntitiesInTimePeriod	Get token
GetAllGroupMemebersLocationAndName	Update password
GetAllFriendsInArea	Edit privacy settings
GetAllFriendsInRadius	Allow user2 to access user1's info
GetAllGroupMembersInArea	
GetAllGroupMembersInRadius	
PostLocationData	

Table 3.1: Currently planned aSTEP API functions.

will be presented, prioritized by the most important first.

Route Matching Algorithm
In this sprint, the algorithm for comparing routes must be designed and developed. It should be developed to an extent so that it is constructed as pseudocode of the algorithm, to be handed over to the outdoor aSTEP group, and be implemented in the aSTEP system.

User Data
The storage of user information such as username, password, and login token, must be designed, and how it should be handled regarding communication between the application and the aSTEP server.

RideShare App
A base for the functionalities of the RideShare app should be implemented. The implementation should include a location gathering service. The collector must include the Google Play Services and the app should be able to store the collected location data. The application should also have functionality to run as a background service, so that location data can be collected at all times.

System Architecture and Communication
Communication between the RideShare application and aSTEP server should also be researched and an implementation of the communication should be initialized in this iteration, ensuring the next iteration will have a base for communication.

The set of requirements enables the design phase to be initiated.

3.2 Design

This section contains documentation of the design of the RideShare solution. The parts to be designed in this sprint are the overall system design, reflecting the

changes in the aSTEP system, and the algorithms to generate stable routes and route matches.

3.2.1 System Design

In the first iteration, the RideShare system was designed to consist of two parts: The RideShare app and the aSTEP system. Because the aSTEP does not provide the anticipated services, such as storing basic user information, this data must be stored elsewhere. This forces a redesign of the system. This section contains descriptions of the redesign of the different parts, and a definition of their respective responsibilities.

For users to able to contact each other when a match is given, the solution will need additional data storage to save the user information, and hence fulfill the requirement. The aSTEP system is supposed to be kept as generalized and modular as possible, and when combined with the lacking user information storage, app specific data could also be stored together with the user information. The app specific data is information such as match scores between routes.

An appropriate solution for storing this extra information is setting up a custom RideShare server, to utilize together with the RideShare app and aSTEP system. The RideShare server could store the extra user data and connect the user to the aSTEP user ID, so that each RideShare solution user has a unique aSTEP user, with contact information stored on the RideShare server. To make sure that the information is stored appropriately, the creation of the user is performed through the RideShare server, that further can create the user in the aSTEP system.

The RideShare server has two main purposes. First, to store additional information about the system users, and second to store the score for the previously assessed scores between given stable routes. The different purposes and responsibilities of the parts depicted in Figure 3.3 will be elaborated in the following sections.

The RideShare app continues to be responsible for the interaction with the user as well as collecting the location data needed to track routes and recommend ride matches. The app must therefore communicate with both the RideShare server and the aSTEP server through their respective APIs.

The API and server associated with the aSTEP system will be responsible for saving and retrieving location and route data for each user in the system. The aSTEP system is also responsible for the basic user profile, consisting of a userID and a password. A user profile is used to identify location data, and to store other users' permissions to access the actual user's location data. The RideShare server

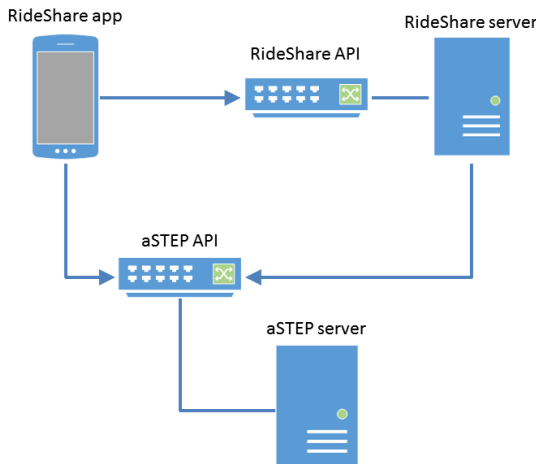


Figure 3.3: System design, including all major parts of the solution.

will be used to store additional information regarding users and routes. The user information that will be stored, is the additional contact information.

Beyond storing scores for route matches and additional user info, the RideShare server will also be responsible for sending request to the aSTEP server, concerning calculations about stable routes and route matches. To be able to gain access to each user's location data from aSTEP, the RideShare server has to store some sort of login information. The aSTEP system provides a login token, so that a password does not need to be stored on the RideShare server.

The system design developed in the second iteration, described in this section, can be seen in Figure 3.3.

3.2.2 Algorithm

The RideShare system uses a custom developed algorithm to determine whether two routes are a good match. This algorithm is loosely based on a simplified version of Ghoseiri et al. [7], with regards to the process of the evaluation of a match. In the paper, the algorithm considers multiple aspects when assessing matches, such as smoking, gender and age preferences. The criteria are practical to some extent, but the focus of the RideShare system is the time and distance criteria, and hence the other criteria are discarded.

This paragraph contains definitions to clarify the terms used in this section. A position is spatial-temporal, whereas a location is a spatial point and a time is a temporal point. A route is a chronologically sorted list of positions.

To illustrate how the algorithm is supposed to work for different routes an example is introduced. The example can be seen in Figure 3.4. The example introduces four

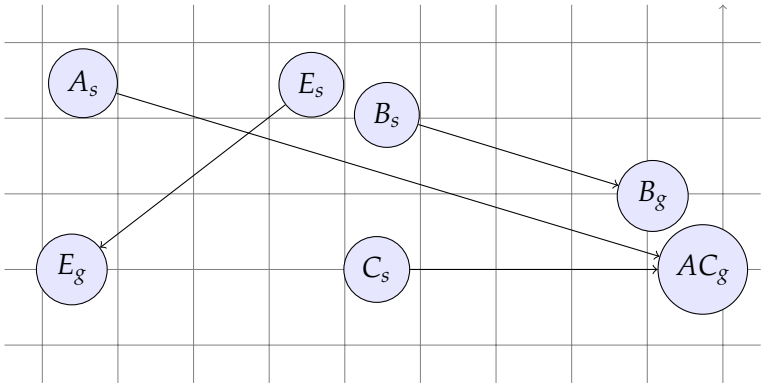


Figure 3.4: Alice, Bob, Carol and Eve's stable routes. Each grid represent a 5×5 km area.

User	Departure	Arrival	Distance
Alice	07:20	07:55	36.5 km
Bob	07:35	07:50	14 km
Carol	08:05	08:30	15 km
Eve	07:30	07:50	18 km

Table 3.2: The four users departure and arrival times as well as their distances.

users, each with one stable route in this example. In addition to the location of their routes, we also have their departure and arrival times that can be seen in Table 3.2.

For this example, the goal will be to find out if any route matches with Alice's route $\{A_s, A_g\}$. The outcome is a score representing if Alice, on the way through route r, should pick up some of the persons Bob, Carol or Eve, for each of their routes.

The expected outcome is Bob will match reasonable in both distance and time difference and hence be given a score indicating a match. Carol is expected to match reasonable in distance as well but will lack a match comparing travel times and therefore not produce a match. Comparing with Eve a somewhat match on time is expected while the distance will be insufficient. The score represents the match of the routes of Alice and each of the other, and should be expressed as a decimal value from 0 to 1 which tells how good the match is. 0 represents a bad match, a major detour or not compatible times, and 1 represents a perfect match, and requires no detour. A comparison will be made for r_1 against r_2. To summarize, the algorithm takes a route r and two positions to compare against; start (s), and goal (g) as arguments. The algorithm then returns a decimal value between 0 and 1.

Time Distance Analyzer Algorithm

There are two aspects to be evaluated in the algorithm: time and distance. These are assumed to be evaluated by two sub algorithms that both takes the route and two points as input and return a value from 0 to 1 describing how closely they match.

If either time or distance is assessed unacceptable they are being assessed a score of 0, thus the whole match is unacceptable and given the a final score of 0. From the scores of the sub algorithms, the final score can be expressed as the average score $\frac{d+t}{2}$, where d is the distance score and t is the time score. To increase flexibility of the algorithm, two constants are introduced d_γ and t_γ. These constants enables weighting of the time and distance score independent of each other by expanding the expression to $\frac{d \times d_\gamma + t \times t_\gamma}{d_\gamma + t_\gamma}$.

For our example we simply set both d_γ and t_γ to 1.

Algorithm 1 Time Distance Analyzer Pseudocode

Require:
let d_γ be the modifier for distance in $\mathbb{R}_{>0}$
let t_γ be the modifier for time in $\mathbb{R}_{>0}$

```
1: function TimeDistanceAnalyzer(r, s, g)
2:     d ←DistanceAnalyzer(r, s, g)
3:     t ←TimeAnalyzer(r, s, g)
4:     if d > 0 ∧ t > 0 then
5:         return (d × d_γ + t × t_γ) / (d_γ + t_γ)
6:     else
7:         return 0
8:     end if
9: end function
```

The time distance analyzer can be seen in Algorithm 1. Line 2 and 3 are calls to the algorithms that solve the subproblems of scoring the time and location, and line 4 to 8 is the logic described in the previous paragraph.

The following sections will cover the sub algorithms. First, the distance analyzer algorithm will be described, followed by the time analyzer algorithm.

Distance analyzer

The distance analyzer algorithm takes a route and two end points and assesses the score for the detour when only considering the distance. To accomplish this, it is required to define how long the longest acceptable detour should be. The algorithm also needs a way of calculating the distance between two discrete locations, which can be performed by the function for euclidean distances. This leads to the following algorithm, where the symbol $\mathbb{L}$ is used to denote the set of all possible locations:

$$\begin{aligned} dist : \mathbb{L} \times \mathbb{L} &\to \mathbb{R}_{\geq 0} \\ a, b &\mapsto \sqrt{(a_{latitude} - b_{latitude})^2 + (a_{longitude} - b_{longitude})^2} \end{aligned}$$

Algorithm 2 Distance Analyzer pseudocode

Require:
let β be the largest acceptable detour length in $\mathbb{R}_{>0}$

function DistanceAnalyzer(r, s, g)
  let r_s be the closest point to s in r
  let r_g be the closest point to g in r
  $d_s \leftarrow dist(r_s, s)$ ▷ Pickup detour distance
  $d_g \leftarrow dist(r_g, g)$ ▷ Set off detour distance
  return $1 - \frac{d_s + d_g}{\beta}$
end function

A disadvantage of the algorithm presented here is that the distance is calculated as a euclidean distance rather than a road distance. This would be a obvious place for future improvements to the algorithm.

The distance analyzer algorithm, as seen in Algorithm 2, finds the points on the route that are closest to the start and end points. From those points, the algorithm approximates the detour required to take by the driver to pick up the passenger. The approximation assumes that the driver's new route distance will be the original length plus the distances to the passengers start point and goal point. This assumption does not represent the actual detour and thus this is an area for future optimization.

The final score for the distance analyzer is calculated by taking the total detour length and dividing it by the length of the largest acceptable detour. This results in a value between 0 and 1 when the detour is less than the largest detour, and

greater than one when the detour is longer. The value is inverted by subtracting it from 1 to make better scores greater values, as required by Algorithm 1. If the value is greater than the acceptable detour length, the inversion in the return statement will generate a negative value, hence causing Algorithm 1 to return a result representing no match.

In the example introduced in the beginning of this chapter, the longest detour is set to be 25% of Alice's total commute and Alice's route is modeled as the line $y = \frac{4}{7} - \frac{2x}{7}$. It is assumed that Alice's route have tracking points continuously in the line, and therefore a line to point distance calculation is used in this example, this means r_s and r_g will be calculated simultaneously as the distance is calculated. This gives the the results showed in Table 3.3 when applying the distance Algorithm 2.

User	d_s	d_g	**return value**
Bob	3.57 km	1.85 km	0.550
Carol	4.12 km	0 km	0.609
Eve	9.62 km	4.12 km	-0.505

Table 3.3: The detour distance for Alice compared to the other users as well as the score of the Distance Analyzer for each route.

Table 3.3 presents the results for the distance analyzer. The algorithm assesses the best results for Carol, followed by Bob while Carol yields a negative result since Alice's detour becomes longer than the acceptable 25%.

Time analyzer

The time analyzer algorithm works similarly to the distance analyzer algorithm. It is also designed to take two points and a route as input and to return a score representing the time differences.

When determining time difference two things should be taken into consideration; the detour duration and the existing differences in time. The detour duration can be approximated by multiplying the detour distance with the time it takes to travel a distance unit.

Each position in each driver's route has a timestamp for when the location was visited. This can be used to determine the relative time distance between two points by using the following function:

$$\begin{aligned} time : \mathbb{L} \times \mathbb{L} &\to \mathbb{R}_{\geq 0} \\ a, b &\mapsto |a_{time} - b_{time}| \end{aligned}$$

The calculated detour time and relative time difference enables an approximation of the inconvenience the ridesharing is for the participants in the potential match. From the driver's perspective, the route starts at the normal time minus the detour duration, to still arrive on the usual time at the destination. This makes the detour the only inconvenience in regards to time. If the passenger has to arrive earlier than the driver, the driver would also need to take that into consideration, thus making the relative time distance the inconvenience in addition to the detour.

From the passenger's perspective, the opposite is true when it comes to the relative time. It is only an inconvenience when the driver must set them off before they need to arrive. It is also worth noting that if the relative time is the same as the detour time, the only inconvenience for the driver is the detour.

The formula for the total inconvenience when taking both participants into account is then defined as the following:

$$|d - t| + d$$

where d is the detour time and t is the relative time.

Algorithm 3 Time Analyzer pseudocode

Require:
let δ be the acceptable time difference in $\mathbb{R}_{>0}$
let γ be the translation from distance to time in $\mathbb{R}_{>0}$

1: **function** TIMEANALYZER(r, s, g)
2: let r_s be the closest point to s in r
3: let r_g be the closest point to g in r
4: $d \leftarrow (dist(r_s, s) + dist(r_g, g)) \times \gamma$ ▷ The total detour time
5: $t \leftarrow max(time(r_s, s), time(r_g, g))$ ▷ The largest time difference
6: **return** $1 - \frac{|d-t|+d}{\delta}$
7: **end function**

Algorithm 3 shows the pseudocode for the time analyzer algorithm. Line 2 to 4 are reminiscent of line 2 to 5 in Algorithm 2 with the difference that they sum the detours and multiplies the result with the distance to time conversion. Line 5 assigns the greatest time difference between the closest points to t. The reason for choosing the greatest value is to use a pessimistic approach, so that the real delay will not be worse than the calculated.

For the Alice example, defining the speed as constant through the whole route is preferable to keep the example more simple. Alice travels 36.5 km in 35 minutes,

this gives a pace of $\frac{35min}{36.5km} = 0.959\frac{min}{km}$. For this example the acceptable time difference is set to 15 minutes $\delta = 15$. The time analyzer algorithm is applied to the example with the aforementioned data, the result can be seen in Table 3.4.

User	d	t	**return value**
Bob	5m:12s	3m	0.63
Carol	3m:57s	35m	-0.75
Eve	13m:11s	18m	0.10

Table 3.4: The detour time and time difference for Alice compared to the other users as well as the score of the TimeAnalyzer for each route.

In the results from the example in Table 3.4, it can be seen that Bob receives a relatively high score, while Carol's score is below zero, and Eve is just within the acceptable range.

With the scores from the two subalgorithms, Algorithm 1 is able to assess the matches' final scores. The final results of the algorithm example can be seen in Table 3.5.

User	d	t	**return value**
Bob	0.550	0.63	0.59
Carol	0.609	-0.75	0
Eve	-0.505	0.10	0

Table 3.5: The result for Alice's match compatibility to the other users calculated by the Time Distance Analyzer.

The results in Table 3.5 shows that with the set detour limits in distance and time the only compatible match for Alice is Bob with a score of 0.59. Although Carol scores acceptable in the distance analyzer, the time difference is unacceptable thus assessed a score of 0. Eve scores below 0 in the distance parameter and is therefore also given a final score of 0.

The two modifiers d_γ and t_γ were set to 1 in this example, but these modifiers, along with the acceptable detours in time and distance, can be adjusted to adapt scores if the given outputs do not reflect the expected outcomes or the users' actual willingness to share rides in a given situation.

Complexity
Considering the complexity of the three algorithms, the Time Distance Analyzer complexity is only dependent on the calls on lines 2 and 3, while the rest of the algorithm is input independent. Line 2-3 are the calls to the two sub algorithms.

In the Distance Analyzer Algorithm 2 line 2 and 3 is where the whole route should be iterated through, the complexity of this sub algorithm is $O(|r|)$ where $|r|$ is the number of positions in route r. Line 4 to 6 perform arithmetic operations that run in constant time, and can be ignored when describing the asymptotic notation.

For Time Analyzer the complexity is the same as in Distance Analyzer because of the similarity of the first few lines and the last lines are simple arithmetic operations. The exceptions are *dist*, *time*, and *max*. *time* and *dist*, which both are described earlier, are considered to be $O(1)$ operations. *max* takes two values, compares them and returns the largest of the two. This is also a constant operation, and hence we can determine that the Time Analyzer algorithm is $O(|r|)$.

Thus we have that the main algorithm is $O(|r|)$, the number of locations in a route, for each stable route in the aSTEP system.

3.2.3 Location

As documented in the analysis in Section 3.1.4 the choice of activity recognition and location gathering was the Google Play Services.

In the following test some values will be decided, in concern to the design of the location tracking. These values are yet uncertain and might be adjusted later as more knowledge of the practical implication of these are gathered.

Activity recognition should be polled continuously to check whether a user is registered as in vehicle. When a user is in a vehicle with a probability level above 80%, a location request must be be performed.

The application must be logged into and running in the background while the device is turned on to be able to differ between the users activity at all times and collect data. The interval for collecting location data from the location services will be set to every two minutes, while in a vehicle, to reduce energy consumption. Every location collected during the driving activity must be appended to a list that represents an entire route when the driving activity ends.

When the activity change from vehicle to any other activity, the application will send the list of locations to the RideShare server, where it will take over the processing.

3.3 Implementation

In this section we will document the initiating implementation of the RideShare system, that has been designed until now. In this sprint the implementation of the location gathering as a background service will be documented.

3.3.1 Background Service

The `JobScheduler` is chosen as the scheduler for managing the background service in Section 3.1.3. The `JobScheduler` works by scheduling Intents, that are operations to be performed. An `Intent` that is not yet executed, is called a `PendingIntent`. When a `PendingIntent` is finally executed, variables in the systems may have been changed, hence the context might not be the correct, therefore a `PendingIntent` also requires a reference to the context when it was created. This is performed to ensure that permissions given to the app are still available when the `job` is executed. A `job` is a `PendingIntent` as described in Section 3.1.3 and a set of requirements for when the intent should be executed.

When a job is handed to the `JobScheduler`, information about the nature of the scheduling is also provided. This includes requirements that needs to be fulfilled before the job can run, how often the job should be executed, and how precise the timing of the job should be.

```
1 JobScheduler jobScheduler = (JobScheduler) getSystemService(JOB_SCHEDULER_SERVICE);
2 JobInfo.Builder builder = new JobInfo.Builder(1, new ComponentName(getPackageName(), RideShareService.class.getName()));
3 builder.setPeriodic(120000);
4 int response  = jobScheduler.schedule(builder.build());
```

Listing 3.1: The implementation of jobScheduler.

Listing 3.1 presents the implementation of `jobScheduler` in the `MainActivity` of the app.

In the app, the jobScheduler is implemented as an extension to the `JobService` class which provides the required interface for the `JobScheduler` called RideShare-Service. The class reroutes the call by the `JobScheduler` to a `Handler` class. This `Handler` class, as described by the official documentation [24], allows the service to send a runnable object to the threads message queue.

The implementation makes the location gathering being instantiated as a background service every two minutes. As covered in Section 3.1.4 this interval is

temporary, and will serve to test the implementation, until further experience is gathered regarding optimizing the task frequency.

3.3.2 Location Gathering

This section covers the implementation of the location gathering part of the background service. The Google Play Services is utilized to get the location of a device in a background service.

First, to retrieve location information, the `manifest.xml` file must be edited as to acquire permissions to the course and fine location. When the manifest is set up correctly, the actual location retrieving can be performed. This is done by adding the following lines to the manifest:

```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

The location is retrieved by utilizing the `GoogleApiClient`, then converted to an aSTEP location format. This prepares the location data to be sent to the aSTEP system.

Google API Client

The `GoogleApiClient` is built and connected to when the `JobService` is created, hence performed in the `onCreate()` method, as the `JobService` thread only exists for the execution time of the contents. The `GoogleApiClient` is disconnected in the method `onDestroy()` when the thread is terminated.

The `onConnected(Bundle bundle)` method is called when the Google API Client is ready. When the client is ready, the `getCurrentGLocation()`, which can be seen in 3.2, is called, and the returned Android Location is stored in `currentGLocation`. The current Google location can then be stored in an aSTEP object, that later can be transmitted to the aSTEP system.

```
@Override
public void onConnected(Bundle bundle) {
  currentGLocation = getCurrentGLocation();

  // convert location to astep if available
  if (currentGLocation != null) {
    ASTEPLocation currentAstepLocation = convertToAstepLocation(currentGLocation)↩
        ;
  }
}
```

Listing 3.2: onConnected()

Get Location

The `getCurrentGLocation()` returns the last known location, according to the `googleApiClient`, in the Android location format. Before the location is gathered, because the target API is 23, the method needs to confirm the permissions required to acquire location data. This is performed on line 6 and 7 seen on 3.3. The device location is gathered through the `FusedLocationApi.getLastLocation()` method, using the `googleApiClient` as argument.

```
1  private Location getCurrentGLocation(){
2    Location tempGLocation = null;
3
4    // Support Android M type permission handling
5    try {
6      if (ActivityCompat.checkSelfPermission(context, Manifest.permission.↩
           ACCESS_FINE_LOCATION) != PackageManager.PERMISSION_GRANTED &&
7      ActivityCompat.checkSelfPermission(context, Manifest.permission.↩
           ACCESS_COARSE_LOCATION) != PackageManager.PERMISSION_GRANTED) {
8        Log.d("BGS-LOC", "pass permission test");
9        tempGLocation = LocationServices.FusedLocationApi.getLastLocation(↩
             googleApiClient);
10     }
11
12     // Get the current googleApiClient/LocationServices location
13     tempGLocation = LocationServices.FusedLocationApi.getLastLocation(↩
           googleApiClient);
14
15   }catch (Exception ex){
16   }
17
18 return tempGLocation;
19 }
```

Listing 3.3: getCurrentLocation()

Convert to aSTEP location

The aSTEP location format contains the essential location information: latitude, longitude, accuracy, and a timestamp. The 3.4 takes an Android location format data, creates an aSTEP location instance based on the argument location, and returns the aSTEP location.

```
1  private ASTEPLocation convertToAstepLocation(Location gLocation) {
2    Log.d("BGS-LOC", "convertToAstepLocation");
3    ASTEPLocation aLocation = new ASTEPLocation(gLocation.getLatitude(),
4    gLocation.getLongitude(),
5    gLocation.getAccuracy(),
6    gLocation.getTime());
7    return aLocation;
8  }
```

Listing 3.4: ASTEPLocation()

3.3.3 Route Matching Algorithm

The implementation of the route-matching algorithm is delegated to the previously mentioned outdoor location based service group, as they are responsible for implementation in the aSTEP system. They were given the pseudocode algorithm developed in sprint 2 and cooperated with during the implementation process. The implementation will not be documented here, as the actual implementation is not performed by us. However, the implementation should be tested, and we generated test data with accompanying result expectations. The test data was also handed over, this will be described in the following test section.

In cooperation with the outdoor group, it was decided to send every variable for the algorithm as parameters in the API call. This was done so that we would have control over the algorithm to ensure it performed according to our expectations.

3.4 Test

The purpose of the test section is to document the current functionality of the implemented solution. An informal test was performed on the app to assess the behavior and precision of the location gathering as a background service.

3.4.1 Background Service

To assess the functionality of the background service, it was necessary to receive visual information, confirming that the service was actually running. The background service was tested by making a IDE print a message to an debug log each time the `jobScheduler` executed the `PendingIntent`.

In accordance with the implementation, the output appeared every 10 seconds, independent of the app being open, in the background or closed completely on the test device. This proved that the background service was functioning as intended.

3.4.2 Location Gathering

Location gathering was also performed in the test of the background service. The location test was performed by trying to access the last known location in each scheduled `PendingIntent`, and to print the result in the debug log in the IDE. The background service was executed regularly, but the location was only available in an insignificant number of the attempts. The missing locations seemed to be caused by the `googleApiClient` not being connected, as the location gathering is

performed in the `onConnected` method. The missing locations seems to derive from the thread not existing long enough for the `googleApiClient` to connect to the Google Play Services.

Our implementation of the `jobScheduler` made it impossible for the `GoogleApiClient` to connect, hence not being able to assess the device location. Additionally, the interval based location gathering, would be using battery consequently, regardless of the activity of the user. It is preferable to activate GPS location feature as little as possible, because GPS location collection is battery consuming [25].

Chapter 4 Sprint 3

In this sprint the focus will be upon finishing requirements from sprint 2 with the primary focus on implementing a better background service to track the users while in vehicles. The current state of the aSTEP API is examined to ensure that the RideShare solution still uses the API calls correctly. Lastly a test is performed on the background service to ensure correct collection of location data.

4.1 Analysis

This section contains an analysis of the previous sprint with the intention of improving the solution, and considerations of further development of RideShare. Additionally, because the aSTEP system is continuously developed and the RideShare solution is dependent on its status, aSTEP will be examined for changes and additions.

First, the location gathering will be investigated, followed by an examination of the current configuration of the aSTEP system.

4.1.1 Location Gathering as a Background Service

A new background service is examined as the background service implemented and tested in previous sprint was not performing correctly. Furthermore, the constant interval used in the previous background service between gathering GPS location is not ideal, as a user could be stationary for periods of time, thus the location gathering is performed unnecessarily when the user is inactive.

Location Gathering
Since Section 3.3.2 we have discovered a new method for gathering locations. Whereas the previous method relied on requesting each location, this can now be done by requesting location updates from the Google Play Services [26]. By this method, an app can receive regular location updates by subscribing to a location provider. The app could then receive a notification when a new location is detected.

Activity Detection
The RideShare app is supposed to gather locations when the user is driving, and hence activity recognition is explored to be able to detect the user's activity and potentially reduce energy consumption.

4.1.2 Changes to the aSTEP API

Because the aSTEP is in development, there has been changes in the API since the last sprint analysis. The aSTEP API has been updated with new functionality both regarding user management and location based services (LBS). The LBS are split into indoor and outdoor locations calls, where only outdoor services are relevant for the RideShare system.

An overview of the relevant API calls, as of sprint 3, for the RideShare system can be seen in Table 4.1, while the complete set of sprint 3 outdoor location services and user management services can be seen in appendix B.

Path	Method	Description
/locations/outdoor/{username}/routes	POST	Send a route to the aSTEP server
/locations/outdoor/{username}/routes/match	GET	Match all new routes
/users	POST	Create a new user
/users/{username}/outUsers	POST	Request a new out user
/users/{username}/outUsers/{specifiedUsername}	PUT	Accept out user
/users/{username}/token	GET	Get the current valid token for a user

Table 4.1: Relevant aSTEP API functions.

For the location-based services, the aSTEP system now offers different get requests regarding a user's location history, nearby users, and users based on groups or friends. Based on a collaboration between us and OD, the API now supports a call to post routes directly to the aSTEP system. As an interface to the algorithm results, the "routes/match" returns the matches made by the algorithm.

The services offered by the user management is also expanded since the previous sprint. The implementation of the concept of groups entails multiple new API calls. The group concept is implemented as an access control that restricts users access to other users data. The RideShare system design will need to reflect these changes, and will be further analyzed in the following sections. The particulars of the different API calls will be elaborated in the following sections as they are used.

4.1.3 Requirements for the third sprint

The third sprint is will be used to satisfy the requirements not met in the previous sprint. The app is currently a working template and can instantiate a background service, but is not completed. The background service redesign and implementation are the main focus of this sprint, together with some testing of the implementation of the route-matching algorithm.

Location Gathering as a Background Service
The original requirement of location gathering was not fulfilled by the implementation in sprint 2, thus, it must be reimplemented. The background service should be sensitive to the user's current activity and only record locations when the user is driving.

Conclude the Route Matching Algorithm Through Test and Collaboration
The route-matching algorithm was designed in sprint 2 and handed over to the outdoor group responsible for the implementation in aSTEP. The algorithm is supposed to be implemented within this sprint, and the implementation should be tested for result correctness. The testing must be done with some test data, along with result expectations.

4.2 Design

This design section contains documentation of the redesign of the background location gathering service and the design of managing access to user data.

4.2.1 User Management

User management must be performed for the RideShare system to utilize the route-matching algorithm because the algorithm requires access to the routes and locations based on the different users.

There are two ways of resolving user data sharing using the aSTEP API. The first solution is to create a group for all the RideShare users. The second is to create one administrating RideShare user that has an access edge to every RideShare user in aSTEP system. The alternatives are elaborated in the following paragraphs.

Groups
Groups in the aSTEP system exist to allow users to gain access to each others data. The group functionality could be utilized by making a RideShare group in which all RideShare users are included. An issue with this solution is that all users have granted all other users access to their own location data. The solution allows a malicious programmer to create a program that either copies or edits the locations of all users of the RideShare app. The intruder would only need a registered RideShare user and would thereby be included in the group.

Edges
Edges work in a similar way to groups, but only allowing access from one user to another. This can be accomplished by establishing communication using the same

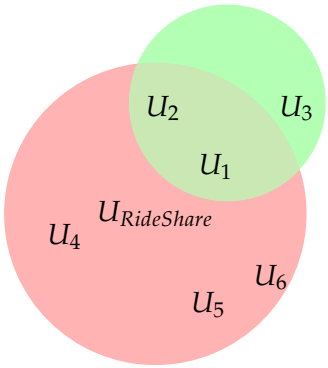


Figure 4.1: How RideShare would use groups. In the red group all users in that group can see all data from other users in the same group.

method as with the groups and creating a RideShare user and instantiate an edge from the RideShare user to every other user in the RideShare system, illustrated in Figure 4.2. This way, only when the RideShare system uses its own user the remaining user data can be accessed.

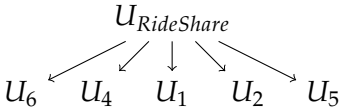


Figure 4.2: How RideShare would work with edges.

The edges method allows edges to provide the same benefits as groups for this use case while disabling the security flaw mentioned above. Therefore, the edges method is selected as the user management design pattern.

Creating users
In sprint 2 it was also decided to create users through the RideShare server, this has been changed to reduce the design complexity and simplify the implementation.

When a user registers in the RideShare app, the app will issue an API call directly to the aSTEP API, instead of performing the operation through the RideShare server. The username and additional information, such as phone number and full name, will still be transferred to the RideShare server to enable contact information sharing with other RideShare users.

4.2.2 Location Gathering

The location gathering should be performed unnoticeable for users and in the background without any user interactions. This requires certain design features

that will be presented in the following sections.

Overall Design

The informal tests performed in sprint 2, Section 3.4, showed that polling locations with fixed intervals controlled by a timer proved difficult to get working reliably, as the Google API client rarely seemed to connect before the task finished executing. Thus we explore the possibility of using a broadcast receiver which does not provide fixed time intervals, but receives updates within a defined time frame.

To ensure location gathering is not performed more than necessary, the activity recognition service should be running continuously and decide the location gathering status. The service should start requesting locations when the user is detected as driving. The location updates request is still done through the Google Play Service location API. As long as the activity registers the user as driving, the location should be received with reasonable intervals. The gathered locations should be collected into a group of locations representing the current route. When the the activity recognition has not been registered as driving for five minutes the route is considered ended, and the route building should be finalized and send to the server.

The currently used intervals are estimated by us. The method of ensuring useful data is done by a trial and error method, as there is little to base the intervals on. If the tests find that the data are not precise enough, the interval values will be changed accordingly.

Polling frequency

The location update and activity recognition frequency is balanced between energy consumption and accuracy.

We consider the initial task of the location gathering service: The activity recognition. The activity recognition is used to decide the status of location requests, as it has lower energy consumption than the GPS unit [25] since the GPS system prevents a phone form going into its normal sleep cycles. Many newer smartphones [27] have dedicated processes to track the sensors responsible for activity recognition and therefore use little power. The interval of activity recognition will be set to every five seconds, which can be optimized to use less power or be more accurate if needed.

Currently the location gathering will be a simple solution which will request with static intervals, whereas dynamic intervals based on factors such as speed could

possibly be evaluated for battery optimization later on. The location updates interval will be set to most frequently at one minute, but targeted at every two minutes. Although the interval values probably are within the time area, the values are susceptible to changes depending on test results.

4.3 Implementation

The implementation section contains descriptions of selected parts of the implementation performed in sprint 3. The following paragraphs will describe the contents of the implementation of the background location gathering. The new implementation is founded on the poor results of the location gathering method implemented in sprint 2, Section 3.3.2, which was not working as intended. Code examples in this section will not include potential logging and debugging lines as for better readability and they do not influence the functionality of the code.

Implementation plan
According to the design, Section 4.2.2, the locations should only be requested when the user is driving. Therefore, the implementation starts by instantiating an activity recognizer. The activity recognizer can thereafter check the received activity, and begin location gathering if the activity is recognized as driving with a confidence above 80%. When the confidence is lower than the threshold for a period of time, the driving activity can be considered finished, and the route must be stored and later transmitted to the aSTEP system.

Activity recognition instantiation
The location gathering should first detect that the device is in a driving activity and then begin recording locations, according to the design described in Section 4.1.1. To achieve the functionality, the app must connect to a Google API client and subscribe to activity recognition. The code for subscribing to the activity recognition is shown in Listing 4.1. The activity recognition can thereafter check that the detected activity is `inVehicle`, and accordingly activate the location gathering.

```
1 if(activityDetectionBroadcastReceiver == null) {
2   activityDetectionBroadcastReceiver =
3     new ActivityDetectionBroadcastReceiver(googleApiClient, this);
4
5   LocalBroadcastManager.getInstance(this).registerReceiver(
6     activityDetectionBroadcastReceiver,
7     new IntentFilter("fapptory_inc.rideshare.BROADCAST_ACTION"));
8 }
```

Listing 4.1: Initialization of activity recognition.

The `activityDetectionBroadcastReceiver` is an instantiation of the `ActivityDetectionBroadcastReceiver` class, utilizing the `googleApiClient` and `this` which in this case references the current `MainActivity` instance to receive detected activities.

The broadcast receiver is then applied by registering to the RideShare app's `BROADCAST_ACTION`, so that the `activityDetectionBroadcastReceiver` only handles broadcasts sent by RideShare and not by other apps.

Activity Detection Broadcast Receiver
The main operations in regards to activity detection and location gathering are done by the activity detection broadcast receiver.

When the `activityDetectionBroadcastReceiver` receives a RideShare broadcast, the detected activities in the broadcast is stored in the `ArrayList<DetectedActivity> detectedActivities`. The `detectedActivities` is iterated over, shown in Listing 4.2 to find the driving activity `DetectedActivity.IN_VEHICLE`. The `startLocationUpdates()`, which is described later, is called if the confidence is above the threshold and if locations are not currently being recorded. Line 8 controls the location cache in case a drive is stopped temporarily and starts again, appending the cached route to the actual route.

```
1  for (DetectedActivity da: detectedActivities){
2    if(da.getType() == DetectedActivity.IN_VEHICLE){
3      if(da.getConfidence() >= ACTIVITY_CONFIDENCE_VALUE){
4        if(!isCurrentlyCollectingLocations) {
5          startLocationUpdates();
6          isCurrentlyCollectingLocations = true;
7        }
8        if(potentialStopDrivingActivity && potentialAstepRoute.size() > 0){
9          astepRoute.addAll(astepRoute.size(), potentialAstepRoute);
10         potentialAstepRoute.clear();
11       }
12       lastDetectedVehicleActivity = System.currentTimeMillis();
13       potentialStopDrivingActivity = false;
14       drivingActivityDetected = true;
15     }
16   }
17 }
```

Listing 4.2: Iteration over received list of activity recognition.

Location Updates Request
The `startLocationUpdates()`, shown in Listing 4.3, is called to gather locations regularly. The location updates request is based on the Google API client instantiated in `MainActivity` and a `locationRequest`. The `locationRequest` is defined with properties regarding location update interval, fastest interval, and location accuracy priority, respectively set to 60 seconds, 60 seconds and `PRIORITY_HIGH_ACCURACY`.

```
1 public void startLocationUpdates(){
2   if(/* Omitted: Check permissions */) {
3     LocationServices.FusedLocationApi
4       .requestLocationUpdates(googleApiClient, locationRequest, this);
5   }
6 }
```

Listing 4.3: Start location updates functions.

4.4 Tests

In this section the different tests that were performed are described and the results are examined and evaluated upon. The goal is to ensure that the functionality, performance, and reliability of the system is acceptable according to our requirements.

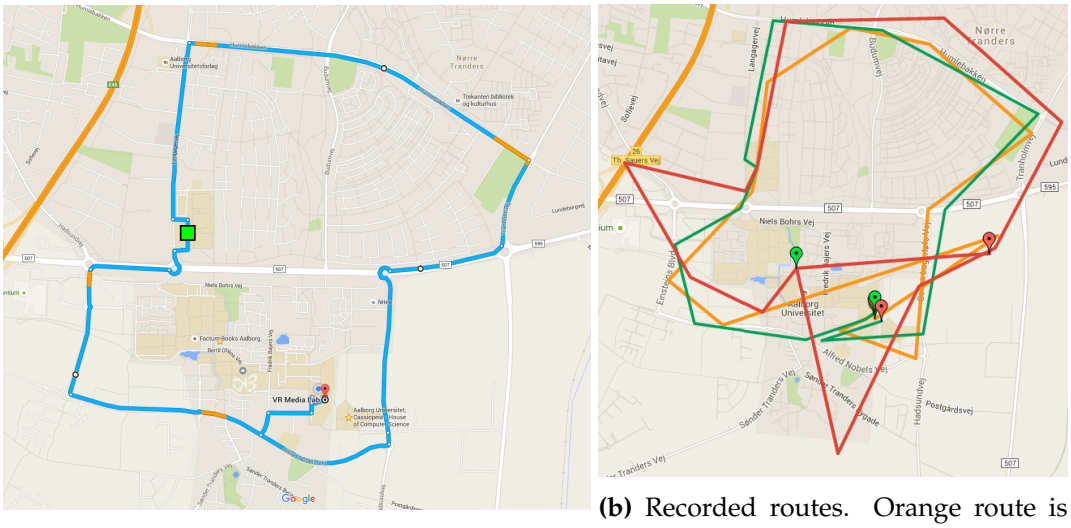
4.4.1 Location gathering as background service

The RideShare app was built and installed on three units to test on a variation of devices. The devices were a Sony Xperia Z2, a Samsung Galaxy S6, and a Huawei Nexus 6P, all with Android Marshmallow 6.0.1 as the operating system. The app was opened and the necessary permissions were granted, so that the app could function properly. The app was configured with the values seen in Table 4.2.

Variable description	Value	Unit
Fastest location update interval	60000	milliseconds
Target location update interval	60000	milliseconds
Activity confidence value	80	percent
Activity stop threshold	180000	milliseconds
Minimum number of route locations	4	locations
Request activity updates	5000	milliseconds

Table 4.2: Parameter configuration in test of the background service route tracking.

The route driven to test the location gathering can be seen in Figure 4.3a. The route was driven with the intention of gathering the whole route as a single route on each of the devices. The drive had one stop at the point marked by a green square in Figure 4.3a. The stop had a duration of two minutes, thus not passing the threshold value of three minutes. The drive continued and ended at the start position, where the persons of the respective devices walked from the vehicle to the group room.



(a) Test drive route, driven clockwise from the red pin.

(b) Recorded routes. Orange route is Nexus 6P, red route is Xperia Z2 and green route is Galaxy S6.

Figure 4.3: Actual route and recorded route data.

All three devices were able to gather the driven route. The routes were recorded with approximately the same start and end time, and the recorded routes can be seen in Figure 4.3b. As the figure shows, the gathered route data relatively accurately represent the actual route.

The Xperia device had the largest deviation from the actual route, and this could be caused by multiple factors. Because the device was placed below the debugging laptop, interference with the GPS signals could occur. The device could also have a poorly manufactured GPS unit. This is not further investigated, as the two other devices gathered fairly accurate locations during the test.

The test reveals that the app is able to gather locations whilst the app is not actively being utilized by a user. The location accuracy seems to be device specific. Location gathering as a background service is considered accomplished and the respective requirement is evaluated as fulfilled.

4.4.2 Route Match Algorithm

The route matching algorithm was implemented by one of the aSTEP outdoor groups, and to check that the implementation reflects the intention, a test was performed.

The implementation was tested with a sequence of routes inserted into a previous empty route database. By adding routes sequentially, we could anticipate the

matching outcome for each addition.

Test Data
A number of cases should be tested, and we generated route data for each. Before the route data was generated, users U1, U2, U3, and U4 were created in the aSTEP system.

First, the stable route generation must be confirmed functioning. A route was created for U1 and two copies were made and were slightly deviating in location and time, and skewed respectively one and two weeks ahead of the original route. These three routes should be assessed a high matching score, close to 1, and be considered a stable route for U1.

Second, the algorithm should not be matching routes that were not relevant in location or time. U2 gets two routes added not matching in location, as seen in Figure 4.4. The blue U2 route is during the daytime and the black during nighttime. These routes should thus be assessed a low match score, assumed 0.

Because the algorithm should match routes of different users, we thirdly created two routes for user U3. The routes were differencing slightly in locations and time but were made with the intention to match with each other, with one week difference in time. The routes were also made to roughly overlap with the routes of U1, so that U1's stable route would roughly be a sub-route of the U3's route.

Lastly, the U4 route was made to discern location and time matching of the algorithm. The route is stable and approximately at the same time as the U1 and U3 routes, but is directed in the opposite way, northwest instead of southeast. The U4 route should match poorly with U1 and U3 routes, close to 0, but could score a bit higher, as the locations overlap with U3 route.

Test Execution
We had no influence over the actual method of the testing, but we received the following test results in Figure 4.5 from the outdoor group.

Test Results
The test results seen in Figure 4.5 reveal that the algorithm is performing correctly. The results correspond to the assumptions made during the data generation. All three routes of U1 are matched highly, as expected. The U3 route is matching reasonably well with the U1 routes, and neither of the U1 routes are matching with the U3 route. The results reflect that U3 can pick up U1 along the way while U1 will have to make a significant detour to pick up U3.

Test Conclusion
The algorithm test results are considered sufficient for the current application. However, testing, including user interviews, over large amounts of data for ac-

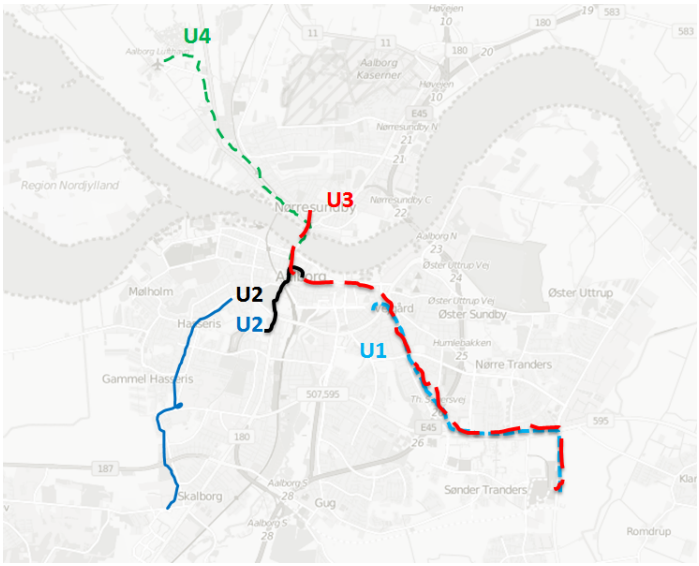


Figure 4.4: Generated routes to be tested by the algorithm.

	U1R1	U1R2	U1R3	U2R1	U2R2	U3R1	U4R1	U4R2
U1R1	1.000000	0.988664	0.916667	0.000000	0.000000	0.000000	0.000000	0.000000
U1R2	0.988664	1.000000	0.923623	0.000000	0.000000	0.000000	0.000000	0.000000
U1R3	0.916667	0.905331	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000
U2R1	0.000000	0.000000	0.000000	1.000000	0.000000	0.000000	0.000000	0.000000
U2R2	0.000000	0.000000	0.000000	0.000000	1.000000	0.000000	0.000000	0.000000
U3R1	1.000000	0.988664	0.916667	0.000000	0.000000	1.000000	0.000000	0.000000
U4R1	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1.000000	0.983333
U4R2	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.983333	1.000000

Figure 4.5: Results generated by the implemented algorithm.

tual users is necessary to confirm the practical convenience of the score results. The design of the algorithm allows adjustment of several parameter concerning distance, time and the weight of those in the final score by changing parameters in the RideShare server call to aSTEP system.

Chapter 5 Sprint 4

This chapter contains the documentation of the work performed in sprint 4. This is the last iteration and sprint of the project, thus, the sprint is focused on completing the solution and fulfilling the missing high priority requirements.

The documentation is structured chronologically and starts with the analysis of the requirements that is not yet fulfilled. Thereafter, the design section contains descriptions of the solution design for the necessary parts. Next, the implementation of the design is documented. The chapter ends with a test of the whole solution, as the development is stopped, and a status check is needed to evaluate the project and solution.

5.1 Solution Status Analysis

This section contains descriptions of the analysis performed in sprint 4. The analysis is performed to evaluate and review the current status of the solution, and hence enabling the further development of the solution parts.

The status is considered with regards to the requirement specification and the previously performed tests so that the solution can fulfill the requirements and pass an acceptance test. The analysis is divided into three sections, each dedicated to one of the three main parts of the solution.

The solution is analyzed throughout the previous sprints' analyses, and the necessary features are outlined in their respective sprints. The current analysis will be focused on revealing the lacking implementation of high prioritized requirements from the MoSCoW analysis performed in sprint 1, rather than analyzing new material.

5.1.1 aSTEP

The only part of aSTEP that is specialized to the requirements of the RideShare project is the algorithm, and it is tested and evaluated as functional, in Section 4.4.2. However, communication between aSTEP and the RideShare system must be tested to ensure a correct implementation of the API. No further development is currently considered necessary for the aSTEP part of the solution.

5.1.2 RideShare Server

The RideShare server is currently not implemented, but its functionalities and position in the RideShare solution are analyzed and designed in the previous sprints, e.g. in system design in Section 3.2.1. Because the RideShare server is sufficiently described in previous sections, there will be no further analysis of the server requirements.

The server must fulfill the requirement of storing contact information for the users, as this is not provided by the aSTEP system itself. Additionally, the RideShare server needs to retrieve the calculated matches from aSTEP system and store them locally on the server, enabling the RideShare app to display the matches to their respective users.

5.1.3 RideShare App

The visual interface of the RideShare app is currently consisting of a static, temporary placeholder list of two fictitious matches and an empty settings menu. The match interface needs to be implemented to comply with the requirement of displaying matches to the user. Furthermore, to separate user accounts and fulfill the requirement regarding login, the login interface and functionality must be implemented.

5.2 Design

The design of the implementation planned for sprint 4 is described in this chapter. The primary focus for the implementation is finishing the app and implementing the RideShare server. An important part is the implementation of the API calls to the aSTEP, as these are fundamental with regards to logging into the aSTEP system and to retrieve the route matches from aSTEP.

5.2.1 RideShare App

To make the RideShare app acceptable by fulfilling the 'Must have' requirements, the missing features from Section 2.2 must be implemented. A list of matches needs to be implemented so that the user interface reflects the respective users' matches. To implement the user matches list, the app must utilize the API calls regarding route matches. These calls must be in the correct aSTEP format to be accepted and to work properly. The actual API call could be generalized to a single class so it

can be reused for different API calls and to support potential future expansion. Every API call is decided to be in their own class for building the correct URL to be used with an API client.

It is important to make the API calls asynchronous for the user experience and to avoid user interface stalls. If the functions are made synchronous, the app will wait for a response for the API calls, and hence rendering the user interface unusable. The stall is affecting the whole app, and it could lead to other issues or failures. A list of the API calls that must be implemented to make the RideShare app functional can be found in Table 4.1.

5.2.2 RideShare Server

As described in Section 3.2.1, it was decided that additional user information not stored in the aSTEP server should be stored in a RideShare server. It was also decided in Section 3.2.1 that it is the RideShare server that is responsible for calling the aSTEP API to store the calculated matches, thus enabling the RideShare server to provide the route match information for the RideShare app. In order to store the data received from aSTEP, a database is needed on the RideShare server. The database needs two tables to perform the required operations: one for user data and one for matched routes.

As there are few tasks performed on the RideShare server, a simpler implementation of the API calls can be implemented. Both the API calls and the connection to the app should be implemented as asynchronous because there can be several requests to and from the server simultaneously.

5.3 Implementation

This section contains documentation of the implementation of the designs described in the previous section regarding the solution status. First, the RideShare server implementation is described, followed by the description of the app to aSTEP communication implementation.

5.3.1 RideShare-server Implementation

The server is implemented as a Java application running on a Windows machine. It communicates with the aSTEP API by using an HTTP connection class. The data sent through these connections will be explained in the following sections.

The app communicates with the server through the `ServerSocket` class and the `Socket` class. These classes provide input and output streams which can be written to and read from. A `BufferedStreamReader` reads the input stream as this class provides an easy to use interface for reading the input line by line. The server writes to the output stream using the `PrintWriter` class. This class is similar to the standard output stream in the way that it prints the string provided to the stream.

The messages send back and forth are of two types: Strings and JSON objects. The simple strings are used for error messages such as "Unknown message" and "Invalid token" while the JSON objects are used for more complex results such as arrays of matches.

These following two messages are used on the clients.:

Update matches	"`UPD:[TOKEN]`"
Register	"`REG:[TOKEN]:[PHONENUMBER]:[E-MAIL]:[FULL NAME]`"

The server checks the received token sent from the aSTEP API and gets the username in both cases. Update matches gets the updated set of matches for a user and register adds the user to the user details table in the database on the server.

The communication is contained in a loop that is implemented on the server, so that the server is ready for new clients as often as possible. The loop is configured as seen in the following list:

1. Load server configurations
2. Open Server side socket
3. If it is less than 24 hours since last match update from aSTEP goto 5
4. Create a thread to get match updates from aSTEP
5. Wait for a client
6. Create a thread to handle client request in another thread
7. goto 3

The 24-hour interval, in line 3, was chosen to reduce the load on the aSTEP server, and minimizes work while presenting users with relevant information. When receiving matches from aSTEP, the system runs the algorithms as described in Section 3.2.2 on routes that were received within the last two week. The matches are then stored in the servers own database.

All contact to the database is handled in the Database class. Because the class needs to load the database configurations from a file on the harddisk, it implements the singleton pattern so that the file only have to be accessed once. It would

be inefficient if the server had to read this file every time it needs to access the database.

The database is only containing two tables: user details and matches. The user details table keeps additional information about the users of the system, such as contact information and an alias they can use apart from their aSTEP username, stored as 'fullname'. The table description can be seen in Table 5.1.

Column	Type	Modifiers
username	character varying(50)	primary key
phonenumber	character varying(30)	not null
email	character varying(255)	
fullname	character varying(255)	not null

Table 5.1: The table definition of user details in the RideShare system.

A table of metadata for matches is stored on the RideShare server, and contains the information that is specified in the system design. The table description can be seen in Table 5.2, where the different data types and names are defined. The SQL statements used for creating the database can be found in Appendix A.

Column	Type	Modifiers
user1	character varying(50)	references userdetails(username)
user2	character varying(50)	references userdetails(username)
starttime	timestamp without time zone	not null
score	integer	

Table 5.2: The table definition of matches in the RideShare system.

5.3.2 Communication with aSTEP

This subsection contains an overview of the implementation of the API calls utilized to communicate with aSTEP. Both the API calls in the app and on the RideShare server will be presented.

RideShare App

In the design phase, Section 4.2, it was decided what API calls should be implemented in the app. A specified list of calls only to be performed on the app can be seen in Table 5.3 below.

Path	Method	Description
/locations/outdoor/{username}/routes	POST	Send a route to the aSTEP server
/users	POST	Create a new user
/users/{username}/token	GET	Get the current valid token for a user
/users	POST	Invalidate old token and get new
/users/{username}/token	POST	Invalidates previous token for a user

Table 5.3: Caption

The API calls are implemented using an `HttpURLConnection` class. This is a `URLConnection` class for HTTP-specific data transfer. `HttpURLConnection` is used to send and receive streaming data whose length is not known in advance, and this is needed as every API call can vary in length. E.g., the API call to post a route will be the one that varies the most in size as routes differ in number of locations.

Using an object-oriented structure, each API call is implemented to overwrite a `AsyncTaskAPIHelper` which will handle the inputs, such as username, password, token, and routes. The inputs are then transformed to a part of a URL which would fit each of the relevant API calls. The `AsyncTaskAPIHelper` result is then parsed to the `HttpURLConnection` which will use a string builder to create the final URL from the API base URL, `http://astep.cs.aau.dk:80/api/`. Lastly, the `HttpURLConnection` is trying to perform the API call from where response codes are handled.

It should be noted that passwords are not in the URL part of the API call, as these are logged in the aSTEP server. This would be a major security issue, because potential hackers could gain access to the passwords in plain text. Instead, password and tokens are parsed as part of the body or header, which is used by REST. This will hide the content from the URL and instead be transferred in an `OutputStream` encoded in raw bytes. Two examples of API calls can be seen in Table 5.4.

POST new user	
URL:	http://astep.cs.aau.dk:80/api/users?username=USERNAME
CURL:	curl -X POST -header 'Content-Type: application/json' -header 'Accept: application/json' -d 'PASSWORD' 'http://astep.cs.aau.dk:80/api/users?username=USERNAME'
POST route	
URL:	http://astep.cs.aau.dk:80/api/locations/outdoor/outdoor/routes?authorization=TOKEN &route=longitude;latitude;precision;timestamp&distance_weight&time_weight &largest_acceptable_detour_length=146&acceptablehttptime_difference=32h
CURL:	curl -X POST -header 'Content-Type: application/json' -header 'Accept: application/json' 'URL from above'

Table 5.4: Create user and send route to aSTEP API calls

As seen in 'POST new user', the username is passed in the URL while the password is included in a header. On the other hand, the 'POST route' is only passed in the URL as there is no information that needs to be hidden. The post route example

is simplified to reduce its size and make it easier to read. Longitude, latitude, and precision are all placeholders for doubles. The timestamp is in the ISO 8601 format with year, month, day, hour, minute, and second. The rest of the parameters is integers used to control the algorithm as explained in Section 3.3.3. In the given example the route given is only consisting of one location, for multiple locations, the URL would have to repeat the '&route=...' part.

Every decision made regarding parsing data in the header or body to the aSTEP server was made by the aSTEP API groups.

RideShare Server

A communication method with the aSTEP server is implemented on the RideShare server. The RideShare server only requires two API calls to comply with the requirements to the solution. One for getting matched routes from aSTEP and one for checking if a user is a valid aSTEP user. Both API calls are implemented as their own `HttpURLConnection`. The validate user check is performed when a user registers in the app. First a user is created at aSTEP then a request to the RideShare server is issued, where a check is performed to ensure the user correctly registered in aSTEP. The check is done by sending the token received from aSTEP to the RideShare server and then checked up against aSTEP again. If the user is valid the additional information, provided when registering, will be stored on the RideShare server for later retrieval.

The route match API call is called once a day as described in the previous chapter. The API call uses the token form the requesting user to fetch matches from the aSTEP server. aSTEP then returns every match for the particular user, with a score, the matched routes, and the full name of who the matches is with. This information is then send from the RideShare server to the application to be displayed. Currently, the only limiting factor is that only matches with users that are registered in the database are saved, but currently RideShare is the only app that uses the route functionality.

5.3.3 RideShare App

The app is further improved upon in this sprint. While the implementation of UI is considered trivial, it is not documented in detail in the report. The app is updated so that the route match users stored in the RideShare server are shown in the user interface at the main screen. The user matches are displayed similarly as in the prototype design of the app.

5.4 Test

In this final test section we will document the performed full system test. The test was an acceptance test, and was performed to verify that the solution complied with the established requirement specification. The acceptance test was designed to test the whole system with manually generated data. The scope of the test was to check that the functional 'Must-have' requirements from Section 2.2 are fulfilled. Some requirements were validated as functional in previous tests, and these are omitted from the acceptance test. First, the test is explained and described and, subsequently, the test execution and results are documented.

There have been extensive tests of the solution parts during the development, and the test results fulfilling requirements do not have to be performed again. 'User location tracking and storage', as well as 'automatic determining regular routes' were assessed acceptable in respectively Section 4.4.1 and 4.4.2. The latter is also partially accounting for the requirement 'suggesting ride partners' the only part missing for this requirement, was to check whether the ride suggestions were displayed correctly to the user in the app. The nonfunctional requirement of development cooperation was accounted for in Section 1.2. The remaining requirements that needed to be accounted for and acceptance tested were the following:

- Graphical user interface
- User accounts, including login and registration
- Communication with the aSTEP system
- Showing users their ridesharing matches in the app
- User privacy

5.4.1 Acceptance Test Design

This section is documenting the acceptance test design, and serves to describe the test method performed on the solution to evaluate the compliance with the requirements.

To make the app respond as it would in a real world test, mock test data was added to the RideShare server database. The content can be seen in Appendix C, and was consisting of user information as well as hard coded route matches. The data sets are a combination of data retrieved from the aSTEP database as well as data we constructed to generate matches for the test user `test`. As we were using specialized mock data for the purpose of testing, only the user `test` has any matches. The setup will consist of a computer running an instance of the

RideShare server and an android emulator. An Android emulator was used in the testing instead of an actual device so that we could avoid opening ports to the RideShare server.

The first task in the test consisted of opening the app and registering a new user with username `test` and the password `test`. If this succeeded, the user should then logged out and then logged in again with the same credentials. When the user is logged into the app again, the appropriate matches stored in the RideShare database will be fetched from the server and should be displayed in the app. If the lists were shown for the user `test`, but not for the newly registered user, the overall test would completed and acceptable.

The GUI would be tested through the availability of the different functions in the app. The user functionality will be tested when registering a user and logging in, this will concurrently also partially test the communication with the aSTEP system.

The rest of the requirements pertaining communication with aSTEP were verified in the tests performed in the previous sprints. If the matches were pulled correctly from the RideShare server and showed to the user, will be tested when the user `test` is logged in. The requirement concerning user privacy is tested by seeing if location data about other users are displayed to the user currently logged in.

5.4.2 Performing the Test

First a new user, `user22`, was created and this can be seen in Figure 5.1a. In the process of creating a new user, some unexpected behavior regarding Scandinavian letters were observed. Some additional tests were performed to further explore this observation, and they can be seen in Table 5.5.

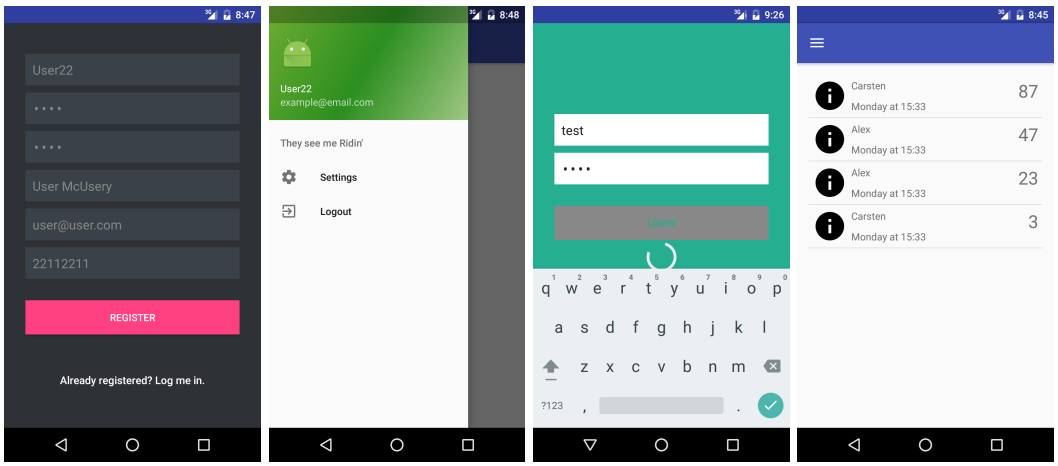
Username	Password	Result
a	a	200 OK
ø	ø	404 not found
å	a	404 not found
b	å	401 unauthorized

Table 5.5: Registered users login test.

Some additional aSTEP API test points to that the issue was a problem in the Android application as the special characters were working correctly when tested directly in the aSTEP API web interface.

The registering of the user `user22` was later successfully executed when using standard English letters and can be seen in the right side of Figure 5.1b. As expected,

no matches were found to the newly registered user `user22`. The user was then logged out of the app. The user `test` was then, as seen in Figure 5.1c, logged in successfully, and redirected to the main interface of the app, where the user was presented with the appropriate matches, shown in 5.1d.



(a) Registering a new user. **(b)** App drawer over empty mainscreen. **(c)** Login screen for a returning user. **(d)** Main-screen for an user with matches.

Figure 5.1: Screenshots from the acceptance test.

5.4.3 Test Conclusion

The acceptance test confirmed that the developed solution fulfills the functional Must-have requirements. The tests explored the full specter of the app functionalities. It can be concluded that the app is usable when both the RideShare server and aSTEP API is functional. The app is able to track a users location when driving in a vehicle, store the location data and display route match data from aSTEP when expected.

Chapter 6 Evaluation

This chapter contains the reflection, conclusion and future work. First we will reflect upon the project, with regards to both the internal group work and the multi-group experience. Thereafter, we conclude on our work for the project and solution. Last, we describe our thoughts about potential of the solution and possible future work.

6.1 Reflection

This section contains our reflection of the different project aspects. First we reflect on our own project, then the multi-group project is reflected upon and evaluated.

6.1.1 Project Reflection

The system design was influenced by the aSTEP progression during the development as the system relied on aSTEP. Throughout the project, there were multiple changes in the aSTEP system, which had different impacts on our project. Most significant was the change from aSTEP storing supplemental user information, such as e-mail address, to not storing anything other than username and password. The system design could have been better and there would have been more time for implementation if the final aSTEP system design had been established from the beginning.

The implementation was successful and dynamic, adjusting to changes made in the aSTEP project and our own tests performed throughout the sprints, in accordance with the iterative method. All group members were included in every part of the project. Each member had a possibility to discuss the individual parts, despite being assigned to another part during that time period. The tasks were evenly distributed on the group members, and more complex tasks were assigned to multiple members when necessary.

We were ambitious with the extent of the project, therein objectives regarding custom user settings of having a car or not, despite the project leader Bin Yang proposed low ambition levels and just making a working basis. Our ambition level is reflected in the requirement specification, and as seen in the acceptance test in Section 5.4, only the 'must have' requirements are fulfilled. The workload during the project has been high while there has been a joint attitude of making the

solution as high quality as possible. The ambition level and the established requirement specification together with unexpected task completion durations have caused delays during the sprints so that actual deadlines were sometimes not met. This happened during sprint 2 and 3, where our internal deadline would extend two weeks past the multi-group project deadlines. The missed deadlines had no effect on the aSTEP project but made our own scheduling harder to meet.

The solution development could have been more realistic if we had potential costumers to request features of us. This would apply more pressure to and reliance on the group collaborations and flow of information within the aSTEP project.

The RideShare app is a functional prototype, and is seen as a success since it utilizing the aSTEP system.

6.1.2 Multi-group Project Reflection and Evaluation

Through this semester, there has been a high focus on collaboration between the groups in this multi-group project. The collaboration effort consisted of two main parts: a weekly meeting where updates from every group were made, and shared group rooms. The factors made it possible to discuss important decisions, such as the structure of the aSTEP and requirements from the different groups, without scheduling a meeting elsewhere, while other groups were in the immediate vicinity. If one group needed to speak to another group, then they would quickly arrange that over Slack, that was the selected online communication platform for the semester project. Slack was a good choice as it provided great group structure and bot integrations, such as meeting reminders. The Slack platform enabled us to resolve small questions and to arrange meetings that helped in the collaboration effort.

We had collaborations with a few groups such as the Friend Finder group about the user interface aesthetics of the apps, to achieve a similar feel between the aSTEP apps, and the user management group about what user specific user features that were needed. Our tightest collaboration, however, was with the outdoor location based service (OD) group, as we developed and designed our route matching algorithm for them to implement. During the project, we had several meetings with them where we would discuss what we would need from the API and how implementation should be done to meet our requirements. In return they were concerned about performance and reducing very specialized implementations for apps in the aSTEP core as the core should be as generalized as possible. We later on participated in an API test with OD, where they were interested in whether we would understand and be able to use the API.

We did not have as many collaborations as some of the other groups, as we only re-

quested functionality and the aSTEP core would usually then implement it. But the collaboration we did have with the other groups were rewarding as we exchange ideas and thoughts.

Since this was the startup of a new project, there was confusion and frustration early in the period that had to be resolved. The lack of structure and detailed agreements sometimes resulted in a negative attitude towards specific people or groups who made small mistakes, or did not comply with the other groups expectations. The issues could have been avoided by having a dedicated scrum master for the overall project. We were already developing in an agile fashion so it would not be to hard to implement. In general, we feel that a better management and direction of the overall project was lacking. A possible solution could be stricter guidelines from the project leader, or to have some sort of administrative group to delegate responsibilities.

6.2 Conclusion

The RideShare solution, consisting of an Android application accommodated by an assisting server, utilizing the aSTEP system that was developed in this project, and this section documents the conclusion of the project.

The aSTEP system was developed concurrently with the RideShare solution, and both systems were dependent upon each other to be expedient. The parallel development was influencing the development of the RideShare solution, but also gave the possibility of establishing requirements for the aSTEP system. The cooperation development process gave challenges as the other parts of the system were continuously altered, but a stable design was finally achieved and lead to a system that was functional.

The solution fulfills the established requirements regarding both functionality and cooperation with the other semester project groups, as stated in the test in Section 5.4. There is improvement potential for the solution, but the 'Must have' requirements are met, and the solution is acceptable in this regard.

The project was aimed at solving the problem statement, as defined in Section 1.4:

How can one design and develop an app that automatically suggests ride sharing companions, based on common locations in origin and destination, utilizing the aSTEP platform?

The problem statement has been the initiating factor of the development process and was decomposed into multiple requirements. While the 'Must have' requirements are fulfilled, there are still remaining requirements regarding parts of the solution that are not completed. However, the 'Must have' requirements were sufficiently covering the minimum level of functionality, and hence, the problem statement is fulfilled and the solution is considered successfully accomplished.

The semester project, solution, and the multi-group project solution lays the foundation for a potentially comprehensive location based service for both indoor and outdoor positioning. Although the RideShare app could benefit from a user interface redesign in terms of aesthetics, it solves the problem statement and can be applied in the real world.

6.3 Future Work

This section contains description of work that could be done in future semesters, if the RideShare system should be refined and further advanced. The suggestions described in this section are partially based on the still not yet fulfilled requirements.

6.3.1 User Acceptance

As of this semester our system is solely developed based on our analysis of the interest and development in ridesharing solutions. Before further development it would be ideally to actual test whether the expected interest in the solutions actual exist. In addition validation of a marked for the solution, users interviews or similar research could also be used to evaluate the current state of the system and which changes or improvements users or potential users would prefer.

6.3.2 App Functionality

Currently, the app does not take users who are not driving into account. It could be useful to consider users who are biking or walking to work, and give them the opportunity to get a ride match. This could also lead to improving the algorithm, as to decide who can be matched against whom. For instance it would not make sense to see if someone who is riding a bike could pick up other users.

When using the current version of the algorithm it is implemented so that two users are matched with each other. But as it is now, only one of the users will receive the information about the match. The user receiving the notification is the

one who have the shortest detour to pick up the other, so he or she can decide if it is worth it. In the future a change could be made so that users who wants to be picked up could also see the matches and request a ride.

Registration of Users

The edges described in Section 4.2.1 was not implemented as they were not finished in the API. Implementation could be done during user registration with a few API calls to aSTEP. It could also be possible to integrate login possibilities such as Google or Facebook as alternative login methods and user info providers.

Another feature which can become important in the future is allowing users to use the same aSTEP user for different applications. This could be done by utilizing tokens and improving how they are handled in the aSTEP system.

User Interface

Since the user interface was not a priority of this project, it could benefit from an redesign. This redesign could include a more consistent visual theme between the views, and functionality that makes the app more easier to use. Profile pictures for accounts could make users more trustworthy to users whom they match with. More user information could also be displayed when tapping on a user match. Displaying details about a route that a user was matched with could be represented on a map, where a seemingly natural integration is with Google Maps.

Improvements is needed handling and displaying error messages users, which currently are mostly non-informative or non-existent, because many of those messages were written for development purposes.

6.3.3 Server

Some user interface changes will require server support. Currently, there is no support for storing profile pictures.

It could also be an improvement if the server could send push notifications with new routes matches to a user's app when detected, instead of the current implementation where the app request the server for updates when the app is opened.

The current option for establishing contact between users is by phone numbers. The user's phone number is sent to strangers also using the app, which could be unwanted by users. A better solution would be a messaging system, integrated in the app itself or another similar way of users whom matched, to communicate.

6.3.4 Algorithm

Several refinements can be made for the algorithm. Some actual user test of the algorithms accuracy and some more formal testing would give a better basic. Both in regards to deciding which parts of the algorithm could use improvement most, as well as gathering some information about which parameters are most important to user when considering possible matches.

The input parameters of the algorithm could also be evaluated or expanded upon. As mentioned in 2.1 user could specify preferences of the people whom they will be matched against, i.e. gender, or if a person smokes.

One possible thing to improve upon is the search space for matches. The algorithm does not use any filtering of routes other than a simple time comparison. Additional filtering could be done by using geo-fencing. Another less complex solution could be to truncate the coordinate in to a lower accuracy location. By lowering the number of coordinate decimals in the accuracy, a larger margin of precision is expressed, with the consequences of making a 'grid' as can be used for filtering. Truncating the location values down to two decimals as described by Zhang [28], for example, yields an accuracy of 1.1 kilometers.

One of the most important feature improvements to the algorithm, is to consider maps, speed limits, traffic data and terrain when calculating detours. This feature could increase the accuracy of the estimated detour time and distance, and hence yield more accurate scores. It could be implemented by utilizing the Google Maps API.

The above documented possible future work does not reflect all possible directions which the project could take, but rather it is a reflection of the thoughts and discussions which our work on the system has fostered.

Appendix A Database Implementation

```
CREATE DATABASE ramsydata;
CREATE TABLE userdetails(
  username VARCHAR(50) PRIMARY KEY,
  fullname VARCHAR(255),
  phonenumber VARCHAR(30),
  email VARCHAR(255)
);
?
CREATE TABLE matches(
  user1 VARCHAR(50) REFERENCES userdetails(username),
  user2 VARCHAR(50) REFERENCES userdetails(username),
  matchfound DOUBLE PRECISION,
  lastUpdate TIMESTAMP,
  score INT,
  PRIMARY KEY(user1, user2, matchfound)
);
```

Appendix B Sprint 3 API functionality

In the following table are the API functionality as they were in sprint 3 with regards to UM and outdoor location based service. Indoor location based service is omitted. The list is based on the information available at `http://www.astep.cs.aau.dk/`. At the URL the most recent version of the API will be available at any given time.

Outdoor location services	User Management
Get user past location	Create group
Get user past locations area	Get administrated groups
Get user past Locations radius	Add administration
post user current location	Remove administration
Post user current route	Get invited users
Get all new route matches	Invite user to group
Get all locations friends	Revoke group invitation
Get all users in area	Get members of a group
Get locations friends	Remove user from group
Get subset of users area	Create user
Get users outside area	Get groups that a user is invited to
Get subset of users radius	Decline group invitation
Get locations friend timestamp	Get groups that user is member of
	Join group
	Leave group
	Get outdoor-user
	Request outdoor user
	Delete outdoor-user
	Validate outdoor-user
	Change password
	Get token
	Issue token

Table B.1: Currently planned aSTEP API functions.

Appendix C Sprint 4 RideShare database test data

```
SET statement_timeout = 0;
SET lock_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SET check_function_bodies = false;
SET client_min_messages = warning;
SET row_security = off;
-- Name: plpgsql; Type: EXTENSION; Schema: -; Owner:
CREATE EXTENSION IF NOT EXISTS plpgsql WITH SCHEMA pg_catalog;
-- Name: EXTENSION plpgsql; Type: COMMENT; Schema: -; Owner:
COMMENT ON EXTENSION plpgsql IS 'PL/pgSQL procedural language';

SET search_path = public, pg_catalog;
SET default_tablespace = '';
SET default_with_oids = false;
-- Name: matches; Type: TABLE; Schema: public; Owner: postgres
CREATE TABLE matches (
    user1 character varying(50) NOT NULL,
    user2 character varying(50) NOT NULL,
    matchfound double precision NOT NULL,
    lastupdate timestamp without time zone,
    score integer
);
ALTER TABLE matches OWNER TO postgres;
-- Name: userdetails; Type: TABLE; Schema: public; Owner: postgres
CREATE TABLE userdetails (
    username character varying(50) NOT NULL,
    phonenumber character varying(30),
    email character varying(255),
    fullname character varying(255)
);
ALTER TABLE userdetails OWNER TO postgres;
-- Data for Name: matches; Type: TABLE DATA; Schema: public; Owner: postgres
COPY matches (user1, user2, matchfound, lastupdate, score) FROM stdin;
niels   alex    1463383750000   2016-05-16 07:29:37.902 20
alex    carsten 1463383750000   2016-05-16 07:29:37.984 50
svend   carsten 1463383750000   2016-05-16 07:29:37.986 70
svend   alex    1463383750000   2016-05-16 07:29:37.987 90
test    alex    0   2016-05-16 15:33:19.818857  23
test    alex    1   2016-05-16 15:33:19.818857  47
test    carsten 1   2016-05-16 15:33:19.818857  3
test    carsten 21  2016-05-16 15:33:19.818857  87
.
-- Data for Name: userdetails; Type: TABLE DATA; Schema: public; Owner: postgres
COPY userdetails (username, phonenumber, email, fullname) FROM stdin;
test    +4511223344 N   Susie Hund
alex    +4512345678 N   Alex
carsten +4512345678 N   Carsten
svend   +4512345678 N   Svend
```

```
50 niels   +4512345678 N   N
51 wiingreen   +4542267385 N   Claus Worm Wiingreen
52 .
53 -- Name: matches_pkey; Type: CONSTRAINT; Schema: public; Owner: postgres
54 ALTER TABLE ONLY matches
55     ADD CONSTRAINT matches_pkey PRIMARY KEY (user1, user2, matchfound);
56 -- Name: userdetails_pkey; Type: CONSTRAINT; Schema: public; Owner: postgres
57 ALTER TABLE ONLY userdetails
58     ADD CONSTRAINT userdetails_pkey PRIMARY KEY (username);
59 -- Name: matches_user1_fkey; Type: FK CONSTRAINT; Schema: public; Owner: postgres
60 ALTER TABLE ONLY matches
61     ADD CONSTRAINT matches_user1_fkey FOREIGN KEY (user1) REFERENCES userdetails(↩
       username);
62 -- Name: matches_user2_fkey; Type: FK CONSTRAINT; Schema: public; Owner: postgres
63 ALTER TABLE ONLY matches
64     ADD CONSTRAINT matches_user2_fkey FOREIGN KEY (user2) REFERENCES userdetails(↩
       username);
65 -- Name: public; Type: ACL; Schema: -; Owner: postgres
66 REVOKE ALL ON SCHEMA public FROM PUBLIC;
67 REVOKE ALL ON SCHEMA public FROM postgres;
68 GRANT ALL ON SCHEMA public TO postgres;
69 GRANT ALL ON SCHEMA public TO PUBLIC;
```

Bibliography

[1] *traffic jam - definition of traffic jam in English from the Oxford dictionary*. 2016. URL: http://www.oxforddictionaries.com/definition/english/traffic-jam (visited on 03/24/2016).

[2] Matthew Barth and Kanok Boriboonsomsin. *Real-World CO2 Impacts of Traffic Congestion*. College of Engineering - Center for Environmental Research and Technology. 2008. URL: http://uctc.net/research/papers/846.pdf (visited on 05/03/2016).

[3] *InformalArchitecture.png*. 2016. URL: http://daisy-git.cs.aau.dk/Yang/astep/uploads/5891b5377ac9d75841374e0711d22078/InformalArchitecture.png (visited on 05/19/2016).

[4] Nelson D. Chan and Susan A. Shaheen. "Ridesharing in North America: Past, Present, and Future". In: *Transport Reviews* 32.1 (2012), pp. 93–112. DOI: 10.1080/01441647.2011.621557. URL: http://dx.doi.org/10.1080/01441647.2011.621557.

[5] Andrew Amey, John Attanucci, and Rabi Mishalani. "Real-time ridesharing: opportunities and challenges in using mobile phone technology to improve rideshare services". In: *Transportation Research Record: Journal of the Transportation Research Board* 2217 (2011), pp. 103–110.

[6] *T-share: A large-scale dynamic taxi ridesharing service*. English. 2013, pp. 410–421. ISBN: 978-1-4673-4909-3. DOI: 10.1109/ICDE.2013.6544843. URL: http://dx.doi.org/10.1109/ICDE.2013.6544843.

[7] Keivan Ghoseiri et al. *Real-time rideshare matching problem*. Mid-Atlantic Universities Transportation Center, 2011.

[8] *Ride-Hailing Apps Go the Extra Mile*. InTransition. 2015. URL: http://www.intransitionmag.org/spring2015/ride_hailing_in_suburbs.aspx (visited on 03/31/2016).

[9] Xun Li et al. *Smartphone Evolution and Reuse: Establishing a more Sustainable Model*. 2011. URL: https://www.cs.ucsb.edu/~franklin/cv/pubs/35GreenCom10.pdf (visited on 05/24/2016).

[10] DSDM. *MoSCoW Prioritisation*. Ed. by DSDM. DSDM. 2014. URL: https://www.dsdm.org/content/moscow-prioritisation (visited on 05/18/2016).

[11] David Benyon. *Designing Interactive Systems*. Pearson, 2014.

[12] Google?, ed. *Introduction - Material Design - Google design guidelines*. Google. URL: https://www.google.com/design/spec/material-design/introduction.html#introduction-principles (visited on 02/23/2016).

[13] *Material properties - What is material? - Google design guidelines*. Google. URL: https://www.google.com/design/spec/what-is-material/material-properties.html (visited on 02/24/2016).

[14] *<uses-sdk> | Android Developers*. Google. URL: http://developer.android.com/guide/topics/manifest/uses-sdk-element.html (visited on 02/02/2016).

[15] *Support Library*. Google. URL: http://developer.android.com/tools/support-library/index.html (visited on 03/02/2016).

[16] *Dashboards | Android Developers*. Google. URL: http://developer.android.com/about/dashboards/index.html (visited on 03/14/2016).

[17] *Android 5.0 APIs*. Google. URL: http://developer.android.com/about/versions/android-5.0.html (visited on 03/02/2016).

[18] Jeff Friesen and Dave Smith. *Android Recipes: A Problem-Solution Approach for Android 5.0*. 4th. Apress, 2015.

[19] *Overview of Google Play Services*. Google. URL: https://developers.google.com/android/guides/overview#the_google_play_services_apk (visited on 03/02/2016).

[20] *Overview of Google Play Services*. Google. URL: https://developers.google.com/android/images/play-services-diagram.png (visited on 03/02/2016).

[21] *FusedLocationProviderApi*. Google. URL: https://developers.google.com/android/reference/com/google/android/gms/location/FusedLocationProviderApi#public-methods (visited on 03/02/2016).

[22] *ActivityRecognitionApi*. 2016. URL: https://developers.google.com/android/reference/com/google/android/gms/location/ActivityRecognitionApi#public-methods (visited on 05/19/2016).

[23] Dr. M. Elkstein, ed. *Learn REST: A Tutorial*. 2008. URL: http://rest.elkstein.org/ (visited on 03/14/2016).

[24] *Handler | Android Developers*. Google. URL: https://developer.android.com/reference/android/os/Handler.html (visited on 04/05/2016).

[25] Robert Love. *Why does GPS use so much more battery than any other antenna or sensor in a smartphone?* Ed. by Quora. Quora. 2013. URL: https://www.quora.com/Why-does-GPS-use-so-much-more-battery-than-any-other-antenna-or-sensor-in-a-smartphone (visited on 05/09/2016).

[26] *Receiving Location Updates | Android Developers*. Google. URL: https://developer.android.com/training/location/receive-location-updates.html (visited on 05/06/2016).

[27] Techradar, ed. *Mobile coprocessors: the secret to smarter smartphones*. Techradar. 2014. URL: http://www.techradar.com/news/phone-and-communications/mobile-phones/mobile-coprocessors-the-secret-to-smarter-smartphones-1218012 (visited on 05/09/2016).

[28] Sarah Zhang. *How precise is one degree of longitude or latitude?* Gizmodo. 2014. URL: http://factually.gizmodo.com/how-precise-is-one-degree-of-longitude-or-latitude-1631241162 (visited on 05/24/2016).