Rideshare Automatic Matching System

Ridematching Android app utilizing aSTEP

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:
Project Title

Theme:
Scientific Theme

Project Period:
Spring Semester 2016

Project Group:
SW605F16

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

Supervisor(s):
David Frazetto

Copies: 1

Page Numbers: 59

Date of Completion:
May 17, 2016

Abstract:

Here is the abstract

Preface

Here is the preface. You should put your signatures at the end of the preface. We should have things here like, astep, multi-group, thank davide, android, etc.

Aalborg University, May 17, 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

Todo list

Chapter 1 Introduction

Commuters are causing traffic jams in dense areas traveling similar routes, this project embarks the task of informing the individual commuters of other commuters traveling a similar path, so that they could share vehicle. A such task also seeks to reduce the commuters impact on the environment.

1.1 Project environment

This project is part of a collaboration project, AAU Spatio-TEmporal data management Platform (aSTEP), of the SW6F16 semester. The semester goal is to develop a location based service with accompanying applications that utilizes the service.

The aSTEP project is developed in 10 teams in the span of 4 sprints. The sprints are synchronization points of the project groups, and this report will contain documentation of each sprint chronologically. ~~There are four sprints in the project period.~~ The first sprint is intended for analysis and collaboration development, the second and third are development sprints, and the fourth is intended for test and finalization. The 10 project groups are assigned to their respective tasks, with 7 project groups developing the core aSTEP service, while the remaining ~~3~~ three groups are developing applications utilizing the system.

1.2 Problem domain

~~The problem domain of this project is to design and develop a solution which utilize the system , and~~ Because we must use aSTEP, we came up with the idea of a system which can identify and help drivers and passengers with arranging ridesharing. The solution's goal is to provide ridesharing suggestions to the users of the solution. 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 **doi:10.1080/01441647.2011.621557**.

The intention of this project is to produce ~~a~~ an android app that utilizing the aSTEP system and based on historical location data should ease the possibility of using and arranging carpool-style ridesharing.

The scope of such a project can be large and therefore it is chosen early to that the main focus will be to design and implement the basic system which then later

can be expanded on. ~~This focus revolves~~ The purpose of the application will be to propose appropriate candidates for ridesharing. This will revolve around develop and implementing an algorithm that can be utilized to asses if two users of the system should be paired for ridesharing. ~~Hence the purpose of the application will be to propose appropriate candidates for ridesharing.~~ Furthermore, other parts of the application such as graphical user interface is not prioritized, but is developed to be functional enough for test and practical purposes.

Such an application could be beneficial for the users in several ways. There could be environmental and economical savings, as the number of polluting cars driven is reduced as described in [**doi:10.1080/01441647.2011.621557**]. The application would ideally decrease ~~trafical~~ traffical problems such as traffic jam and thereby also being time-saving, and causing less frustration for the drivers.

There are some potential for both social, environmental and economical benefits, as the app allows people meet each other and thus reduce the use of cars, gasoline and the load on the road system.

The future aspect of this solution seem bright since it easily should modified to work for autonomous vehicles.

To define the actual problem, as to initiate the research and development, a concrete formulation is made.

1.3 Problem Statement

Sharing rides is advantageous in 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 [**doi:10.1080/01441647.2011.621557**]. 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?

~~The problem statementis split into several requirements in different areas .~~ To successfully solve the problem statement, we define requirements with regards the different areas of the development and final product.

1.4 Requirement Specification

To provide a clear direction of the solution and to state the success criteria, this section contains the requirements for the solution. The requirements are divided into two main categories: functional and nonfunctional requirements. The requirements are sorted in the MoSCoW structure, enabling the project group to solve the highest prioritized requirements first, hence developing a solution that fulfills the core purpose before adding additional functionality.

add reasoning and source of MSCW categories + functional/non-functional

1.4.1 Functional requirements

Functional requirement are requirements directly related to the tasks and operations of the developed application.

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

Graphical user interface
The application must have a graphical user interface (GUI), as user must be able to operate the application them selves. The GUI must be intuitive, as user may have different experience with using mobile applications.

User accounts, including login and registration.
Unique user accounts is required as it serve as a identifier for each user, using the system. With user accounts, store personal information, and compare user based on the data. It will also provide function for displaying user to other users.

Communication with the aSTEP core.
As the project is a part of the bigger system aSTEP, there mush 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.
As the application must track users as they move around, location sensors must be polled, usually GPS. The application should track users commute. This data should be stored in the aSTEP database.

Automatically determine regular routes.
Automatically determine regular routes the user take. Regular routes should be saved for later comparison. Discarding routes that are not used often.

Automatic ride sharing recommendations.
A automatically comparison with other users routes must be computed. When two

regular routes are found similar, each user must be notified of such, whiteout any user input.

Should-have requirements
The requirements in this subsection are requirements that are important, but not regarded as highly critical.

Give the user option to specify wherever they have a car or not.
Some users might not have a car, and that should be considered as people whiteout car would need to be treated differently from those who have.

Enable user to blacklist other users.
Giving they are frequent users of the application, and have a bad experience with either a driver or passenger, they should be able to blacklist them.

Suggest rides with user who only drives a subset of the way from A to B.
Giving users the option to drive with people who do not have the same source and destination, will increase the pool of which users to suggest, as user can ẗag alongp̈art of the ride.

Could-have requirements
These are the lowest realistic requirements.

Ride reservation or request from, to, time.
Reserve rides with other users. This includes both regular commutes and commutes users would do rarely.

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 economical savings due to their use of the solution.
Provide detailed information of how much fuel users save, how much money saved based on fuel prices, and how much CO2 that is not released into the environment.

1.4.2 Non-functional requirements

The nonfunctional requirements for the solution are stated here.

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

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

User privacy
The application must respect user privacy, especially in regards to whom have access to a user location data and general user data. A solution will be developed in collaboration with other groups of the aSTEP project. The solution should consider elements such as, database storage, user management and other developers.

Should-have requirements
The requirements in this subsection are requirements that are important, but not regarded as highly critical.

Aesthetics matching other aSTEP project applications
The application must be of the same design and guidelines as the other applications developed for aSTEP.

Could-have requirements
These are the lowest realistic requirements.

Placeholder

find requirement

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.

find requirement

Chapter 2 Group Role

The aSTEP is a multi-group semester project, and consists of several parts. This project group is responsible for developing an app that utilizes the aSTEP core, and contribute to the development of the interface of the core.

Insert some shit figures of the aSTEP core structure draft.

Chapter 3 Sprint 1

The first sprint concerns with getting the project started, both the aSTEP project as a whole and the ridesharing project described in this chapter. 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 two contributing parts of sprint 1 will be the initial analysis and overall design.

3.1 Analysis

Ridesharing is an activity which has had phases of popularity in recent history according to **doi:10.1080/01441647.2011.621557**. They describe the current phase as the technology-enabled ridematching phase which started 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. **doi:10.1080/01441647.2011.621557** also list incentives for ridesharing in the current phase, which are the following

- "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" [**doi:10.1080/01441647.2011.621557**].

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 a long 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 **amey2011real** defined as "rideshare service relying heavily on mobile phone technologies".

3.1.1 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.

doi:10.1080/01441647.2011.621557, **amey2011real** sees technology as one of the most important factors in the present and future of dynamic ridesharing.

ShuoMa2013 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.

ghoseiri2011real researched what 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 [**ghoseiri2011real**]. This algorithm might be useful in this project solution.

Actual choices and influences regarding algorithm design for ridematching will be addressed in the design phase in Section 3.2.

3.1.2 Commercial Solutions

Corporate and community solutions are quite different from the academic field solutions. The main focus seems to be centered more around taxi alternatives. The are plenty of services around the world, with the two biggest international services in this field as of 2015 seems to be Uber and Lyft[**ridehail**]. There are also several services with some or all of their focus in Denmark, most notable is services like Haxi and Drivr. These services work mostly in a similar way:

source

1. A customer request a ride through an 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 service's design or architecture is not available. Drivr has two interesting feature in comparison to their opponents, that is a web interface for fleet manage-

ment and business which provides administration and expense control of employee taxi travels.

Besides these taxi-like services, there also exist services that focus on traditional ridesharing between private people, where money is not earned. Here exist both local and international services. These services give the opportunity to either offer a lift, request a ride and the possibility to connect drivers and passengers. These services are usually free, but some of them charge a small fee when connecting people. Some of the more notable are GoMore and iRideshare. source

The smartphones have automatized much of the labor concerning organizing ridesharing and act as a central component in the analyzed systems. However, we still see an opportunity to utilize the smartphone even further in a service that arranges rides between users, thus decreasing the actual required user actions. add arguments

3.2 Design

The temporary design is based on the preliminary analysis of sprint 1, and will outline the main features of the design, concerning system structure and components, and the user interface. add metatext

3.2.1 System design

This section contains documentation of the system design, consisting of the structural and behavioral design. add metatext

Structual Design

The overall structural design of the system is based on the analysis of the first iteration.

To exploit 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 have their own delegated responsibilities, and will perform the necessary tasks, enabling the system to deliver the service. define solution service in analysis/problem?

The application, hereafter referenced as the app, is the program executed on a mobile device, and will gather location data and serve as the user interface.

The solution is assumed to require much processing power and to access location data to analyze the possible combinations of routes. Computing the route combinations in an app would require that each user has a copy of every other users' routes. This is a serious privacy concern as this would enable other applications to leach on to the aSTEP system a gather data about when people are home and where they live. Other concerns include, for calculating the best routes a lot of processing is required draining the battery and producing heat.

clean up battery and privacy references

When under the assumption that a central server can handle the analysis of the routes, the application need only to supply the server with location data to analyze.

Figure 3.1: The overall structure.

A background service would enable the location data sampling and transfer to a server. As seen in Figure 3.2 the Observer class' main goal is to observe what activity the user is currently doing. In Google's API for android applications this is already a feature, so this will be used. If the activity is driving, the background service starts to store the locations with the RouteBuilder class. Locations are also a part of Google's API and includes longitude, latitude and a time stamp. When the Observer detects that the user stops driving, the Observer sends the route to the server.

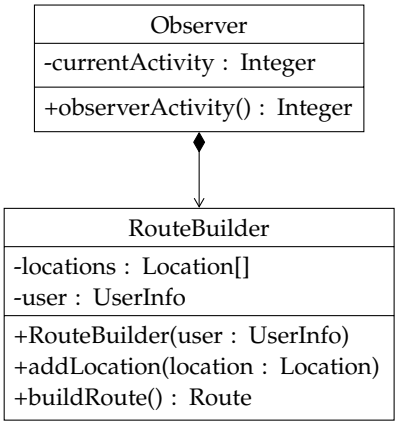


Figure 3.2: The small process which should run at all time to build the routes.

On the server, the route is stored in a database and accessed by the RouteStabilityAnalyser. 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 information is

stored in the database with when this route is expected to occur. 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 to inform the users of a new match.

does not need to be a new stable route??

fix: "This package should be an isolated system within the server."

The application acts as an interface to the background process for the user. This is where the user logs in and can see an overview of candidates for car pooling.

This is the start of a project that will be further developed upon and expanded in the future, emphasizing the importance of reusing the server, to reduce the amount of specialized server resources. By isolating parts that are specific to the rideshare app on the server, most of this system would be reusable. The best example of this is the RouteAnalyser from Figure 3.3. The RouteAnalyzer filter routes according to its concrete implementation. After the filtering, the RouteAnalyser uses the strategy pattern to determine how to compare the routes. This enables other projects to develop their own filters and comparison functions in later projects.

To reduce the search space, a weekly stability was decided upon. This means that the RouteStabilityAnalyser can be limited to analyze one to two months of data. This will be further tested and evaluated in another chapter.

did we?

reference to test and evaluation

add arguments

3.2.2 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, like testing, and to lay the basis for further development, but is still held to some standard which will be documented in the following text.

Design language

The design language is the general look and appearance of a system. This is the foundation for the impressions and feels of the user, 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 [**DIS2014**]:

- Learnability

- Utility
- Safety
- Effectiveness

The UI is designed to comply with the Google's Material design guidelines [**materialDesign**], being "*bold, graphic, intentional*". Some of the Material design properties are stated in the following list, compiled of citations from the design guidelines [**materialProperties**]:

1. Material has varying x & y dimensions (measured in dp) and a uniform thickness (1dp).
2. Material casts shadows. Shadows result naturally from the relative elevation (z-position) between material elements.
3. Content is displayed on material, in any shape and color. Content does not add thickness to material.
4. Input events cannot pass through material.
5. Material cannot pass through other material. For example, one sheet of material cannot pass through another sheet of material when changing elevation.
6. Material grows and shrinks only along its plane.
7. Material never bends or folds.
8. Material can be spontaneously generated or destroyed anywhere in the environment.

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

The design drafts in Figure 3.5 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 login if they already have a user profile, or create a new user profile.

Several views are necessary to support the different parts of the app. The app needs a view for login, registering, presentation of matches, and a settings screen, as can be seen in the figures in Figure 3.6.

As the application must have communication to the aSTEP server, which have an requirement of registration of users, it is necessary to have a register and login screen in the application. The username will be used as a identifier for the user, and it will be used both for the aSTEP server, but also for the application it self. It will be the identifier to differ between each user, and ensure users are matched up with other users correctly when comparing routes.

The application must have a main screen, and it is chosen that should be the route match list. On this screen the user will be presented a list of other users, which most likely is a good match. The users listed should be in an descending order, with the highest rated match at the top. A short description about the users and match should be displayed, such as the users full name, phone number, the match score, and what day of the week the user drive the relevant route.

A settings screen is needed because the users should have the option to change there initial information from when they registered. The settings screen also server as a screen where users can specify their preferences. Such information is, wherever the users have a car with X numbers of seats and if the users is looking for, or giving a lift. In addition should there exist a navigation drawer from where the user can navigate around the application.

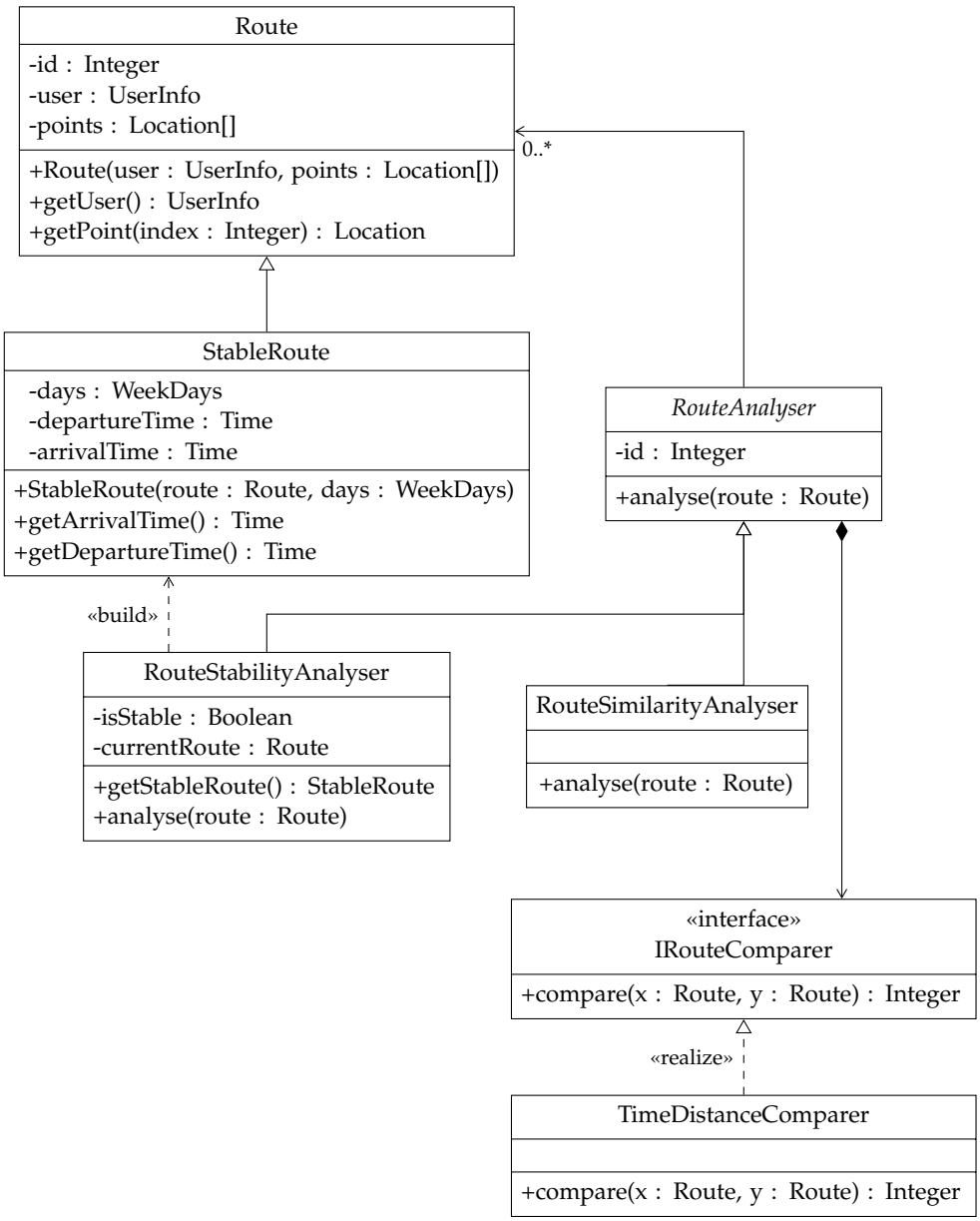


Figure 3.3: The structure of the system inside the server. This package should be an isolated system within the server.

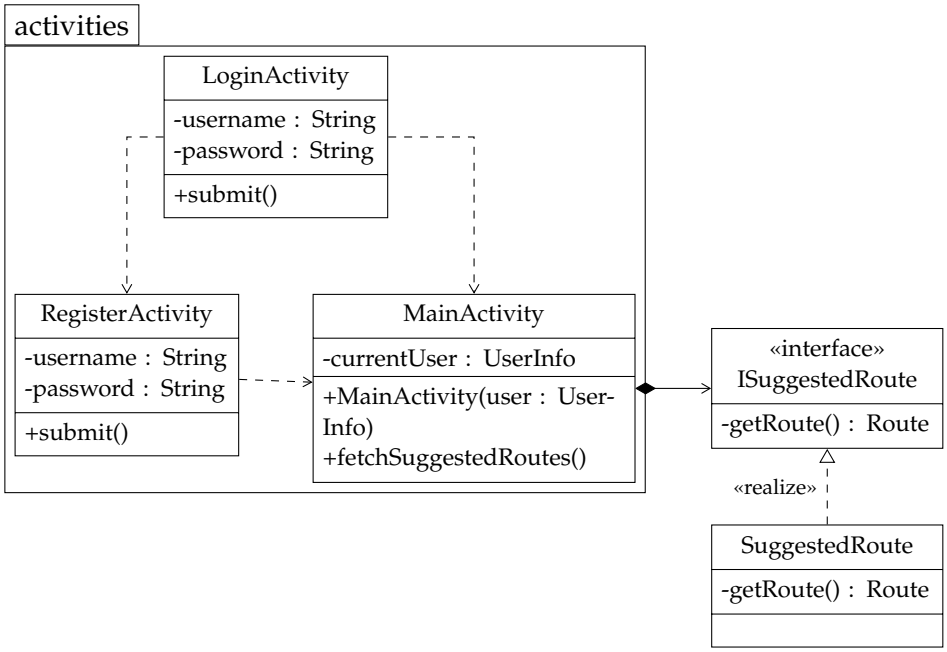
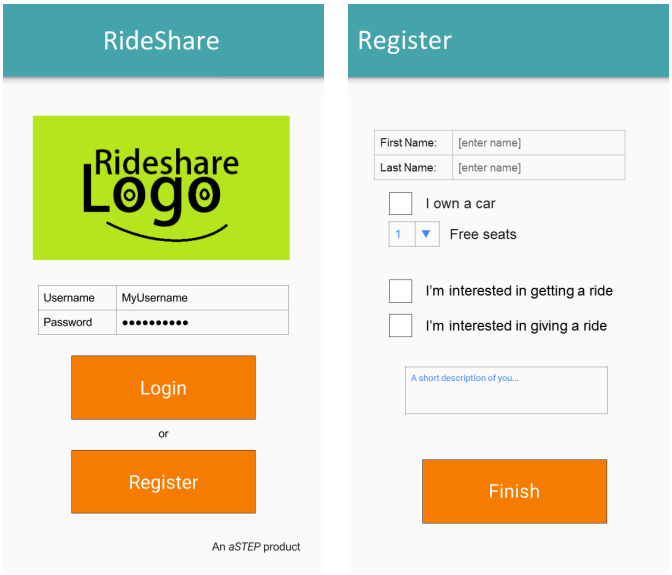
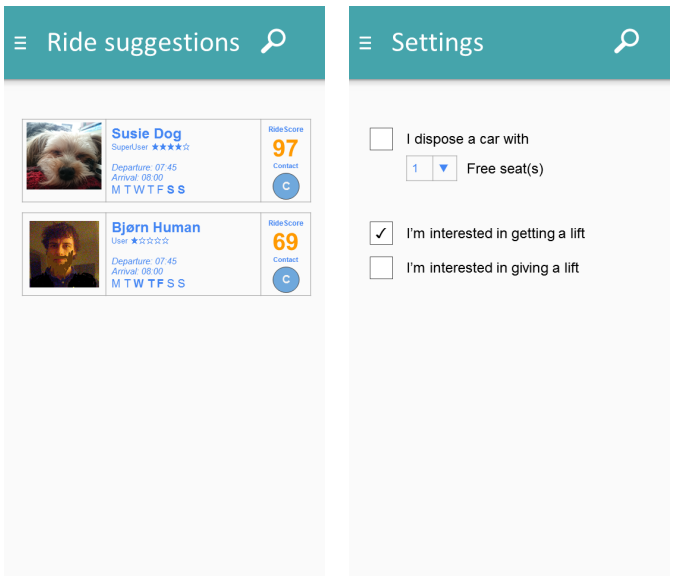


Figure 3.4: The class structure of the application.



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

Figure 3.5: Draft of the login and registering pages of the app.



(a) Main page **(b)** Settings page

Figure 3.6: Draft of the views in the app.

Chapter 4 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 of 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 and improved upon in sprint 3.

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

4.1 Analysis

This section contains the analysis of app functionality, Android app development, Rideshare Automatic Matching SYstem (Ramsy) algorithms and communication with the aSTEP system.

4.1.1 App functionality

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

To be able to separate different routes, and to assign them to their respective users, there has to be a user management system, so that each user has its own 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 user of the application 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 it wants to get a ride or give a ride or both. There are also properties of the user itself, like a description and a picture, that other users can access to decide if they want to share a ride with the named user.

4.1.2 Android Development Platform

When developing Android applications one of the first choices to consider is which API levels to target [**usesSDK**]. 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 [**usesSDK**]. The distribution of the Android version levels are shown in 4.1.

"*The manifest file presents essential information about your app to the Android system, information the system must have before it can run any of the app's code.*", according to Android Developers [**androidManifest**]. The manifest requires to 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.

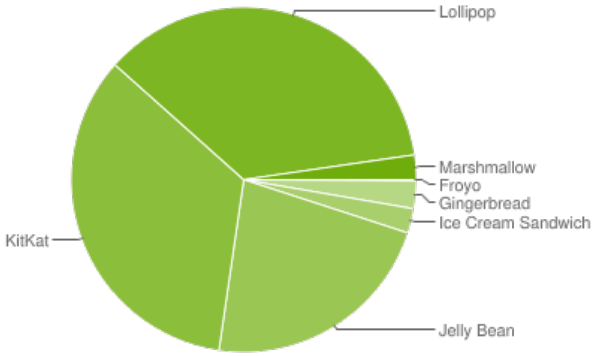


Figure 4.1: Android version distribution, March 2016 [**androidDashboard**]

The last API level which needs to be specified is the "minSdkVersion". This entry specifies the lowest API level which an app supports. A cause of this is that every device with an API level lower than the specified minSdkVersion will not be compatible with the app. Supporting older API versions requires development efforts, typically implementing functionality though the Android Support Library [**androidSL**]. For this project we choose to target a higher API level for 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%. API version 21 is chosen as the minSdkVerison, the main reason for this choice is to minimize development time for user interface, as API level 21 added support for Android's new material design style [**android5API**]. Refraining from further backwards compatibility allows the project focus to be on the functionality of the app.

4.1.3 Periodic Tasks

When the app is installed the program needs to run a task periodically to cache locations. **friesen2015android** writes about how this can be done in the Android API with the AlarmManager and the JobSchedular.

AlarmManager In the first Android version (API 1) the AlarmManager was created to handle alarms. They implemented it as a general class which can call any task after a given period has passed. The alarm can then be set to be reoccuring which means that we can use this to cache location data in a set interval.

JobSchedular In Android 5.0 (API 21) an alternative to the AlarmManager was implemented, the JobSchedular. This functions in a similar way as the AlarmManager, but tries to bach the jobs together and fire them at the same time. This means that the device can save power by avoiding going to sleep just to wake a short moment later. The sacrefice is precision. While the AlarmManager has the option to occur at a exact time, the JobSchedular does not. This is needed to give the AlarmManager enough control to effeciently bach the jobs together.

Because the timing of the location data is less important than the constant stream of data the JobSchedular. This has the advantages of saving power making it more likely that the device will stay powered for the duration of the trip. And by extention giving a complete route.

Something is missing

4.1.4 Algorithms for ridesharing

4.1.5 Communication with aSTEP

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

The status and intended functions are established in collaboration with the other project groups responsible for the aSTEP system side of the development. The main groups we are involved with are the user management group and outdoor location based services groups.

The design of the aSTEP system at this point is 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. However, the only

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 4.1: Currently planned aSTEP API functions.

relevant part for the Ramsy solution for us is the API, as the lower level of the aSTEP core is administrated by other groups.

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, a set of 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 is made to ensure the correct permissions is giving to the correct user, to ensure the appropriate data is returned to each user. An API call can not be done without the user first being authenticated with a valid login.

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

reference this table. Add information about the functions.

At the current stage of the development of aSTEP, the following API functions are available form user management and location services:

When using the aSTEP API it should be ensure to use the correct calls in the right

way. The API is providing a POST-request under the name “PostLocationDat”, as listed in 4.1, which is the method to use when sending location data to the aSTEP core. When using this POST it is important to ensure the correct parameters is used. The call will accept location data as an coordinate consisting of longitude and latitude, a precision value, and a value representing time of day in milliseconds.

4.1.6 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 will be presented.

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

User Data
User information such as username, password, and login token, must be analyzed to figure out where this information should be stored, and how it should be handled regarding communication between the application and the aSTEP server.

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

System Architecture and Communication
Communication between the Ramsy 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. It should be considered how to handle storing the collected location data and how much should be stored where, either on the device or on the aSTEP server.

Mock Data
If there is spare time during the sprint, it should be considered to acquire mock up location data related to mock up users. This data could be used in the third iteration to test if the algorithm works as intended.

4.2 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.

4.2.1 System design

In the first iteration, the Ramsy system was designed to consist of two parts: The Ramsy 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.

?

The aSTEP system still provides the functionality to store locations and route history for users. The Ramsy app continues to collect device location data. The Ramsy solution will need additional data storage to fulfill the defined system requirements, because the Ramsy users need to be able to contact each other, to make the Ramsy app usable.

smt missing?

try to change the structure of this this sencente

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.

A solution that we regards as appropriate for the task is using a custom solution server, to utilize together with the Ramsy app and aSTEP system. The Ramsy server could store the extra user data and connect the user to the aSTEP user ID, so that each Ramsy solution user has a unique aSTEP user, with contact information stored on the Ramsy server.

The Ramsy server has two main purposes. Firstly, to store additional information about the system users, and secondly to store the score for previously assessed scores between given stable routes. The different purposes and responsibilities of the parts depicted in Figure 4.2 will be elaborated in the following text.

The Ramsy application continues to be responsible for the interaction with the user as well as collecting the location data needed to track routes and recommend ride partners. The app communicates with both the Ramsy server and the aSTEP server through their respective API's.

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

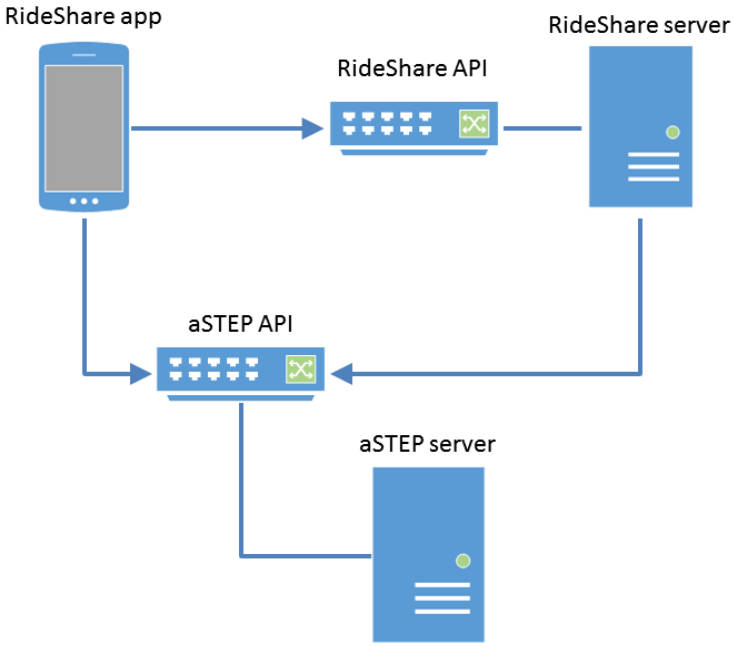


Figure 4.2: System design, including all major parts of the solution.

Beyond storing scores for route matches and additional user info the Ramsy server will also be responsibly 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 Ramsy 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 Ramsy server.

The final system design developed in the second iteration, described in this section, can be seen in Figure 4.2.

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

4.2.2 Algorithm

The system uses a custom developed algorithm to determine whether two routes are a good fit. This algorithm is loosely based on a simplified version of **ghoseiri2011real**, 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 Ramsy system are the time and distance criteria. A position is spatial-temporal, whereas a location is a spatial point and a time is a temporal point. A route is a chronological list of positions. A disadvantages of the algorithm presented here is that the distance is calculated as a euclidean distance rather than a map distance. This would be a obvious place for future improvements to the algorithm.

To illustrate how the algorithm should work for different routes an example is introduced. The example can be seen in Figure 4.3. The example introduces four

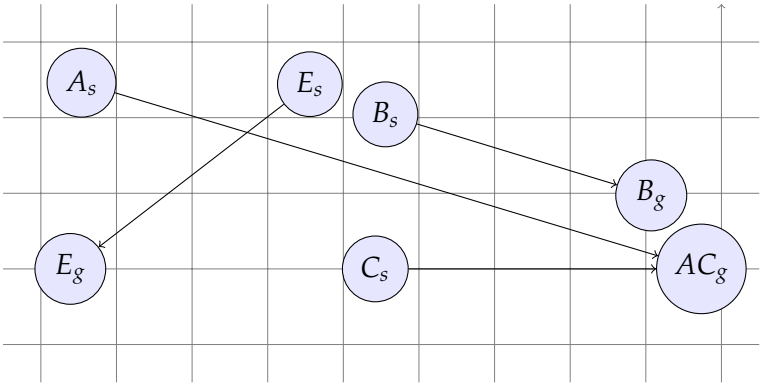


Figure 4.3: Alice, Bob, Carol and Eve's stable routes. Each grid represent a 5×5 km area.

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 4.2.

For this example, the goal will be to find out if any route matches with Alice's route $\{A_s - A_g\}$ The outcome should be 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 way off. 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 Analyser Algorithm

In the algorithm, there are two aspects to be evaluated; 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 depending on how closely they match.

If either time or distance is deemed 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.

The final algorithm 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 analyser algorithm will be described, followed by the time analyser algorithm.

Algorithm 1 Time Distance Analyser 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 TimeDistanceAnalyser(r, s, g)
2:     d ← DistanceAnalyser(r, s, g)
3:     t ← TimeAnalyser(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
```

Distance analyser

The distance analyser algorithm takes a route and two end points and assesses the score for the detour when only considering the distance. To do 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:

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

The distance analyser 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 every detour and thus this is an area for future optimization.

The final score for the distance analyser 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

Algorithm 2 Distance Analyser pseudocode

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

1: **function** DISTANCEANALYSER(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: $d_s \leftarrow dist(r_s, s)$ ▷ Pickup detour distance
5: $d_g \leftarrow dist(r_g, g)$ ▷ Set off detour distance
6: **return** $1 - \frac{d_s + d_g}{\beta}$
7: **end function**

it from 1 to make greater values better matches, 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 4.3 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 4.3: The detour distance for Alice compared to the other users as well as the score of the DistanceAnalyser for each route.

Table 4.3 presents the results for the distance analyser. 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 analyser

The time analyser algorithm works similarly to the distance analyser 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, this is 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 have 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 shows the pseudocode for the time analyser algorithm. Line 2 to 4 are reminiscent of line 2 to 5 in Algorithm 2 with the difference that they sums 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,

Algorithm 3 Time Analyser 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** TIMEANALYSER(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**

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 analyser algorithm is applied to the example with the aforementioned data, the result can be seen in Table 4.4.

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

Table 4.4: The detour time and time difference for Alice compared to the other users as well as the score of the TimeAnalyser for each route.

In the results from the example in Table 4.4, 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 4.5.

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 4.5: The result for Alice's match compatibility to the other users calculated by the Time Distance Analyser.

The results in Table 4.5 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 analyser, 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 algorithm, the Time Distance Analyser 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 Analyser 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 Analyser the complexity is the same as in Distance Analyser 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 with that, we can determine that the Time Analyser algorithm is $O(|r|)$.

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

4.2.3 Location

In order to be able to locate a user and to construct routes, it is necessary to collect location data. The location data should only be collected under certain circumstances, such as when the user is driving in a vehicle. Such collection of location data can be done by using already existing services, such as the Google Play Service.

Location data in RideShare is only useful if it is formatted in GPS coordinates, stored in a collection that represents a route from A to B. There are several ways to collect location data. One of them is by developing a component to do so on the android device. Another way is to use already existing services, such as one of the Google Play Services.

To use Google Play Services, a client library much be included in the app, which will communicate via inter-process communication to the Google Play Services which is already existing on every android device. When the application is connected to the Google Play Services, it will automatically receive silent updates regularly, to acquire new features and bug fixes to the used services. This is illustrated in Figure 4.4[**GapiOverview**].

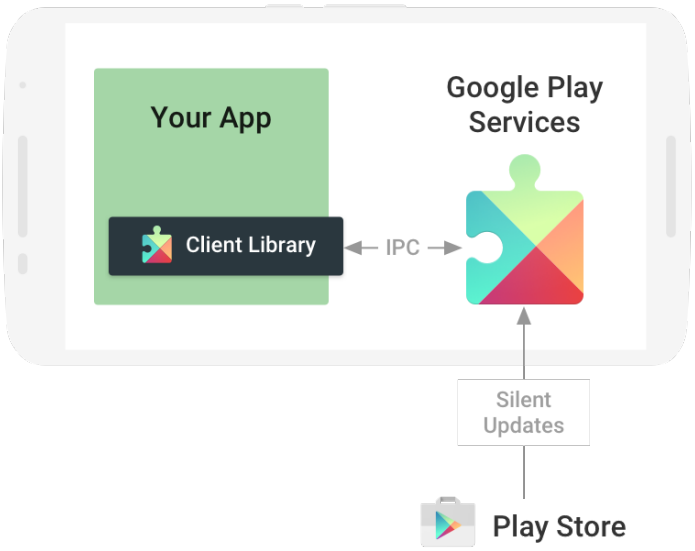


Figure 4.4: Google Play Services communication[**GapiFigure**]

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.

The Google Play Services will allow the application to collect location data, but it is not doing so solely using the GPS in the device. The location service utilizes both network location and GPS to estimate a position as precise as possible [**GapiLocation**]. To ensure the collected location data is relevant, it is needed to store data only 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 uses several sensors on the device to determine what kind of activity the user is currently performing, therein driving, walking, etc. The service will return a probability level from 1 to 100, where 100 is certain that a user is performing the activity. The activity recognition will be used to prevent unnecessary data will be stored on the database. When a user is in a vehicle with a probability level above 75 %, data should be stored and used for computations. placeholder

placeholder

The application must be running in the background while the device is turned on, for it to be able to differ between the users activity at all times and collect data. It is chosen only to collect location data from the location services 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.

4.3 Implementation

4.3.1 JobScheduler

The `JobScheduler` was chosen as the scheduler for the background service in Section 4.1.3. 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 4.1.3 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(10000);
4 int response  = jobScheduler.schedule(builder.build());
```

Listing 4.1: The implementation of jobScheduler.

Listing 4.1 presents the implementation of `jobScheduler` in the `MainActivity` of the app.

In the app, the job is implemented as an extension to the `JobService` class which

provides the required interface for the `JobScheduler` called RideShareService. The class reroutes the call by the `JobScheduler` to a `Handler` class. This `Handler` class, as described by the official documentation [**handler**], allows the service to send a runnable object to the threads message queue.

The implementation makes the location gathering be performed as a background service every 10 seconds. The interval is temporary, and will serve to test the implementation.

4.3.2 Location Gathering

To get the user location in the background service, the Google Play Services is utilized. The Google Play Services provides necessary location data, due to the integration on the platform.

describe BGS before this

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:

```
1 <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
2 <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.

ensure this is documented

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 4.2, 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.

```
1 @Override
2 public void onConnected(Bundle bundle) {
3   currentGLocation = getCurrentGLocation();
4
```

```
5   // convert location to astep if available
6   if (currentGLocation != null) {
7     ASTEPLocation currentAstepLocation = convertToAstepLocation(currentGLocation)↩
          ;
8   }
9 }
```

Listing 4.2: 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 4.3. 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 4.3: getCurrentLocation

Convert to aSTEP location

The aSTEP location format contains the essential location information: latitude, longitude, accuracy, and a timestamp. The 4.4 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 4.4: ASTEPLocation()

4.4 Test

~~The test performed in this sprint was regarding the location gathering~~ The purpose of the test chapter 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.

4.4.1 Background service

To assess the functionality of the background service, it was necessary to receive some kind of information, confirming that the service was actually running. The background service was tested by making a simple IDE console output, that got printed 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.

4.4.2 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 once in a significant number of 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 services.

The implementation of the ~~jobScheduler~~ `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 ~~locationing~~ location feature as little as possible, because GPS location collection is battery consuming[**gpsbattery**].

Chapter 5 Sprint 3

Metatext for the third iteration

5.1 Analysis

This analysis section contains analysis of the previous sprint with the intention of improving the solution, and considerations of further development of Ramsy. Additionally, because the aSTEP system is in development and the Ramsy solution is dependent on its status, the 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.

5.1.1 Location gathering as a background service

A new background service is examined as the background service implemented and tested in Section 4.4 was not performing correctly. Furthermore, the constant interval 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 e.g. sleeping.

Location Gathering
Since Section 4.3.2 we have discovered a new method for gathering locations. It can be done by requesting location updates from the Google Play Services [**receivingLocationUpdates**]. 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, and rely on the Google implementation which is already optimized within Android.

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

The activity detection could improve two factors: Energy consumption and location filtering. The location filtering should be done by only recording locations when the user is driving. The energy consumption is lowered as the GPS unit is only utilized when the locations are necessary to collect.

5.1.2 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 to user management and location based services. The LBS are split into indoor and outdoor locations calls, but the outdoor services are the only relevant calls for the Ramsy system.

An overview of the relevant API calls, as of sprint 3, for the Ramsy system can be seen in Table 5.1, while the complete set of sprint 3 outdoor location services and user management services can be seen in the appendix.

fix appendix

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 5.1: Relevant aSTEP API functions.

For the location based services, the aSTEP system now offers different get requests regarding a users location history, nearby users and users based on groups or friends. 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 Ramsy 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.

5.1.3 Requirements for the third sprint

~~Sprint four is ...~~The third sprint is deemed to satisfy the requirements not met in the previous sprints. The app is currently a working template and can instantiate a background service, but is not completed. The background service redesign and implementation is the main focus of this sprint, together with the route matching algorithm testing.

~~some requirement~~Location gathering as a background service
~~placeholder~~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 users 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 delivered to the outdoor groupresponsible for the implementation in aSTEP. The algorithm is supposed to be implemented within this sprint, and the implementation should be tested for ability to produce results and correctness. The testing must be done with some test data, along with result expectations.

clarify?

5.2 Design

5.2.1 ~~System design~~

~~In the second sprintthe system were designed to consists of three parts: The app, the server, and the system. As thesystem have evolved small redesigns to the system design is necessary~~This design section contains documentation of the redesign of the background location gathering service and the minor system changes, as well as the design of the user data access management.

5.2.1 System design

This section will describe how the communication between the Ramsy system and the aSTEP system is ~~handled. And~~ performed, and how features from the aSTEP API ~~is~~ are used to achieve the ~~desired outcomes.~~ required functionality. The Ramsy system continues the triple part design, but has to adjust to the aSTEP progression and changes.

The aSTEP API ~~is, as described in Section 4.1.5,~~ remains a REST API~~. This means~~, thus meaning that all API calls are done through HTTP requests[1]~~.~~

~~We~~ . The aSTEP system provides an assortment of API functions, but the Ramsy system will only be using a subset of the features~~presented to us, those being; Locations, Routes , Users, and some way to~~ : Routes and Users, excluding categories such as indoor LBS. The API functions are utilized to send recorded routes to aSTEP, and to create and edit users. Because there are restrictions regarding user information sharing in aSTEP, it is also necessary to allow Ramsy to access the data from the aSTEP server for its users. ~~In the API there~~ A overview of the API functionality as of sprint 3 can be found in Appendix A.

[1] ~~Overview of all the HTTP-request available in can be found at~~

5.2.2 User management

User management must be performed for the Ramsy 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 ~~doing this; by creating~~ resolving user data sharing using the aSTEP API. The first solution is to create a group for all the ~~app usersor by having an edge from a seperate~~ Ramsy users. The second is to create one administrating Ramsy ~~-user to every~~ user that has an access edge to every Ramsy user in aSTEP ~~-user that uses the app~~system. The following paragraphs will elaborate the alternatives.

~~Groups~~ Groups

Groups in the aSTEP ~~program exists~~ system exist to allow users to gain access to ~~eachothers~~ each others data. This could be ~~used~~ utilized by making a ~~user for the~~ Ramsy ~~system where all users who uses the app is includedin. The problem with this~~ group in which all Ramsy users are included. An issue with this solution is that all users have ~~now granted~~ granted all other users access to their ~~local datato anyone how happens to use the same app, regardless of the context. This means that a malicios programmer could~~ own location data. The solution allows a malicious programmer to create a program that ~~plots~~ either copies or edits the locations of all users of the Ramsy app. ~~They~~ The intruder would only need a ~~new user registered as using the~~ registered Ramsy ~~app and would there for~~ user and would thereby be included in the group.

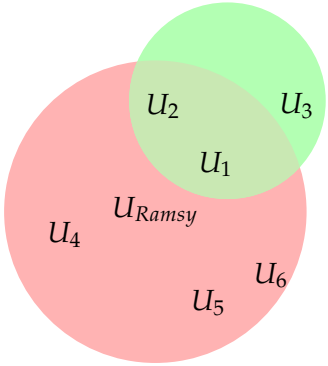


Figure 5.1: How the Ramsy-system would use ~~gropus~~groups. In the red group (the red circle) all users in that group can see all data from other users in the same group.

~~Edges~~ ~~works is~~ Edges

Edges work in a similar way to groups~~but instead of granting all users access~~

~~to every other users data, the edges only allow~~ , but only allowing access from one user to another~~. Using~~ , corresponding to a group of two users. This can be accomplished by establishing communication using the same method of ~~establishing communication~~ as with the groups ~~we create~~ and creating a Ramsy ~~-user and creates a edge from that~~ user and instantiate an edge from the Ramsy user to every user in the Ramsy system, illustrated in Figure 5.2. This way~~it is~~ , only when the Ramsy system uses its own user ~~it is allowed to access data from the users~~the remaining user data can be accessed.

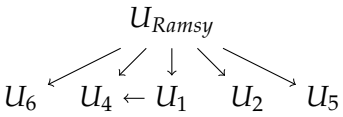


Figure 5.2: How the Ramsy-system would work with edges.

~~This means that edges can~~ The edges method allows edges to provide the same ~~benifits without opening the same security flaws. Therefor the edges is determined to be the superior system.~~

~~A overview of every API calls, as of sprint 3, required by the -system can be seen on Table 5.2~~benefits as groups while disabling the security flaw mentioned above. Therefore, the edges method is selected as the user management design pattern.

~~Path Method Description~~**Creating users**
~~/locations/outdoor/{username}/routes POST Send a route to the server/locations/outdoor/{username}/r~~
~~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~~

In sprint 2 it was also decided to ~~created a user~~ create users through the Ramsy server, this ~~have~~ has been changed to simplify the implementation ~~.~~ and reduce design complexity.

When a user registers in the ~~app~~ Ramsy app, the app will issue an API call ~~will be issued~~ directly to the aSTEP API~~directly from the app where before this was done~~ , instead of performing the operation through the Ramsy server. The username and ~~the additional information~~additional information, such as phone number and full name~~are still transfered~~ , will still be transferred to the Ramsy server ~~for storing~~to enable contact information sharing with other Ramsy users.

5.2.3 Location Gathering

The location gathering should ~~as previously discussed be performed seamlessly~~ be performed unnoticeable for the user and in the background without any ~~user interaction necessary~~ necessary user interactions. This requires certain design ~~requirements which~~ features that will be presented in the following ~~text. The location gathering should be a background service which will run i the background once the user have logged into the app.~~ sections.

Overall Design

The informal tests performed in sprint 2, Section 4.4, showed that polling locations with fixed intervals controlled by a timer proved difficult to get working reliably, as the Google API client never seemed to connect before the task had 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 interval.

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 done through the Google Play Service location API which Google recommends above the native Android location framework [**apploc**]. 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 in 5 minutes the route is considered ended, and the route building should be finalized and send to the server.

Polling frequency

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

First, we consider the initial task of the location gathering service: The activity recognition. The activity recognition is used to decide the location requests, as it has lower consumption than the GPS unit ~~.~~ [**fuckGPS**] since the GPS system prevents a phone form going into it's normal sleep cycles. Many newer smartphones [**coCPU**] 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 ~~a couple of~~ every 5 seconds, which later can be optimized to use less power or be more accurate if needed.

The location request which will be called when a user is detected driving, and will in most cases need to activate a device's GPS and thus consume a lot of power. The first iteration of location gathering it will be a simple solution which will request with static intervals, whereas dynamic intervals based on factors such as speed possibly could be evaluated for battery optimization later on. The location updates interval will be set to ~~range of every 1-2 minutes.~~ most frequently at 1 minute, but targeted at every 2 minutes. Although the interval values are within the time scope, the values are susceptible to changes depending on test results.

5.2.4 ~~App-to-aSTEP communication~~

5.3 Implementation

The implementation section contains descriptions of selected parts of the implementation performed in sprint 3.

The main focus is the location gathering service, which will record locations continuously while the user is driving, but not otherwise.

5.3.1 Location Gathering Background service

This subsection will describe the contents of the implementation of the background location gathering. The new implementation is ~~based~~ founded on the poor results of the location gathering method implemented in sprint 2, Section 4.3.2, that was not working as intended. The code examples in this section will not include potential logging and debugging lines as ~~to save space~~ for better readability and they do not influence the functionality of the code.

Implementation plan
According to the design, Section 5.2.3, the locations should first be requested when the user is driving. Therefore, the implementation starts by instantiating an activity detector. The activity detector can thereafter check the received activity, and begin location gathering if the activity is above a confidence threshold for driving activity. 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 5.1.1. 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 5.1. The activity recognition can thereafter check that the detected activity is a driving activity, 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 5.1: Initialization of activity recognition.

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

The broadcast receiver is then applied by registering to the Ramsy app's `BROADCAST_ACTION`, so that the `activityDetectionBroadcastReceiver` only handles broadcasts sent by Ramsy 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 extra data in the broadcast is stored in the `ArrayList<DetectedActivity> detectedActivities`. The `detectedActivities` is iterated over, shown in Listing 5.2 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 5.2: Iteration over received list of activity recognition.

Location Updates Request

The `startLocationUpdates()`, shown in Listing 5.3, 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 ~~quality~~ accuracy priority, respectively set to 60 seconds, ~~120~~ 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 5.3: Start location updates functions.

~~To send the routes gathered through the location service, the communication with~~

5.3.2 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 ~~must be established.~~ 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 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, but this will be described in the following test section.

5.3.3 App-to-aSTEP communication

This section contains an overview of the implementation of the API calls utilized to communicate with aSTEP.

In the previous design phases 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 5.2.

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 5.2: Caption

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

Using an object orientated structure, each API call is implemented to overwrite a `helperClass` which will handle the inputs, such as username, password, token, and routes. The inputs are then transformed to a part of an URL which would fit each of the relevant API calls. The `helperClass` 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 will be logged on the aSTEP server. This would be a major security issue. Instead password and tokens are parsed as part of the body, which is used by REST. This will hide the content from the URL and instead be transfered in an `OutputStream` encoded in raw bytes.

Two examples of API calls can be seen in List 5.3.3. The post route example is simplified to reduce size and make it better explainable.

fix references and captions

1. POST new user
   (a) URL: http://astep.cs.aau.dk:80/api/users?username=USERNAME
   (b) CURL: curl -X POST –header 'Content-Type: application/json' –header

'Accept: application/json' -d 'PASSWORD' 'http://astep.cs.aau.dk:80/api/users?username=US

2. POST route to aSTEP
   (a) URL: http://astep.cs.aau.dk:80/api/locations/outdoor/outdoor/routes?authorization=TOKEN &distance_weight&time_weight&largest_acceptable_detour _length=146&acceptablehttptime_difference=32
   (b) CURL: curl -X POST –header 'Content-Type: application/json' –header 'Accept: application/json' 'URL from above'

As seen in Create user in List 5.3.3, the username is parsed in the URL while the password is included in a header. On the other hand, the post route is only parsed in the URL as there is no information that needs to be hidden. Every decision made regarding parsing data from the apps to the aSTEP server is decided by the individual aSTEP API groups.

5.4 Test

A test of the location gathering background service was performed to examine the functionality, performance and reliability.

5.4.1 Location gathering as background service

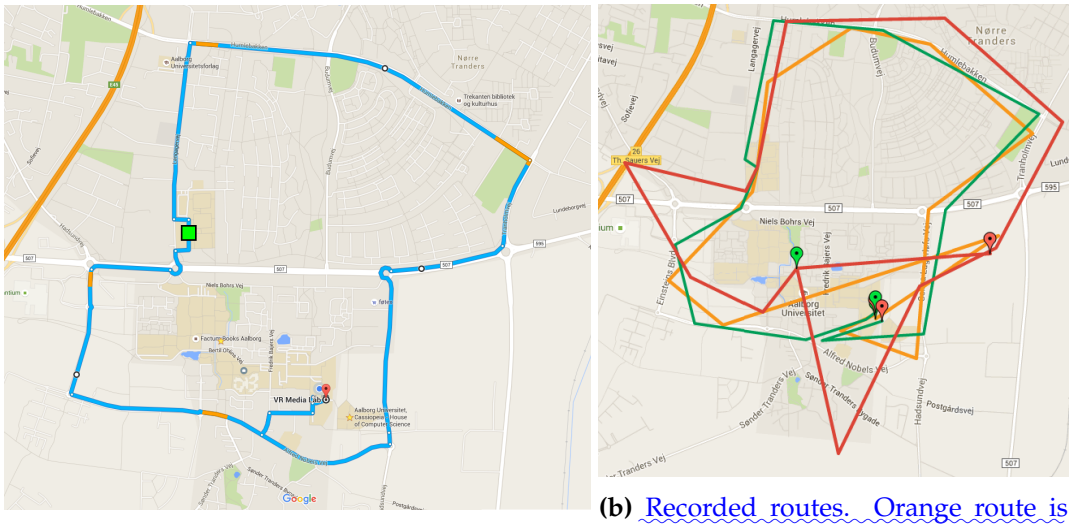
The Ramsy app was built and installed on three units to test on a variation of devices. The devices were a Sony Xperia Z2, a Samsung Galaxy S6 and a Huawei Nexus 6P, all with Android Marshmallow 6.0.1 as operating system. The app was opened and the necessary permissions were granted, so that the app could function properly. The app was configured with the values seen in Table 5.3.

5.4.2 ~~App-to-aSTEP communication~~

The route driven to test the location gathering can be seen in Figure 5.3a. 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 5.3a. The stop had a duration of 2 minutes, thus not passing the threshold value of 3 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.

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



(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 5.3: 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 5.3b. 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.

5.4.2 Route Match Algorithm

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

The implementation is tested with a sequence of routes inserted into a previous empty route database. By adding routes sequentially, we can anticipate the matching outcome for each addition.

Test Data
A number of cases should be tested, and we generate route data for each. Before the route data is input users U1, U2, U3 and U4 are created to the aSTEP system.

First, the stable route generation must be confirmed functioning. A route is created for U1 and made two copies that were slightly edited 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 this user.

Second, the algorithm should not be matching routes that are not relevant in location or time. U2 gets two routes added not matching in location, as seen in Figure 5.4. The blue U2 route is during the day time and the black during night time. 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 the 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 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 5.5 from the outdoor group.

Test Results
The test results seen in Figure 5.5 reveal that the algorithm is performing correctly. The results correspond to the assumptions made during the data generation. All

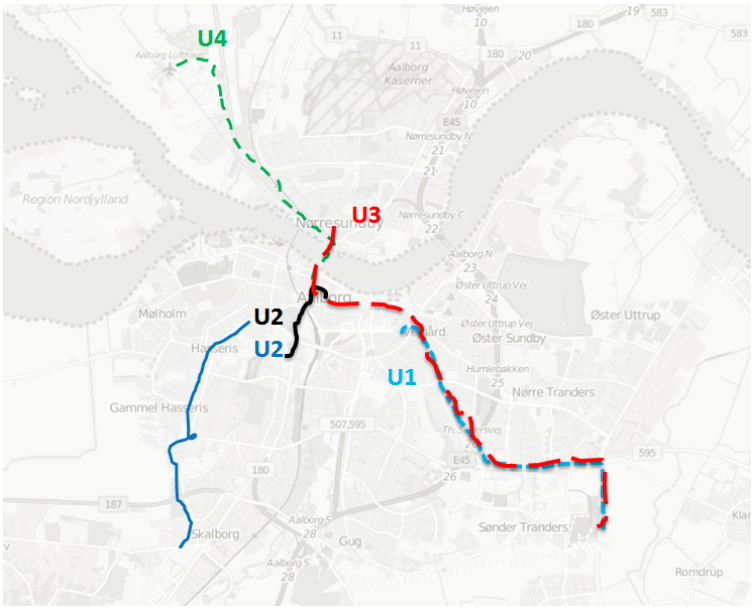


Figure 5.4: Generated routes to be tested by the algorithm.

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.

	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 5.5: Results generated by the implemented algorithm.

Test Conclusion

The algorithm test results are considered sufficient for the current application. However testing, including user interviews, over large amounts of data for actual 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 the weight of those in the final score by changing parameters in the

Ramsy server's call to aSTEP system.

Chapter 6 Sprint 4

Sprint four is the fourth and last iteration of the project.

6.1 Analysis

6.1.1 Missing features for completing the solution

6.1.2 Algorithm results analysis

6.1.3 Requirements for the fourth sprint

Route Matching Algorithm
placeholder

6.2 Design

6.2.1 Algorithm redesign

6.2.2 Algorithm test design

6.2.3 RideShare Server Design

6.3 Implementation

6.3.1 RideShare Server

6.4 Test

6.4.1 Background location gathering

6.4.2 RideShare Server

6.4.3 App-RideShare Server Communication

6.4.4 App-aSTEP Communication

6.4.5 RideShare Server-aSTEP Communication

6.4.6 App-RideShare Server Communication

Chapter 7 Summary

7.1 Reflection

Some to come here

7.2 Conclusion

I made dis.

7.3 Future Work

Fix it.

Appendix A Sprint 3 API functionality

~~Here is the first appendix~~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 A.1: Currently planned aSTEP API functions.