Abstract

BusTUC is a natural language bus route expert system developed at IDI NTNU and provides the inhabitants of Trondheim with route suggestions every day. The Android application described in this paper can be considered an extension of BusTUC with real-time data and GPS. By having BusTUC suggest multiple travel routes given the users location and adjacent bus stops, the application is able to narrow those travel routes down to the 'best' ones.

Contents

1. Introduction

This report describes the motivation, method and technologies needed to create an application which intelligently calculates the best possible travel route for bus users in Trondheim. By using Global Positioning System (GPS) coordinates, real-time data and a natural language bus route expert system the application is able to give the user information describing the fastest way to get from point a to point b.

The primary motivation behind this research is to combine numbers from many different sources to produce the most reliable, accurate and intelligent route suggestion system up to date for Android. The problem with information about bus routes is not that it is hard to find, but that it is spread over multiple system. Consider yourself located in an area which you are not familiar with. You do not know where the closest bus stop is, what the route times are or even which bus you should take. A simple query using this application would solve this predicament.

This report will also contain a description of the underlying technologies located in smartphones and how to utilize these technologies in order to provide a system with all the required functionality.

The components involved in this system are primarily the smartphone, BusTUC and AtB's real-time system.

In a world where users require instant information with minimal effort, high expectations are set for information value and correctness, usability and stability. These requirements are especially important because people will trust the information given and plan their route accordingly.

The core requirements in this project are:

1. Find the user's position by utilizing GPS.

2. Use this position to find the relevant bus stops surrounding the user.

3. Use the relevant bus stops and the destination provided by the user to find different route suggestions.

4. Acquire the actual arrival times with the use of real-time systems.

5. Rank the suggestions and present this information in natural language and with a graphical representation of the suggested routes.

This application is also created as generic as possible. It can be used in any another city as long as it has coordinates for the bus stops and BusTUC to communicate with. It can be viewed as an addition to BusTUC.

Below is an abstract diagram describing the communication flow.

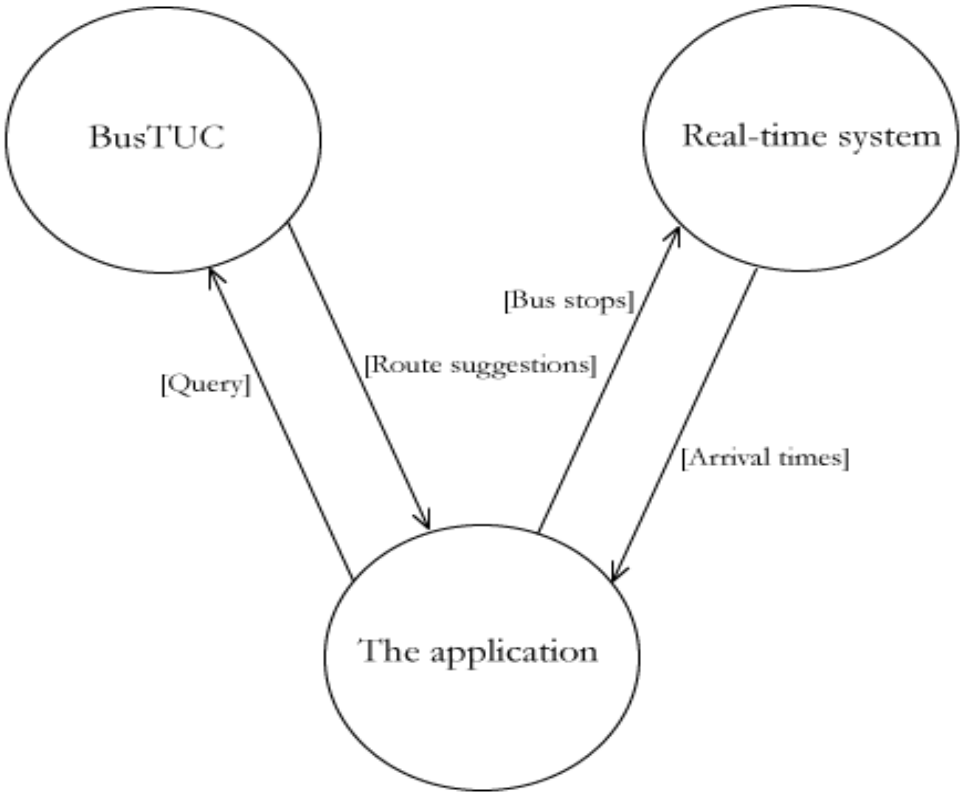

Figure 1.1 Communication flow in the application

As depicted in Figure 1.1, the application will send a query containing the destination and the relevant bus stops to BusTUC. BusTUC will generate multiple bus route suggestions which it will return to the application. The application will examine these suggestions (and probably remove some of them) and use the bus stops where the travel starts to acquire arrival times from the real-time system.

After receiving the actual arrival times, the suggestions will be ranked and presented to the user.

The application has been under development since autumn 2010 as a project given by Tore Amble, associate professor at IDI NTNU. While the primary goal was to develop the actual application, a lot of work went in to discussing different methods, solutions and results. This report will hopefully provide some insight in this discussion as well as describing the final solution (final in this context does not mean commercially ready). The report will also point out the limitations and disadvantages of the application as well as possible solutions.

This report is originally meant for professors and students at NTNU with a technical background.

2. Similar systems

There are many systems which provide useful information when planning a bus travel. One of them, the one that resembles this application the most (at least in theory) is TransitGenie, an app for iPod. It is developed by the BITS Laboratory at University of Illinois and handles public transport in the city of Chicago[1].

TransitGenie works by either having the user enter their current position by hand or having the GPS in the phone figure it out. The user then enters his desired destination and four different travel suggestions are displayed. The different suggestions include directions and an estimated time of arrival.

Figure 2.1 TransitGenie route suggestions

Figure 2.1 shows TransitGenie presenting four different travel suggestions to the user. They are ranked by earliest arrival time, shown in the column to the left. The green man represents walking distance for the user, while the black icons represent buses and the blue icons represent a tram. The number under the buses is the route number.

In Norway the closest thing would be the application developed by Trafikanten[2] for users of public transport in Oslo. Here you can choose between real-time data or route suggestions ranked after arrival. Current position and destination is chosen manually by the user, but position is suggested by the application based on the user's current position.

[1] BITS Laboratory, University of Illinois – http://transitgenie.com/
[2] Trafikanten, Oslo - http://www.mobilen.no/artikler/ukens_app_trafikanten/88217

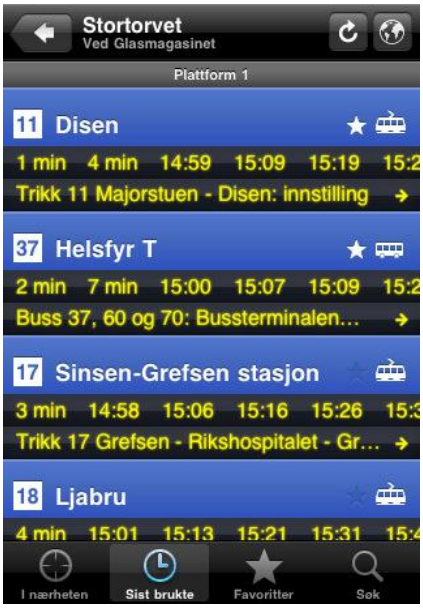

Figure 2.2 Trafikanten real-time window for a single station

In figure 2.2, the Trafikanten application is showing the arrival times of the different types of transportation at 'Stortorvet' station. For instance, the number 11 tram (with Disen as the final destination) will arrive at 'Stortorvet' in 1 minute, 4 minutes, 14:59 etc.

The other part of the application is the route suggestion. This uses logic already used by www.trafikanten.no, the web portal for route suggestions. One important distinction between the web portal and Android application is that the latter is able to utilize GPS.

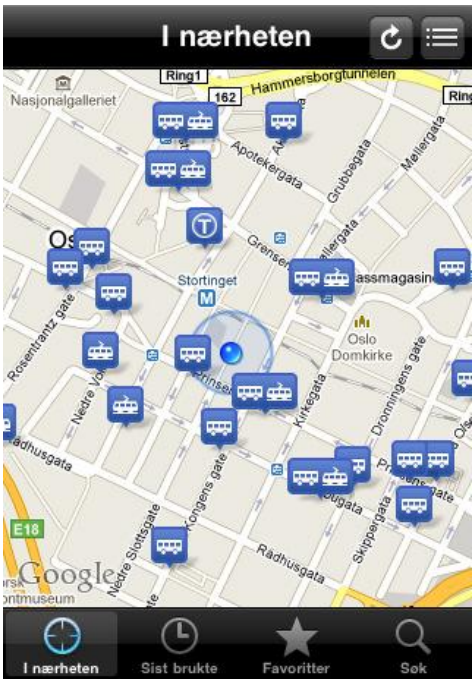

Figure 2.3 Trafikanten application suggesting current position.

The user is presented a map with his/her current position and is able to pick one of the adjacent stops as their starting point.

The user then types in their destination and a route suggestion is presented to the user given the time of the query.

Figure 2.4 Suggestions derived from www.trafikanten.no

Figure 2.4 does not show how the information is provided on the Trafikanten Android application, but on www.trafikanten.no. The same information will be revealed in both, but shown a bit differently.

In the query which produced the route suggestion given in figure 2.4 the starting point was 'Stortinget' and the destination was 'Bøler'. The query was not asked in natural language (e.g 'How do I get from Stortinget to Bøler?', but rather filled in pre-set boxes representing starting point and destination.

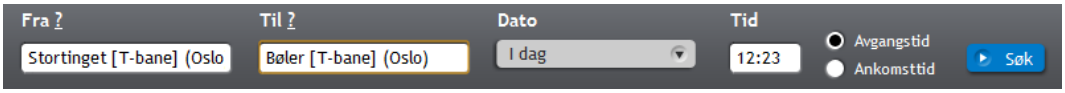

Figure 2.5 Trafikanten.no search bar

The boxes are labeled fra (from) and til (to) as shown in figure 2.5. The user is also able to set the date and time of either departure or arrival (avgang or ankomst in Norwegian).

Some differences between the Trafikanten application and the application described in this paper:

1. The Trafikanten application derives route suggestions assuming only one starting point, while this application considers many.

2. The Trafikanten application is not able to handle natural language queries.

3. The Trafikanten application does not include real-time data with its route suggestions.

3. Underlying Technologies

This chapter will give a quick overview of the underlying technologies which provides the functionality used in the creation of this application. A more general and detailed description off the technologies can be found later the appendix provided with this paper.

3.1 Hardware and OS

The smartphone used for the development of this application is a HTC Wildfire. It runs on Android OS 2.1 developed by Google. There are over 200,000 applications available for Android smart phones.

The hardware provides two key services needed for this application.

1. Deriving user location with GPS.

2. Internet access.

3.2 Deriving location

If you look back at figure 1.1, you'll see that the first step in the communication flow is the query to BusTUC. This query contains a list of relevant bus stops or what you might call potential starting positions for the bus travel.

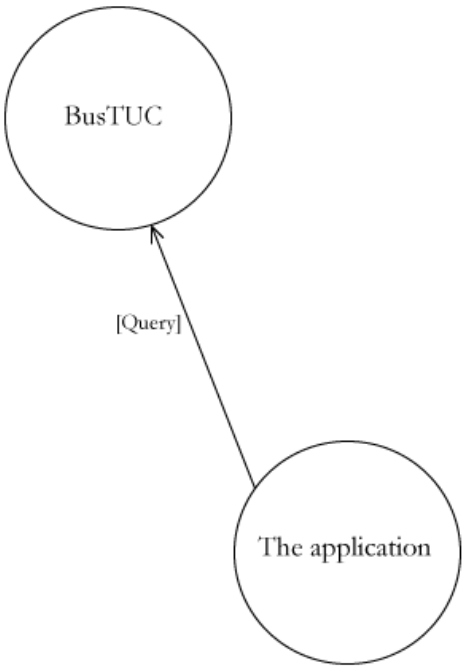

Figure 3.1 Query sent to BusTUC

To be able to find out which bus stops the user might want to travel from, you'll need to find out where the user is located. This is done by utilizing the GPS receiver in the smartphone.

The location is found by the GPS receiver by calculating the distance between multiple satellites and comparing them. The accuracy of the position will increase with the amount of satellites it gets a signal from. 3 satellites are enough to calculate a 2D position (longitude and latitude) which is good enough for this purpose. More information about GPS and location calculation can be found in the appendix.

3.3 BusTUC

Let's return once again to the figure 1.1 and look at the technology which provides the different route suggestions.

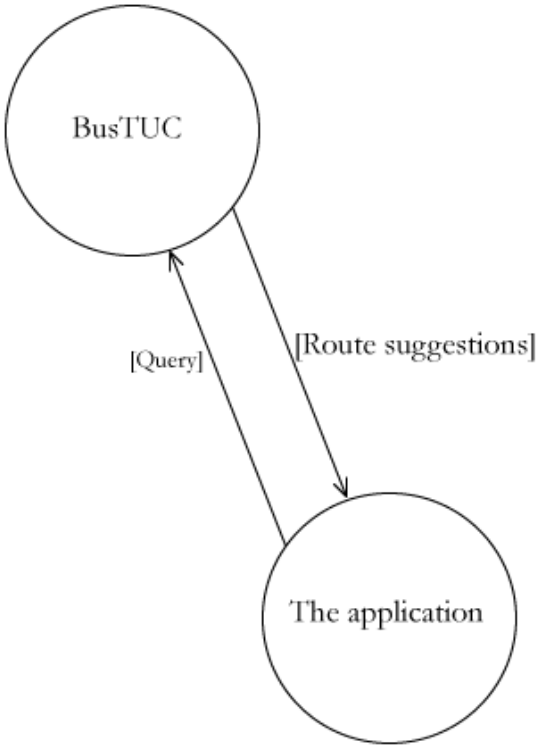

Figure 3.2 BusTUC returning route suggestions

The technology which provides the route suggestions is a bilingual natural language bus route expert system developed at the Department of Computer & Information Science, NTNU[3]. It get its name, TUC (The Understanding Computer), from a general natural language system developed by Tore Amble, a professor at NTNU. It was implemented first for Team Trafikk and later for AtB, the companies responsible for the buses in Trondheim and is considered to have reached the intelligence of a savant. Savant is defined as a term used for a person with certain social disabilities, but has some extraordinary skills when it comes to memory or speed calculations. Their natural language understanding is excellent within their domains of expertise, but lacking when it comes to other conversation topics. BusTUC is indeed very robust when it comes to queries about Trondheim's bus routes. The following examples are in Norwegian.

[3] BusTUC - http://www.idi.ntnu.no/~tagore/rapporter/BusTUC.pdf

'Når går bussen fra Solsiden til Dragvoll? / When does the bus go from Solsiden to Dragvoll?'

This will return a travel route together with departure or arrival times. It will assume you want to travel as fast as possible unless you specify arrival or departure times.

'Når går første buss etter 15.00 fra Solsiden til Dragvoll? / When does the first bus after 15.00 go from Solsiden to Dragvoll?'

This will return the first available travel router after 15.00.

The BusTUC system is separated into 3 different components:

- A parser system which consists of a parser, a grammar, a lexical processor and a dictionary
- A knowledge base (KB), divided into an application KB and a semantic KB
- A query processor, containing a routing logic system, and a route database.

The dictionary contains about 4590 words, 1204 bus stop names, 80 buses and 9970 name variants only for the Norwegian part of the system. The different name variants are important because people do not necessarily call certain bus stops by their 'proper' name. There are also over 5000 grammar rules, a semantic net with 6500 word meaning entries and 1500 more rules which help translate the output from the parser to a route database query language. The semantic knowledge base contains 960 nouns, 1100 verbs and 450 adjectives. All in all there are about 130500 lines of Prolog code. The parser uses a generalization of Definite Clause Grammars, called Consensical Grammar (CONtext SENSItive CompositionAL Grammar). Compositional grammar means that the semantics of the subphrases creates the semantics of a phrase.

The semantic knowledge base creates the foundation and the logic is generally generated by it. When there is a need for changes, these will be made in the fact database and in the semantic knowledge base, while the general grammar and dictionary remains untouched (This is in theory. In practice the dictionary has been extended to have a bigger vocabulary). The semantic knowledge base contains a restricted list of legal combinations of verb, nouns, adjectives and prepositions.

The query processor translates natural language into a form called TQL (Temporal Query Language) which is a first order event calculus expression. The TQL expressions consist of predicates, functions, constants and variables. The TQL is then translated to BusLog, a route database query language.

3.5 Real-time data

The real-time data we want to retrieve from the system is of course the actual arrival times of the buses at a given bus stop. The application uses the same real-time system that is used by AtB. It is called Flash and is developed by SWARCO[4]. Flash is also implemented in Barcelona and Turin as well as other major cities and handles large amount of traffic. The vehicles in question are monitored with the help of GPS and distance measuring.

[4] ATB - http://www.AtB.no/sanntid/category210.html

Let's once again return to the communication flow figure (figure 1.1).

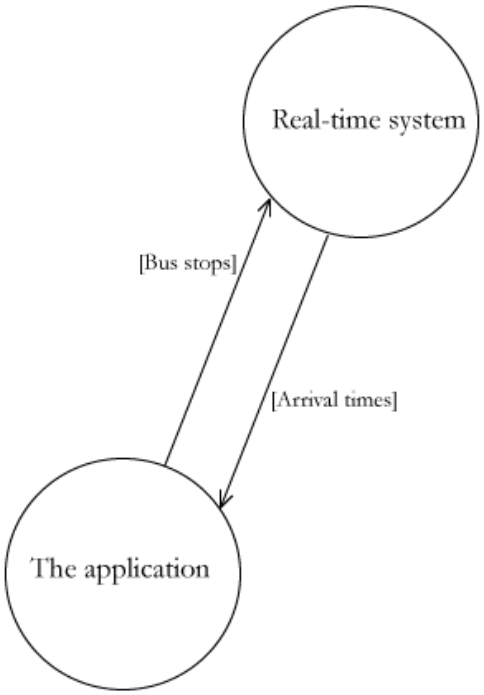

Figure 3.3 Requesting real-time arrival times.

The system returns the arrival times after receiving a SOAP (Simple Object Access Protocol) request from the application specifying the bus stop. The number of calls to the real-time system depends on the amount of suggestions received from BusTUC.

4. Methods

The earlier chapters has hopefully given a overview of the motivation, requirements, data flow and underlying technologies required for meeting the goals set for this project. This chapter will include a more detailed description on how the application displays information, retrieves both the bus stop and user location, acquires route suggestions and updates the planed arrival times with the actual arrival times.

4.1 GPS

4.1.1 Geographic coordinate system

The application uses Google Maps[5] as a map which again uses the World Geodetic System (WGS) as a geographical coordinate system. WGS is a standard within cartography, geodesy and navigation. It comprises:

- A standard coordinate frame for the Earth

- A standard spheriodal reference surface (the datum or reference ellipsoid) for raw altitude data

- A gravitational equipotential surface (the geoid) that defines the nominal sea level

Google Maps is available for websites and different types of applications, Android included.

Figure 4.1 Google Maps is used for this application

Google Maps provides a lot of functionality for developers, including a quick conversion from GPS coordinates to points on the map. Figure 4.1 shows how Google Maps is used in this application. The pin represents the user position and the blue bus stop signs represent the position of the bus stops obviously.

[5] Google Maps - http://code.google.com/apis/maps/

There are some inconsistencies regarding the icons when zooming in on the map. If you compare the two pictures below, you can see that the pin on each picture does not point to exactly the same point. This can be solved by refreshing the icon positions, but it has not been prioritized during the development of this application due to the fact that the actual position stays the same (only the visual representation is slightly off).

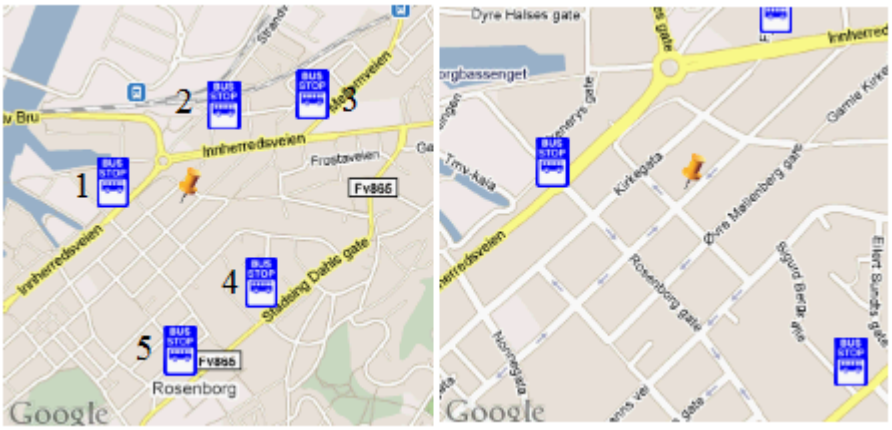

Figure 4.2 Difference in position after zooming out on the map

In figure 4.2, the picture to the right is the original zoom size. The picture on the left is from when the map is zoomed out a bit. The location of the pin is slightly higher on the picture to the left. As mentioned, this lack of accuracy is only in the visual representation and is not actually effecting the actual calculations.

4.1.2 Provider

There are three different 'providers' (as defined by the Android SDK) or ways of utilizing smartphone technology to get user location. Each has their own advantages and limitations.

4.1.2.1 GPS

Derives a location by utilizing the device's GPS chip and is highly accurate in doing so, but a line of sight is necessary for the satellites to provide a fix.

4.1.2.2 NETWORK

This provider uses the cellular network to provide a fast initial fix before utilizing the GPS chip. It is still very accurate and works without provide the satellites a line of sight.

4.1.2.2 PASSIVE

This method derives the location by Cellular ID / Wifi MAC ID look up. This does not require a GPS chip, but is significantly less accurate.

There is one problem with the methods given by the Android SDK to choose the correct provider. It will normally choose 'GPS' as the best possible provider, even though you are currently located inside a building. This will make the application search for a satellite link up indefinitely.

One limitation of this application is that it is only able to run while having a network connection. This is due to the fact that route suggestions require HTTP and SOAP requests.

A discussion around this limitation can be found in the chapter concerning further development.

LocationManager is the class in the SDK which provides access to the system location services. These services allow applications to obtain periodic updates of the device's geographical location. It makes it possible to specify the name of the provider with which to register, a minimum time and distance interval for notifications. This class is combined with a listener.

4.2 Bus Stops

The next information needed before sending a query to BusTUC for route suggestion is the relevant bus stops. The method for finding these bus stops is done by comparing the coordinates of the user location and the coordinates of all the bus stops in Trondheim.

The list of bus stops used in development of this application was provided by AtB. It is included as a XML file in the package, and the application creates an array on startup. One problem with this list is that there is only 1 stop registered per location. So while almost every bus stop has its counterpart across the street (which handles traffic in the opposite direction), only one of these bus stops are registered with coordinates. The method for calculating the air distance between location A (where the user is) and location B (where the bus stop is) in meters is provided by Android SDK.

The Bus Stop list is the only thing (and BusTUC of course) that is specific to Trondheim in this application.

ID	Name	Longitude	Latitude
16000001	Location 1	XX.XXXXXX	XX.XXXXXX
16000002	Location 2	XX.XXXXXX	XX.XXXXXX
.....	……	.....	.....

Figure 4.3 Bus stop list included in the application

The next step now is to find out which of these bus stops that are relevant. There is a need to not just get the fastest possible route from the closest bus stop.

Example

In this example the user is located at 'Gløshaugen NTNU' (shown as a star in figure 4.4) There are here four different bus stops (shown as an arrow in figure 4.4) within a 500 meter radius.

The search algorithm used in this application is able to:

a. Get N closest bus stops.

b. Get all bus stops within a specified radius M.

These values are not mutually exclusive. They can be combined or used individually.

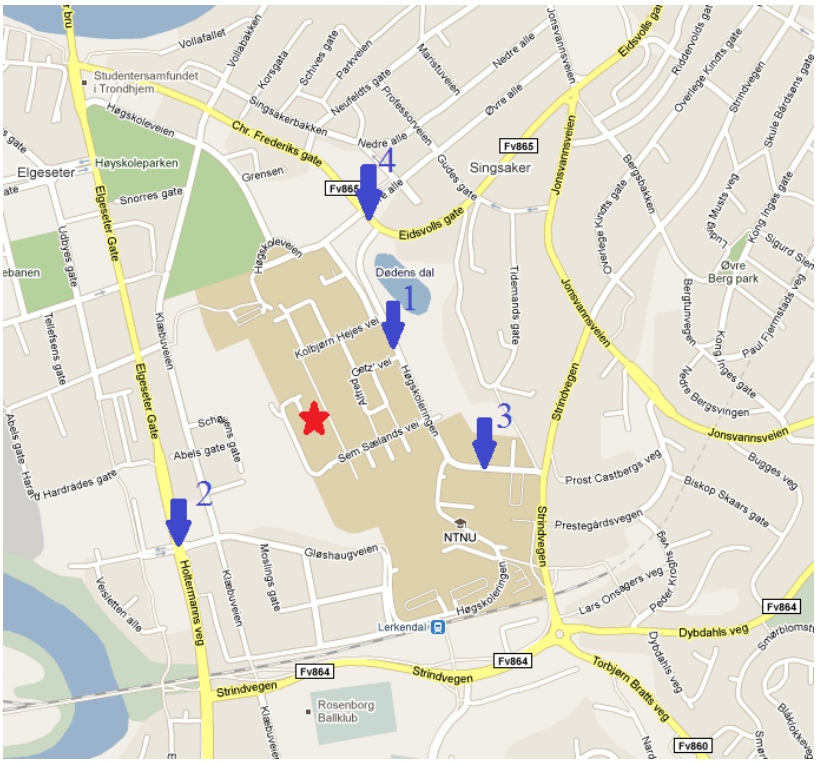

Figure 4.4 Map for example

These four bus stops offer of course different bus routes, so depending on the desired destination some of them will naturally be preferred. If the user wants to travel south, bus stop 2 will probably offer the route with the fastest arrival time, while bus stop 4 will offer the same for users travelling north-east. But some of these routes may only go once every hour, so routes travelling from bus stop 1 or 3 might be preferred from time to time. The motivation for having many possible starting points when calculating travel route is pretty obvious.

The next step after finding relevant bus stops is creating and sending the query to BusTUC.

4.3 HTTP

4.3.1 POST

The method used to communicate with BusTUC is a HTTP (HyperText Transfer Protocol) Browser provided by Java. The common way for people to use BusTUC is through the web interface where they type in a query using the provided form. The form then sends the query to a perl script. This application, on the other hand, creates name-value pairs to simulate the form and sends a request directly to the perl script.

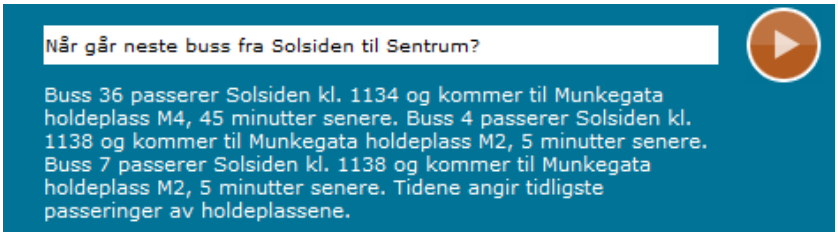

Figure 4.5 A version of BusTUC implemented at www.AtB.no

The perl script has primarily two name-value pairs, language and question. The "lang" name defines what language the input string might be written in, even though it understands both Norwegian and English regardless of the value. This is because there is an internal test in BusTUC which looks up the word in the query and finds which language it belongs to in the bilingual dictionary.

Name	Value
"lang"	"nor"
"quest"	String

Figure 4.6 The name-value pairs of BusTucs input script

The value string is what this application generates. It will take the user generated string (which defines the desired destination) and add the closest bus stops (chapter 4.2).

Below is an example showing the generation of the query string and the returned values from BusTUC.

Example

The user finds himself at Solsiden, Trondheim. The five closest bus stops inside a 500 meter radius sorted by closest first are used to generate a query.

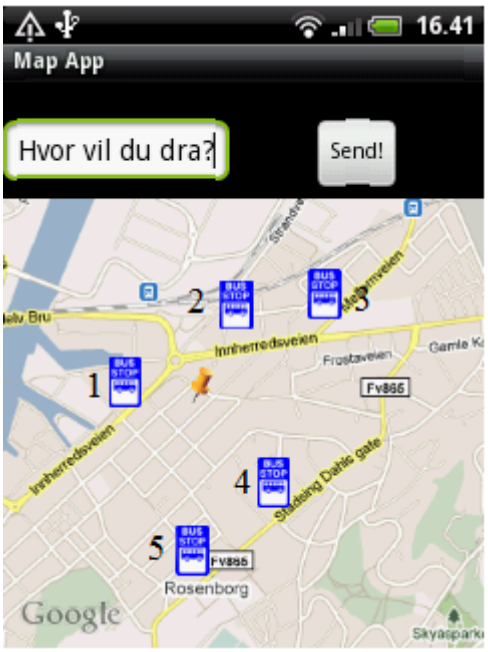

Figure 4.7 The five nearest bus stops within a 500 meter radius.

In figure 4.7 the pin represents the user location. Let's say the user types in 'til sentrum' (to downtown in English). The query sent to BusTUC will be

(Location 1, Location 2, Location 3, Location 4, Location 5) til sentrum

One thing that is worth pointing out is that the application simply sends what the user types together with the relevant bus stops. This means that the user is able to utilize the robustness of BusTUC when it comes to asking questions in natural language. The queries are handled like any other natural language query sent *directly* to BusTUC.

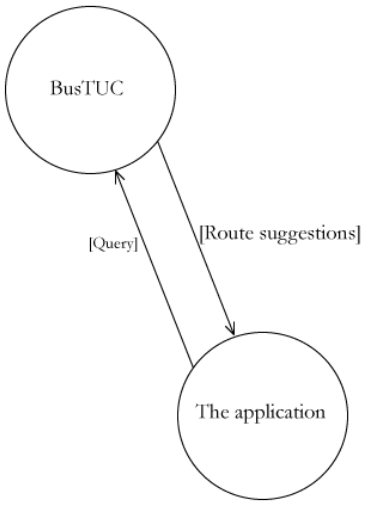

Figure 4.8 Communication flow: BusTuc query.

BusTUC will respond by giving route suggestions formatted in JSON (JavaScript Object Notation), which is a language-independent data format.

Response from BusTUC (JSON)

{"transfer":"false" ,
"timeset":"false" ,
"departures" : [
{"busstopname":"Location1","busstopnumber":16000001,"busnumber":1,"time":1501,"duration":8,"destination":"Sentrumsterminalen"}
{"busstopname":"Location2","busstopnumber":16000002,"busnumber":2,"time":1502,"duration":7,"destination":"Sentrumsterminalen"}
{"busstopname":"Location3","busstopnumber":16000003,"busnumber":9,"time":1502,"duration":6,"destination":"Sentrumsterminalen"}
{"busstopname":"Location4","busstopnumber":16000004,"busnumber":63,"time":1504,"duration":11,"destination":"Sentrumsterminalen"}
{"busstopname":"Location5","busstopnumber":16000005,"busnumber":63,"time":1505,"duration":11,"destination":"Sentrumsterminalen"}
]}

Figure 4.9 Reponse generated by BusTUC

As you can see from figure 4.9, there are two booleans present before the actual route suggestions. Transfer indicates if the user is required to change bus anytime during his/her travel. Timeset indicates if the user has any time requirements for the suggestions. For each route there are six values. Bus stop name, bus stop number, bus number, time (arrival time of bus at the given bus stop), duration (the time spent on the bus) and the destination.

These values help form the route objects which are compared when finding the best possible travel route. More information regarding the inner workings of BusTUC and the route suggestions it returns is found in chapter 4.5 Extending BusTUC.

4.3.2 Real-time

The application has now received several route suggestions from BusTUC and needs to get the real-time values before ranking the suggestions. These real-time values are derived with the use of SOAP objects formatted in XML.

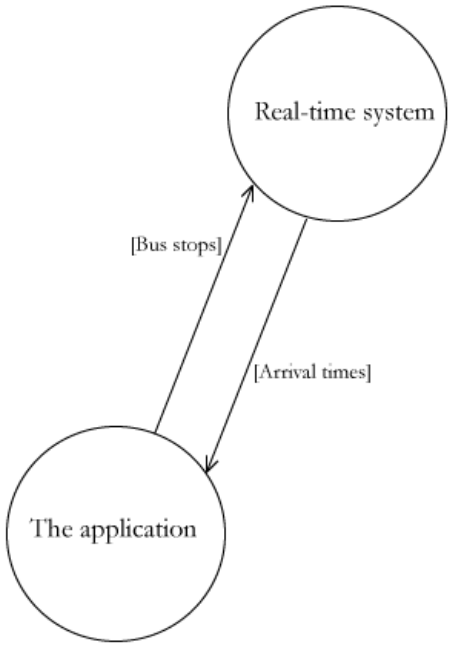

Figure 4.10 Deriving the real-time data.

Some preparation is needed before getting these values because the bus stop numbers returned from BusTUC is not the same numbers as the ones (from now on referred to as 'unique id number') used by the real-time system. The unique id numbers used by the real-time system are even changed from time to time, so this application downloads these numbers when it starts.

The SOAP method provided by the real-time system is called GetBusStopList. It requires only a username and password. The web service will send a SOAP response with a XML node called GetBusStopsListResult.

The information encapsulated inside the GetBusStopsListResult node is a JSON object. The information required by this application is then extracted from the JSON object to create a list containing the unique id numbers required for real-time and the corresponding bus stop numbers.

Unique ID	Bus stop number
111111	16000001
111112	16000002
111113	16000003
……..	……..

Figure 4.11 The list generated the GetBusStopList reponse.

Now the application is able to send specific real-time queries because it knows the relevant bus stop numbers (information received from BusTUC) and the unique id number corresponding to that specific bus stop.

The SOAP method for getting real-time data at a given bus stop is called getUserReal-timeForecast. It requires a username, password and bus stop. The value used for the bus stop is the unique id number.

The response from the server would again be formatted as XML with a JSON object in the getUserReal-timeForecastResult node.

Let's use the values presented in figure 4.11. After receiving a route suggestion from BusTUC which starts at bus stop number 16000002, we can see that the unique id number used by the real-time system for this exact bus stop is 111112. This value is used in the getUserReal- timeForecast method and the information provided in figure 4.12 is returned.

Bus number	Arrival time
3	14:44
4	14:56
2	15:07

Figure 4.12 Real-time data for a specific bus stop.

One thing worth mentioning is that this response includes arrival times of every active bus arriving at this particular bus stop.

Remember from previously in this chapter that one of the suggestions returned from BusTUC was 'bus number 2' stopping at 'Location 2' at '15:02'. The predicted arrival time by BusTUC (15:02) is replaced with the actual arrival time (15:07) before this suggestion is ranked and presented to the user.

Specific implementation notes can be found in chapter 9.1 'Real-time specifics'.

4.5 Extending BusTUC

As mentioned in the chapter above, the application is able to send a wide variety of natural language queries to BusTUC. Some examples of this could be:

Norwegian: *(Location 1, Location 2, Location 3, Location 4, Location 5) til sentrum*
English: *(Location 1, Location 2, Location 3, Location 4, Location 5) to downtown*

Norwegian: *(Location 1, Location 2, Location 3, Location 4, Location 5) til sentrum etter klokken 15.00*
English: *(Location 1, Location 2, Location 3, Location 4, Location 5) to downtown after 15.00 o'clock*

Norwegian: *(Location 1, Location 2, Location 3, Location 4, Location 5) til sentrum før klokken 15.00*
English: *(Location 1, Location 2, Location 3, Location 4, Location 5) to downtown before 15.00 o'clock*

When handling the queries BusTUC will first parse the natural language part. It will derive the desired user destination and set the (optional) time restrictions. Then it will look for direct travel routes between the locations given by the user in the parentheses (this is the list of relevant bus stops) and the destination. Each direct route it finds will be added in the list

of route suggestions. This means that BusTUC might return three route suggestions from one bus stop, while none from another. If the travel route requires a transfer, BusTUC will provide the one that makes the user arrive at the wanted destination earliest.

4.6 Distance between two locations

The method used to calculate the distance between two locations can be found in the Google API. Each location object has a method called distanceTo(Location obj), which returns the air distance between this location and a given location. Distance is defined using the WGS84 ellipsoid. WGS stands for World Geodetic System and is a standard used in cartography and navigation[6]. 84 is the current revision.

4.7 Ranking

The principle used to rank the different route suggestions are based on a couple of assumptions and happens after the route suggestions have updated their times with the times received from the real-time system.

1. The user will not take a bus from a bus stop further away when the same bus is available at a bus stop closer to him.

2. The user would like to arrive at his destination as early as possible (unless routes at specific times are requested).

Let's say that the user would like to travel from a place which is surrounded by 4 different bus stops (Described as Location1,..,Location 4). BusTUC will return something like this:

ID	BusStopName	BusStopNumber	BusNumber	Time	Duration
1	Location 1	16010001	9	1819	6
2	Location 2	16010002	9	1823	5
3	Location 3	16010003	6	1826	8
4	Location 4	16010004	6	1829	11

Figure 4.13 Example of route suggestions from BusTUC

As you see, there are four different route suggestions on two different bus routes. That is quite common in these types of situations, because it is natural that bus stops that exist in the same area will have many of the same buses traveling through them. The exclusion of these types of suggestions is done by implementing the first assumption stated above. In this example the route which starts at location 2 will be removed because location 1 shares the same bus, but is located closer to the user. The route starting at location 4 is removed for the same reason (Location 3 has same bus, but is closer). This allows the application to only run two real-time data requests instead of 4, saving 50% of the original bandwidth and time costs.

[6] WGS - http://en.wikipedia.org/wiki/World_Geodetic_System

Now, after the application calls the real-time system and receives the actual arrival times of the buses at bus stop X, a total time is calculated. The total time consists of the time between now and departure of the bus added by the duration of the travel. WD shown in figure 4.14 is walking distance derived from comparing user location and bus stop location. In this example our new data might look like this:

	Current time: 18:10					
ID	BusStopName	TotalTime	BusNumber	Time	Duration	WD
1	Location 1	21	9	1825*	6	200
4	Location 4	31	6	1830*	11	100

Figure 4.14 Updated table with new arrival times and total travel time. The * indicates a difference between the planed and actual (real-time) arrival times.

The suggestions we have left now illustrates a common problem or obstacle when only providing one route suggestion. Both of these routes can be considered best depending on what you value. On one hand you can say that suggestion 1 is superior because it arrives 5 minutes earlier than suggestion 4. On the other hand, you can say that suggestion 4 is superior because there is a shorter walking distance. This application will rank these types of suggestions instead of excluding one of them. In this particular implementation the total travel time (The difference between now and arrival at destination) is preferred over walking distance, mainly because one of the questions this project was suppose to answer was 'What is the quickest way from A to B?'.

5. Results and discussion

5.2 Retrieving coordinates

5.2.1 Bus stops

The 500 bus stop locations available in this application are described by four different characteristics. The bus stop id, name, longitude and latitude are provided for each stop. As mentioned earlier the list of bus stops was formatted as an XML file located in the application package. The process of translating and parse the XML file into Location objects is handled when the application starts. The Location objects contain longitude, latitude and a name. These objects are needed for calculating the distance between the bus stops and the device's current location. The problem of only having 1 longitude and latitude pair for each bus stop location (e.i not one for each direction) is not really restricting the intelligence of the application at this point.

The bus stop list used in this application was a union between the bus stop list provided by AtB (containing name and coordinates in Google Maps format) and a bus stop list from BusTUC (containing the bus stop id, name and coordinates in another format).

5.2.2 Device location

As mentioned earlier, there are three different location providers available in the framework. This version of the application uses the network provider, which requires internet access. This is due to the fact that making an application which would prioritize providers after what type of resources it had at a given time, requires more testing than I had the time to do. Additional information about this is available under the chapter of further research.

There are some inconsistencies in deriving the device location due to the fact that the application uses the network provider. These happened when the device was inside and used assisted GPS for the initial satellite fix.

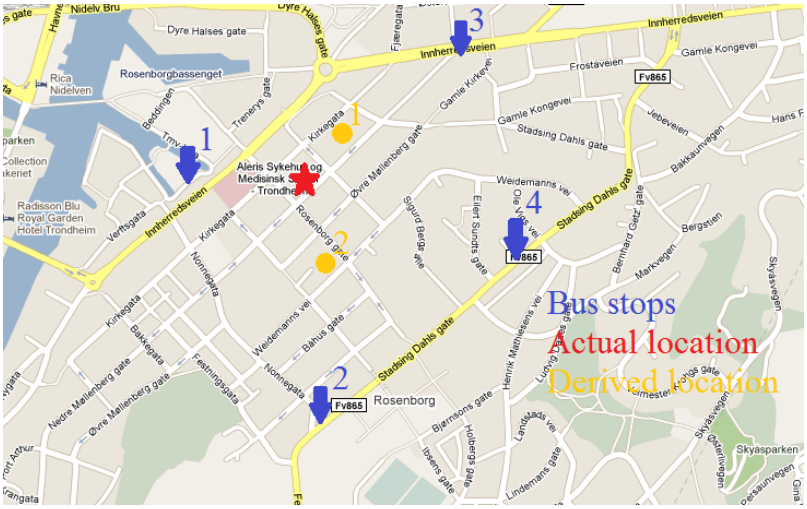

Figure 5.1 Overview

This is an overview over the results received when using this application from Nedre Møllenberg Gate 68, Trondheim. The red star represents the actual location. My returned locations (circle 1, circle 2) are represented by the orange dots and the bus stops are the blue arrows.

While working on this project, the application has received circle 2 a couple of times as position even though the smartphone was located in more or less the same place (the couch of my apartment). The rest of the time (95% ca) the position returned has been circle 1. The cause is unknown, but it might be the ISP.

5.3 Integrating BusTUC

The integration of BusTUC worked very well for this application. When using the methods described under in chapter four the application managed to suggest adequate routes for every destination under testing. The extension is also highly flexible and the output (Route suggestions figure 5.3) from BusTUC is very predictable.

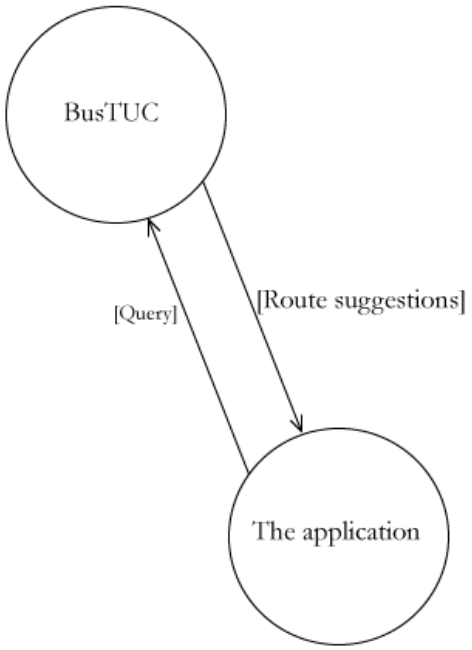

Figure 5.2 BusTUC input and output.

The only problem is that the server can be a little slow in returning the suggestions. This is due to the fact that the extended version of BusTUC is currently hosted on a NTNU server which is relevantly slow. If the android application was connected to a extended version of BusTUC running on a commercially viable server, this would not be a problem.

5.4 Getting real-time data

The process of gathering the real-time data was relatively straight forward. The methods for extracting the unique id numbers as described in chapter 4.3.2 worked well. One thing worth mentioning here is that it may sound unnecessary to download these id numbers (figure 5.3) each time the application loads, but it is really the only way. There are no other ways of knowing if the numbers have been changed, than to actually download the numbers. And so if they have been changed, there is no use to compare them with older numbers just to make sure that they've changed of course.

Unique ID	Bus stop number
111111	16000001
111112	16000002
111113	16000003
……..	……..

Figure 5.3 The list generated the GetBusStopList reponse.

The system which provides the real-time data is hosted on AtB's own servers and is used by them as well.

The gathering of arrival times worked also quite well, even though you could only specify bus stop (not bus number, see figure 5.4). If you could specify bus stop and bus number, some of the workload could have been taken off the application.

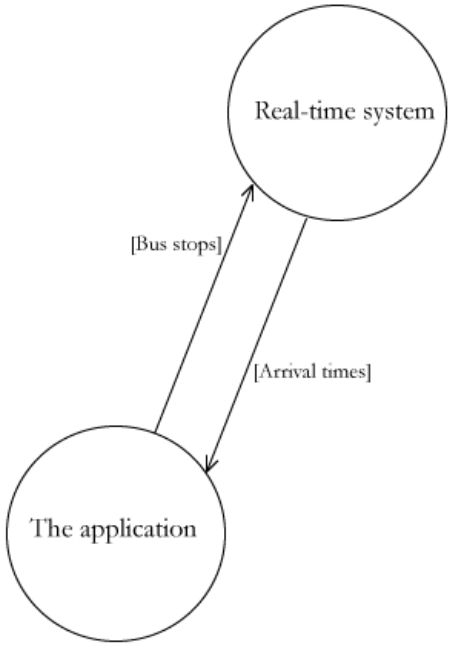

Figure 5.4 Real-time system input and output

5.5 Rank

The ranking as described in chapter 4.7 worked very well. As stated there, the score is built on two assumptions:

1. The user will not take a bus from a bus stop further away when the same bus is available at a bus stop closer to him.

2. The user would like to arrive at his destination as early as possible (unless routes at specific times are requested).

The application will produce around 1-3 suggestions based on these assumptions, all varying in walking distance and time of destination arrival. The main reason there isn't just one suggestion is that there is no objective way of saying one is better than the other. Some might prefer the least amount of walking needed; some might prefer getting to their destination faster. And considering a route suggestion is easily explained in one sentence of natural language (e.g. Bus 9 arrives at Solsiden 13:00), the loss of excluding some of them greatly outweighs the loss of including them all.

5.6 Examples

This chapter will contain examples from three different locations in Trondheim with varying query complexity. The examples will be explained and the data from the external systems (BusTUC and real-time) will be presented as well as the final result.

5.6.1 Location 1

The first location is 'Nedre Møllenberg gate 68' in Trondheim. A *correct* map is shown below to illustrate the accuracy of the application.

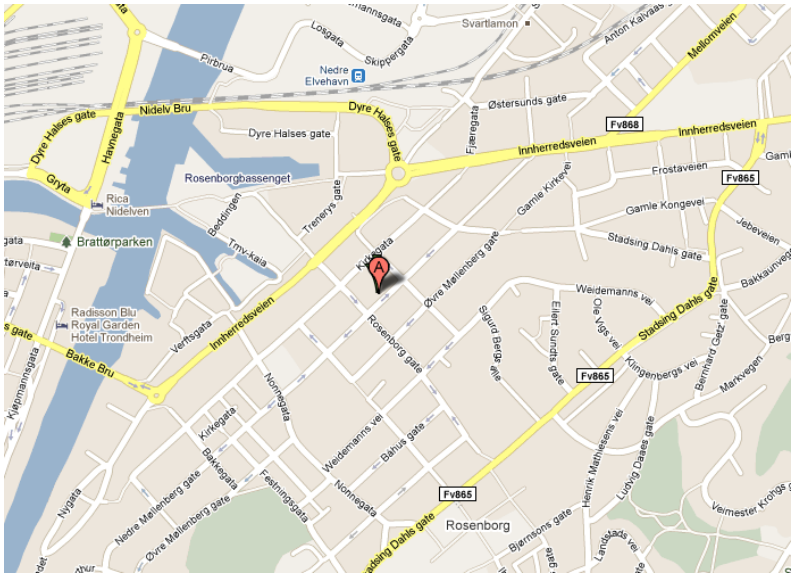

Figure 5.5 The actual location

5.6.1.1 Downtown

After deriving the user location, the application finds the relevant bus stops and displays them on the map. At the top of the application, the input field (where the user types in his destination, currently occupied by the text 'Hvor vil du dra?/Where do you want to go?') and a send button is presented.

Figure 5.6 The blue thumbnail which states 'BUS STOP' represents the corresponding stop, while the pin represents the user.

The slight difference in actual position (figure 5.5) and position displayed by the application (figure 5.6) is due to GPS accuracy. The difference is small enough as to not disturb or invalidate the upcoming search for best possible travel route. This occurs in some of the other examples as well, and will be pointed out there. It is also evident that only three bus stops are visible on the map. This is because the map is zoomed in. If the user decides to zoom out, he will discover two more bus stops which will be included in the query (they would have been included if he had not zoomed as well). The bus stop search parameter is a 500 meter radius around the users' correct position and a maximum of 5 bus stops.

Figure 5.7 The map zoomed out from initial scale.

It is worth pointing out that the pin is has been moved slightly to the north after zooming out (figure 5.6 and 5.7). This was explained in chapter 4.1.

In this example our desired destination is going to be downtown Trondheim. This is where the only user generated input is required, namely typing in 'sentrum' in the text field. As mentioned in chapter 4.5 'Extending BusTUC', the user can use a wide variety of natural language queries, but in this example only 'sentrum' is used. Additional examples with different type of queries will follow. After pressing send, the formatted query is sent to BusTUC.

Figure 5.8 The user typing in his wanted destination

The query was sent at 18.14.

Query:

Norwegian: *(Dokkparken, Strandveien, Sig. Bergs Allè, Buran, Rosenborg skole) til sentrum*

English: *(Dokkparken, Strandveien, Sig. Bergs Allè, Buran, Rosenborg skole) to downtown*

The response is formatted in JSON, but showed below as a table for easier reading.

Query sent: 18.14						
Id	BusStopName	BusStopNr	BusNr	Time	Duration	Destination
1	Strandveien	16011470	3	1819	6	Sentrumsterminalen
2	Strandveien	16010470	36	1820	42	Sentrumsterminalen
3	Buran	16010077	36	1821	41	Sentrumsterminalen
4	Buran	16011077	66	1822	6	Sentrumsterminalen
5	Strandveien	16011470	66	1823	5	Sentrumsterminalen
6	Buran	16011077	6	1830	8	Sentrumsterminalen

Figure 5.9 Response list

One thing worth noticing here is that the BusTUC has 'changed' the original destination from sentrum (downtown) to sentrumsterminalen (Downtown terminal). This is because there is not an actual bus stop named sentrum, so BusTUC provides the actual name of the bus stop.

Before retrieving real-time data based on the suggestion given by BusTUC, some of these values are filtered out due to some of the previous stated assumptions (For a specific bus line, the user will always travel from the closest bus stop). Suggestion 3 and 4 is therefore ignored because Strandveien is closer to walk than Buran.

The bus stop numbers are set as parameters in the XML SOAP query sent to ATB's server.

The application receives the real-time data and updates the arrival time of the buses.

Query sent: 18.14							
Id	BusStopName	BusStopNumber	BusNumber	Time	Duration	Distance	Destination
1	Strandveien	16011470	3	1818*	6	264	Sentrumsterminalen
2	Strandveien	16010470	36	1820	42	264	Sentrumsterminalen
5	Strandveien	16011470	66	1828*	5	264	Sentrumsterminalen
6	Buran	16011077	6	1836*	8	462	Sentrumsterminalen

Figure 5.10 Updated time with real-time data and air distance.

3 out of 4 suggestions had a change in arrival time after receiving the actual time. Suggestion 1 shows that bus number 3 actually arrives a minute earlier than expected. Suggestion 3 and 4 is delayed 5 and 6 minutes respectively.

The last data required before making an intelligent route suggestion is total travel time. Remember that the time was 18.14 when this query was made. Total time would be the time from 18.14 until the bus arrives at the users' destination.

Query sent: 18.14							
Id	BusStopName	BusNumber	Time	Duration	Distance	TotalTime	Destination
1	Strandveien	3	1818*	6	264	10	Sentrumsterminalen
2	Strandveien	36	1820	42	264	48	Sentrumsterminalen
5	Strandveien	66	1828*	5	264	19	Sentrumsterminalen
6	Buran	6	1836*	8	462	30	Sentrumsterminalen

Figure 5.11 The updated results with total travel time.

The obvious outcast here is Suggestion 2. The bus travel duration is 42 minutes and would not have been preferred by the user. A method for excluding them has not been implemented in this version of the application. The issue is discussed in chapter 7.5, further research and development, together with a possible solution for excluding these types of 'stupid' suggestions.

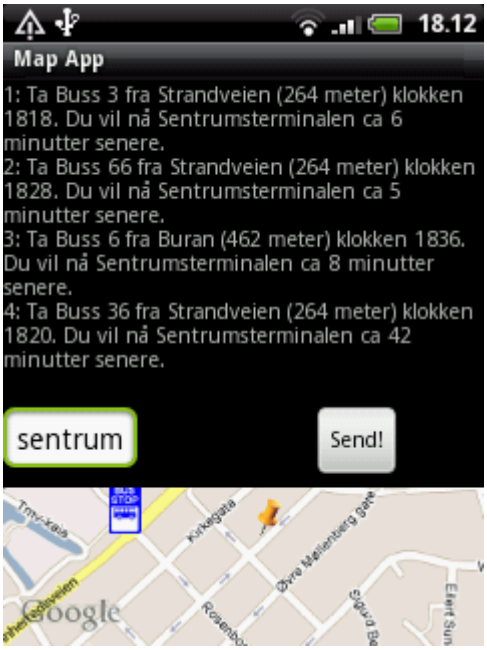

Figure 5.12 The final result displayed

After ranking, sorting and displaying the four unique suggestions, the application is available for a new bus route search.

5.6.1.2 Nardo

The main objective of this example is to show the applications ability to handle transfers as well. The user location is still at 'Nedre Møllenberg Gate 68' (figure 5.5) Whether or not the route is a transfer is provided as the transfer boolean (chapter 4.3) in the data returned from BusTUC. The destination chosen is 'Nardo'.

Query:

Norwegian: *(Dokkparken, Strandveien, Sig. Bergs Allè, Buran, Rosenborg skole) til nardo*
English: *(Dokkparken, Strandveien, Sig. Bergs Allè, Buran, Rosenborg skole) to nardo*

The query is still a simple 'to' query.

Response:

Query sent: 22.31 Transfer: True						
Id	BusStopName	BusStopNr	BusNr	Time	Duration	Destination
1	Strandveien	16011470	3	2246	7	Munkegata –M3
1	Munkegata – M3	16010003	52	2300	11	Nardosenteret

Figure 5.13 Transfer response

The arrival times (arrival at Munkegata for bus number 3 and Nardosenteret for bus number 52) of the buses are still updated with real-time data even though there is only one transfer suggestion. There is still great value in providing correct arrival times for the user. The original destination sat by the user ('nardo') has been replaced by the actual bus stop ('Nardosenteret').

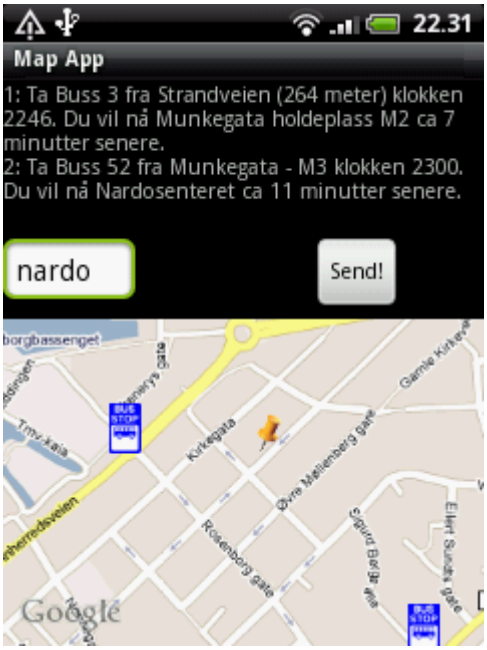

Figure 5.14 Result after 'Nardo' query.

The application is able to handle bus routes which include transfers, which is a necessary functionality for any smart bus route information system.

5.6.1.3 Nardo after 15.00

This example is from the same location (figure 5.5) as the previous ones, but with a time restriction.

Query:
Norwegian: *(Dokkparken, Strandveien, Sig. Bergs Allè, Buran, Rosenborg skole) til nardo etter 15.00*
English: *(Dokkparken, Strandveien, Sig. Bergs Allè, Buran, Rosenborg skole) to nardo after 15.00*

Figure 5.15 Query with time restriction

Response from BusTUC:

Query sent: 23.09 Transfer: True Timeset: True						
Id	BusStopName	BusStopNr	BusNr	Time	Duration	Destination
1	Strandveien	16011470	66	1443	5	Munkegata –M3
1	Munkegata – M3	16010003	52	1455	11	Nardosenteret

Figure 5.16 Response from BusTUC with time restriction

Since the query contained a set time (or time constraint), the arrival times set by BusTUC is presented to the user. This is of course because this query was sent at 23.08, almost 15 hours before the bus arrives.

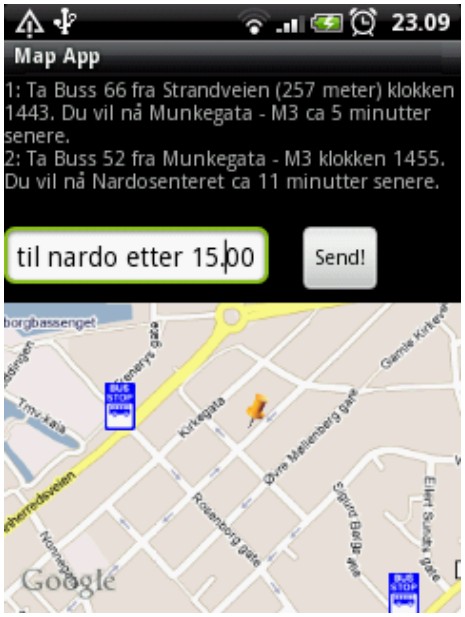

Figure 5.17 Result with time restriction

5.6.2 Location 2

This time the user is located at Olav Tryggvasons gate 22. Here is a picture from Google Maps for the actual position.

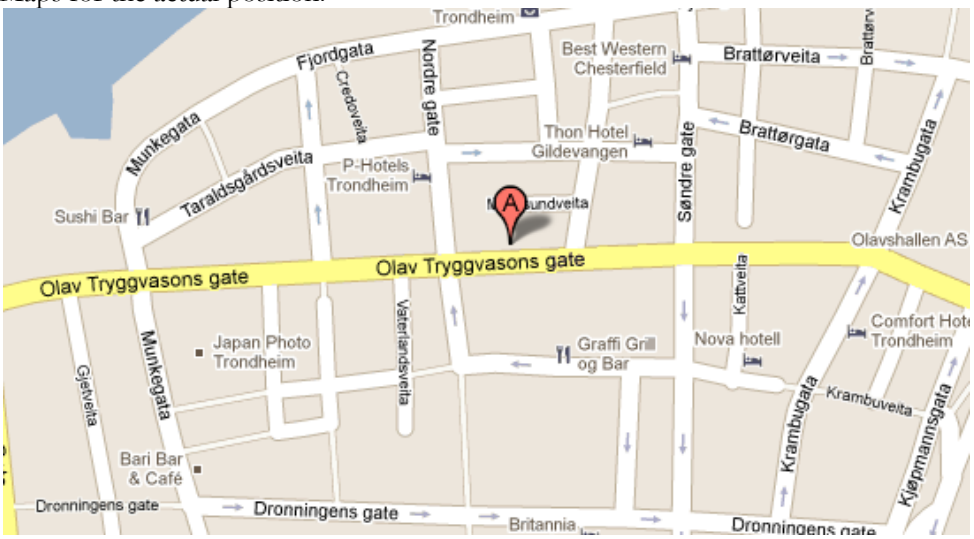

Figure 5.18 Actual position of user.

5.6.2.1 Solsiden

The desired destination is 'Solsiden' this time. The user location and adjacent bus stops are shown on the application.

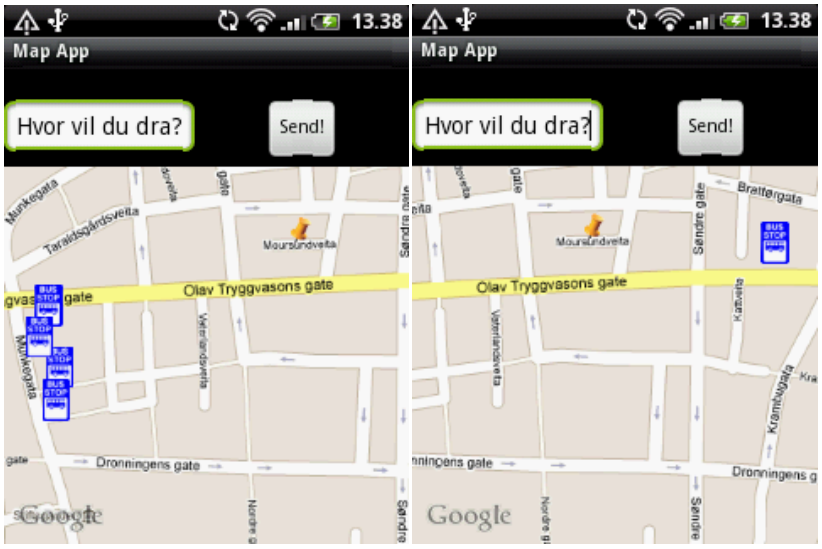

Figure 5.19 The picture on the left is focused east and right picture is focused east.

The search radius has been set for 500 meters and the maximum amount of bus stops is 5.

After typing in 'Solsiden' the following query is sent to BusTUC:

Norwegian: *(Nova Kinosenter,Munkegata M5,Munkegata M4,Munkegata M1,Munkegata M2) til solsiden*

English: *(Nova Kinosenter,Munkegata M5,Munkegata M4,Munkegata M1,Munkegata M2) til solsiden*

BusTUC finds the possible routes and returns the following suggestions:

Query sent: 13.38						
Id	BusStopName	BusStopNumber	BusNumber	Time	Duration	Destination
1	Munkegata - M5	16010005	6	1340	4	Solsiden
2	Munkegata - M4	16010004	36	1345	4	Solsiden
3	Munkegata - M5	16010005	4	1345	4	Solsiden
4	Nova Kinosenter	16010022	9	1345	2	Solsiden
5	Nova Kinosenter	16010022	36	1347	2	Solsiden
6	Nova Kinosenter	16010022	4	1347	2	Solsiden

Figure 5.20 Data received from BusTUC

The application narrows the list down to only one suggestion per bus number (the closest bus stop) and gets the real-time data.

Query sent: 13.38							
Id	BusStopName	TotalTime	BusNumber	Time	Duration	Distance	Destination
1	Munkegata - M5	5	6	1340	4	215	Solsiden
4	Nova Kinosenter	7	9	1344*	2	160	Solsiden
5	Nova Kinosenter	10	36	1347*	2	160	Solsiden
6	Nova Kinosenter	9	4	1346*	2	160	Solsiden

Figure 5.21 Suggestions filtered

Here you can see that both suggestion 2 and 3 gets thrown away because 'Nova Kinosenter' provides the same bus route closer to the user.

Suggestion 1 will be come up on top in this scenario because it will get the user quickest to Solsiden.

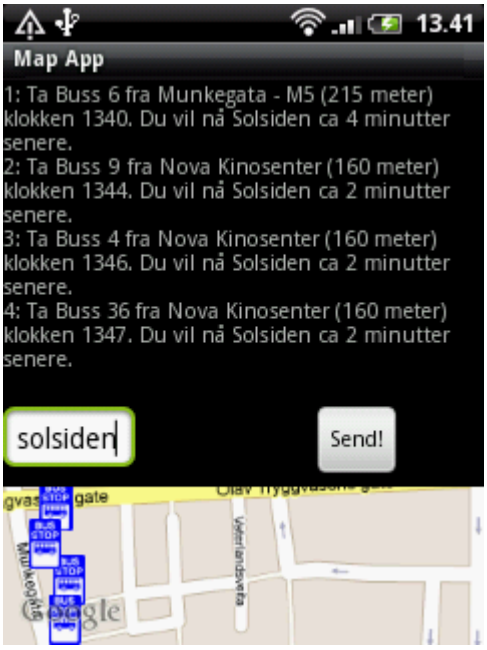

Figure 5.22 Result presented to the user

5.6.2.2 Gløshaugen

Same location as the previous example, Olav Tryggvasons gate 22 (figure 5.18) but different destination: Gløshaugen. This travel requires a transfer, so BusTUC returns this information:

Query sent: 13.43						
Id	BusStopName	BusStopNumber	BusNumber	Time	Duration	Destination
1	Munkegata – M2	16010002	6	1350	2	Torget
	Torget	16010012	52	1357	5	Gløshaugen Syd

Figure 5.23 Returned route suggestions

The planned arrival times are updated with the actual arrival times:

Query sent: 13.43							
Id	BusStopName	TotalTime	BusNumber	Time	Duration	Distance	Destination
1	Munkegata – M2	9	6	1350	2	245	Torget
	Torget	5	52	1357	5		Gløshaugen Syd

Figure 5.24 Updated with real-time data. No changes

No changes and the results are presented to the user. As you can see, the total travel time for 'Munkegata – M2' is 9 minutes. This is from 'now' (13.43) to arrival at Torget (13.52). The application knows that this is a transfer and presents the two connecting travel routes in the correct order.

Figure 5.25 Final result for Gløshaugen

The zoom bug is present again in figure 5.25 after zooming out. The pin has moved itself further up on the map, representing the user location somewhat wrong (again, the actual values used for calculations are not changed).

5.6.2.3 Dragvoll

Last example from Olav Tryggvassons gate 22 (figure 5.18). This time the destination is 'Dragvoll'.

This query received two bus route suggestions, one from 'Munkegata – M5' and one from 'Nova Kinosenter'. Both these suggestions had the same bus number, 9, so 'Nova Kinosenter' is the only presented suggestion due the fact that it is located closer to the user.

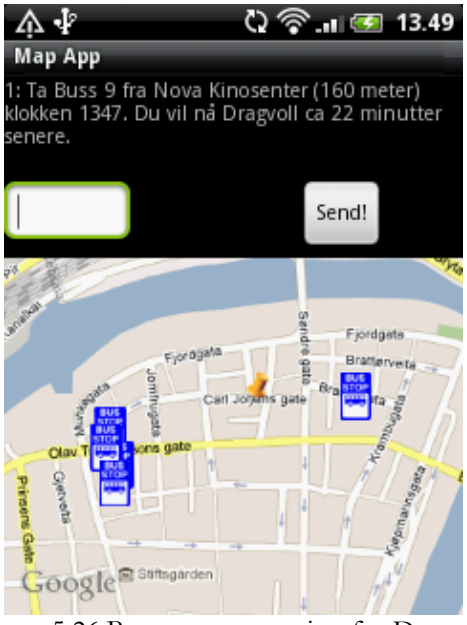

Figure 5.26 Bus route suggestion for Dragvoll

5.6.3 Location 3

The last example is from Nygata 25 and only one example will be provided from this location. This is a more remote (read: less bus stops nearby) location than the others. As you can see, the application chooses the only two bus stops available within a 500 meter radius.

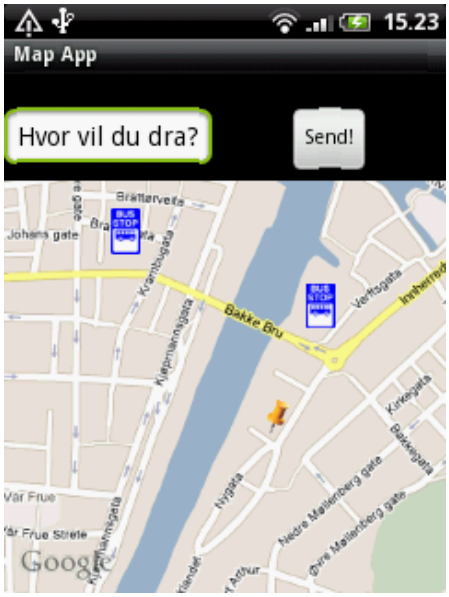

Figure 5.27 Two available bus stops

5.6.3.1 Nardo

In this example the destination is Nardo. The two bus stops picked out by the selection algorithm is sent to BusTUC with the destination:

Query:

Norwegian: *(Bakkegata,Nova Kinosenter) til nardo*

English: *(Bakkegata,Nova Kinosenter) til nardo*

Query sent: 15.25						
Id	BusStopName	BusStopNumber	BusNumber	Time	Duration	Destination
1	Nova Kinosenter	16011022	66	1532	1	Munkegata – M3
	Munkegata - M3	16010003	52	1540	11	Nardosenteret

Figure 5.28 Results returned from BusTUC

There is no direct route between any of the two bus stops and 'Nardo', so BusTUC presents the best transfer. The arrival times are updated and the final suggestion is given to the user.

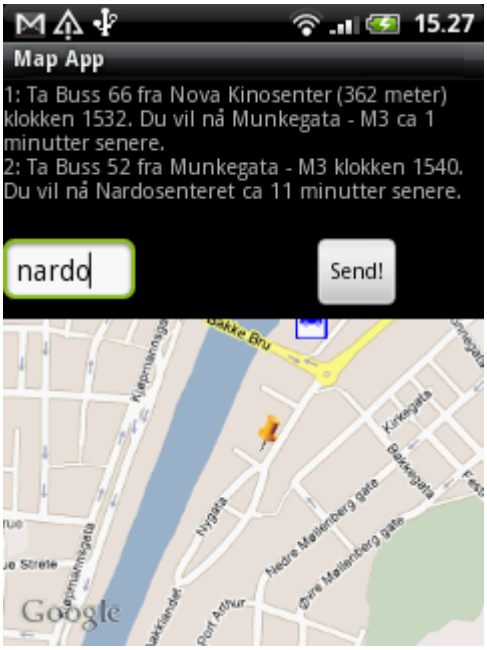

Figure 5.29 Final suggestion for Nardo

6. Conclusion

The developing of applications for smartphones is an exciting field. Even though the technology is fairly new, high expectations are set by the end-users. The smartphone programs need to be intuitive, fast and the data correct. Google knew this when developing the Android API. It is fairly well documented and any information missing can usually be found on the Google code message boards. The integration of Google Maps in Android applications is relatively easy and the functionality it provides is well documented. There weren't many problems at this point. The only thing required was internet access, may it be internet or cellular.

The focus of this project was to create the architecture for a generic public transport suggestion system and it was in many ways successful. The combination of GPS coordinates, real-time data and BusTUC provided enough information to accurately give users detailed route suggestions. The application is also able to exclude some parts of its functionality and still be useful. If a town or area decides to implement BusTUC without having a real-time system, the application is still able to provide users with the locations of themselves and bus stops as well as route suggestions.

The development went fairly well. The language used was Java, but some time was spent learning the Android specific rules and techniques.

One of my biggest concerns throughout this project was accessing the real-time data provided by AtB. They were originally going to be made available in the autumn of 2010, but where delayed until March 2011. This naturally caused some delays regarding the implementation, but it worked out. There were no problems actually accessing and using the data after they were made available.

The extension and integration of BusTUC word very well. A complete set of initial route suggestions are provided by BusTUC after a single call from the application.

The way the suggestions are ranked, with exception of some outliers (chapter 7.5), are very efficient and based on solid reasoning. Keep in mind that the only input generated from the user in this application is the destination. Without requiring more information from the user regarding his/her preferences there are no objectively way to exclude some of the final suggestions.

The application is added as an digital appendix and is also available at http://e-pos.info/magge/BusTUCwGPS-RT.apk

7. Further research and development

The main areas that need improvement before making this application available for the public are graphical design, testing, evaluation and extension of functionality.

7.1 Graphics

Not much effort has been spent on making this application look aesthetically pleasing. An example of how this could be improved is shown below. The language in the example is Norwegian.

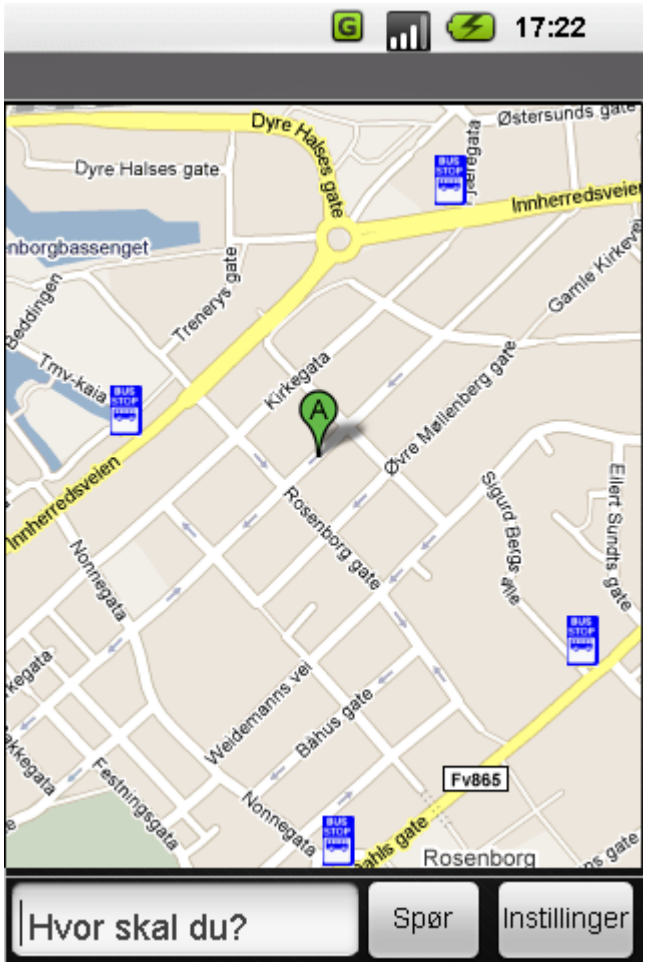

Figure 7.1 Start picture

When starting the application a window which includes a map, query box and a preferences button should appear (Figure 7.1). The map should include user location (the green 'A') and the bus stops nearby. The user should be able to freely navigate the map. When selecting (press on the map) one of the bus stops a new window containing information about the buses which arrives first will appear.

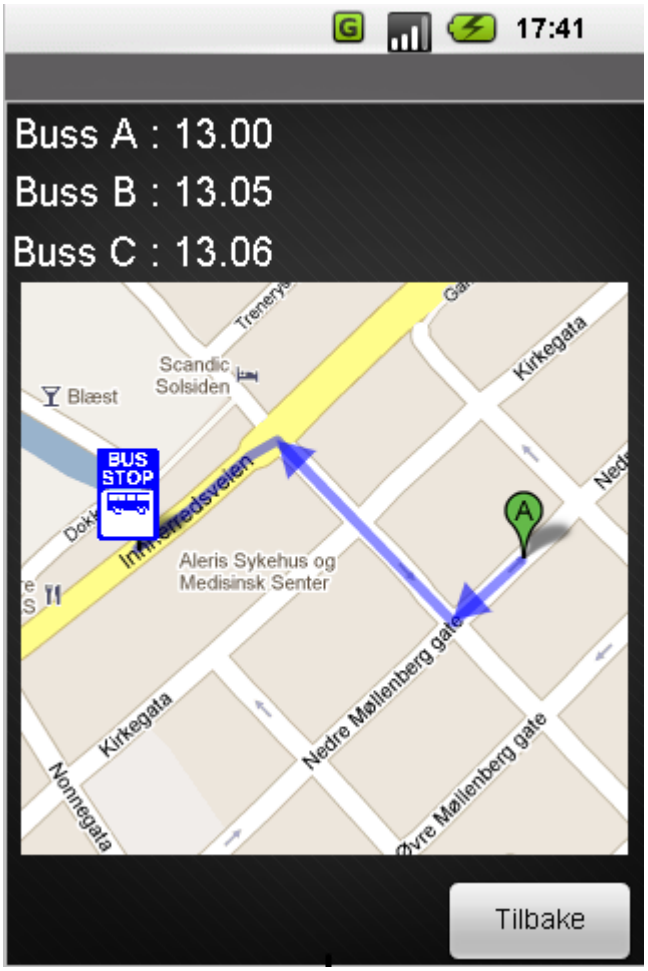

Figure 7.2 Information about a specific bus stop

The new window will also include a walk suggestion for the user. The times displayed in this window will not be the planed arrival times, but the actual times of arrival derived from the real-time system. All of this functionality exists in the application, so extending the application to be able to have these types of 'individual windows' for each bus stop would not be hard.

As mentioned previously in this paper, the problem of narrowing down the number of suggestions without knowing any of the preferences of the user is difficult. So if we extend the application to include a set of options for the user, it would be able to rank or even exclude some of the suggestions better.

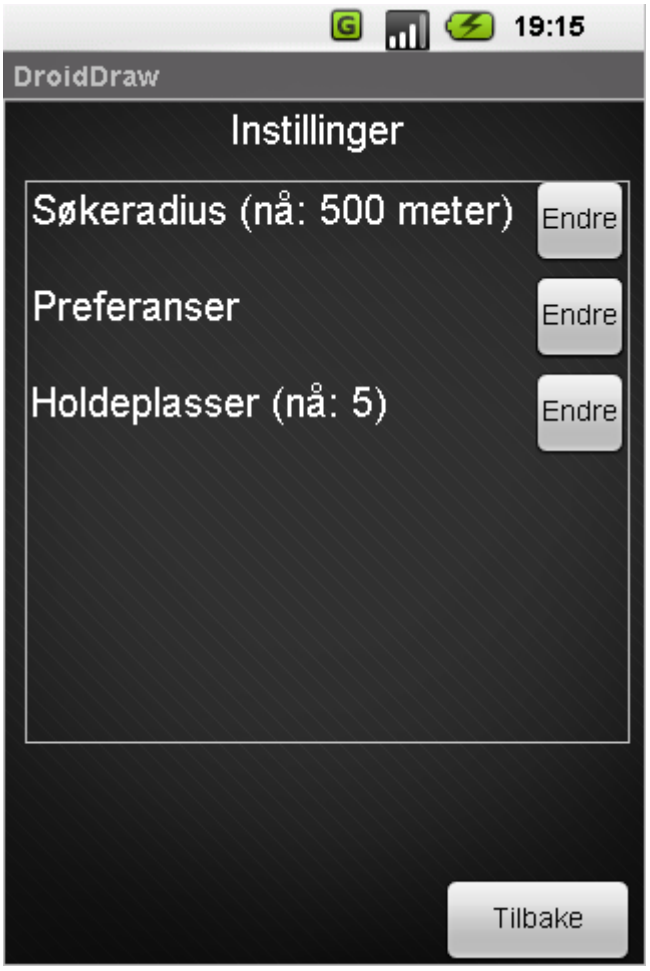

Figure 7.3 The different options available for the user

The user is able to set the search radius and the maximum amount of bus stops. If the user sets search radius to 500 meters and bus stops to 0, the application will find every bus stop within 500 meters. If the search radius is set to 0 and the bus stops to 10, the application will find the 10 closest bus stops regardless of distance between the user and the actual stop.

For a more day to day use, the user would perhaps choose the values shown in figure 7.3. The search radius excludes useless bus stops when you find yourself in a suburban area, while the number of bus stops limits the workload when operating in an area with many bus stops.

The last option is the 'preferences' option. The values set here would affect the ranking of suggestion.

Figure 7.4 Preferences

As shown in Figure 7.4, we can let the user set his/her preferred walking speed. Since the distance between the user and the bus stop is already known the unobtainable routes can be excluded. The average walking speed is about 5 kilometers per hour. Slightly less for older individuals and slightly higher for younger individuals[7]. Let's say that low is set at 2.5 km/h, average is 5 km/h and fast is 7.5 km/h. If the bus arrives at the bus stop in 10 minutes and it is 1 kilometers away it will only be useful for a fast walker.

The last option is to decide if the user prefers a short walking distance or the earliest possible arrival time. Some suggestions might 'survive' all the explained stages of exclusion so this will help us rank according to the user's taste.

7.2 Testing

Developing mobile applications can sometimes be tricky. I've encountered weird exceptions while developing and some even weirder ones while testing. Most exceptions should be caught, but it should go through a more rigoursly testing period before being available to the public.

[7] Pedestrian Walking Speed - http://www.usroads.com/journals/p/rej/9710/re971001.htm

7.3 Evaluation

A small focus group should perhaps be used while developing further. This can help in finding exceptions and get general feedback from people who will use it. The questionnaire given to each test user should at least include questions such as:

- Is it easy to use?

- Were the suggestions given smart? Give a specific example of when it was not.

- Did it return any errors? Give a specific example of when it did.

7.4 Extending functionality

As mentioned earlier in this paper, the lack of user input makes it difficult to narrow down the suggestions provided to the user. By having the user set his own preferences, some suggestions could be filtered out.

- How far are you willing to walk?

This could set the radius down to perhaps only include 1 or 2 bus stops even in areas with higher concentrations of bus stops.

- What is your walking speed?

Since the application knows how far the user is from any given bus stop, it could exclude suggestions which the user will not catch.

7.5 Removing 'stupid' suggestions

In one of the examples, there were a couple of outliers which should have been excluded, but did not.

Query sent: 18.14							
Id	BusStopName	BusNumber	Time	Duration	Distance	TotalTime	Destination
1	Strandveien	3	1818*	6	264	10	Sentrumsterminalen
2	Strandveien	36	1820	42	264	48	Sentrumsterminalen
5	Strandveien	66	1828*	5	264	19	Sentrumsterminalen
6	Buran	6	1836*	8	462	30	Sentrumsterminalen

Figure 7.5 Example of 'stupid' suggestion

The suggestions shown in figure 7.5 were presented to the user in example 5.6.1.1. The second suggestion has a 42 minute bus travel and no one would prefer it. One simple way to remove these types of suggestions is to look at the quickest (the suggestion with the shortest total time, total time being now until arrival at destination) from the same bus stop.

Let's say that the smallest total time for 'Strandveien' is 10 minutes (figure 7.5). Every suggestion that starts at 'Strandveien' should have a smaller total time than 10*2 = 20, in other words: every suggestion provided at each bus stop cannot have a longer total time than double of the quickest.

This is easily implemented and would remove any silly suggestions.

8. Implementation

A simple description of the actual implementation follows in this chapter. The components and the most important methods are mentioned and described.

8.1 Application

The application consists of 5 main components: Controller, browser, formatter, calculator and GUI.

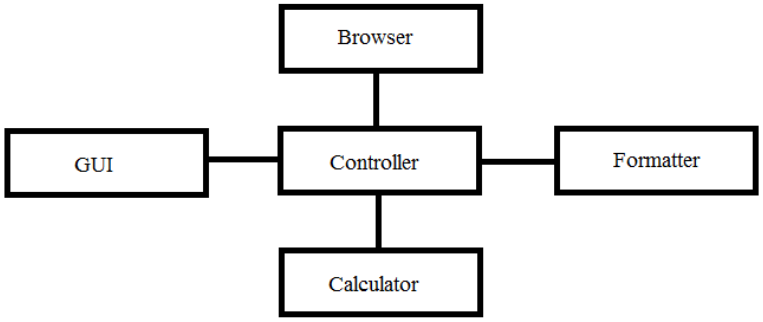

Figure 8.1 Abstract view of the application

8.1.1 Controller

This component handles all the information flow and is in charge of communicating with the user interface: It handles the events generated by button pushes etc and sets the text shown to the user.

This component is also in charge of starting, maintaining the 'Activity' and implementing listeners.

Class	Method	Description
Controller	onCreate()	1. Sets the layout. 2. Creates the list of bus stops 3. Initiates Google Maps 4. Creates Browser objects 5. Adds location listener for GPS 6. Adds button listener for requests
Controller	onResume()	1. Sets the requirements for location listener. (When to update the coordinates)
Controller	partialsort()	1. Takes all the bus stops and sorts them according to parameters(radius and max bus stops)
MapOverlay	Draw()	1. Draws the bus stop signs and pin (user) on Google Maps
LocationListener	onLocationChanged()	1. Runs the partial sort method 2. Updates the relevant points for MapOverlay 3. Zooms the map to the current user position
ButtonListener	onClick()	1. Sends the relevant bus stops to the browser 2. Create route objects from BusTUC response 3. Requests the real-time arrival times from browser

		4. Sends the final solutions to the calculator 5. Displays the result

Figure 8.2 Description of the most important methods in the Controller component

8.1.2 Browser

This component handles all the internet traffic necessary to run this application. It has general methods which send POST or SOAP requests and sends either the pure html or xml to the formatter.

Class	Method	Description
Browser	sendPOST()	1. Sets up the POST requests 2. Sends the POST requests
Browser	sendSOAP()	1. Receives SOAP Header and message 2. Sends the SOAP message
Browser	specificRequest()	1. Receives bus stop id and bus number 2. Calls sendSOAP() with SOAP Header and message
Browser	getBusStopList()	1. Sends the bus stop list SOAP message

Figure 8.3 Description of the most important methods in the Browser component

8.1.3 Formatter

The formatter component handles all encoding of outputs or decoding of inputs. It is required to handle different languages (XML,JSON,HTML), but luckily the inputs are highly predictable.

Class	Method	Description
GetGPS	fCords()	1. Loads the XML file with the bus stop list containing bus stop number and coordinates 2. Creates a list
Browser	parseRealTimeData()	1. Receives the SOAP response 2. Extracts the JSON string from the XML and creates an object
Browser	parseHTTP()	1. Receives the suggestions from BusTUC 2. Extracts the JSON string from the HTML and creates an object

Figure 8.4 Description of the most important methods in the Formatter component

8.1.4 Calculator

After all the data has been formatted correctly it gets sent to the calculator. This component includes objects for routes and bus stops, as well as a ranking algorithm. The output from this component gets sent to the Controller which displays it on the GUI.

Class	Method	Description
Route	Constructor	1. Creates a route suggestion object with arrival time, bus stop name, bus stop number, bus number, travel time, destination, travel, timeset, walking distance and total time.
Calculator	createRoutes()	1. Retreives the JSON object containing route suggestions

		2. Creates an array of route objects
Calculator	suggestRoutes()	1. Retrieves the route suggestion array 2. Ranks them
Calculator	calculateTotalTime()	1. Finds the current time and calculates total travel time

Figure 8.5 Description of the most important methods in the Calculator component

8.1.5 GUI

Notably the component which requires the most work before this application can be considered commercially viable. It handles the general layout of the graphics as well the different graphical components. The map, text field, button. The map gets 'drawn' on by this component after instructions from the controller.

9. Additional notes

This chapter contains more information about the underlying technologies, the actual requests used in the real-time system and a more detailed description of development for Android.

9.1 Real-time communication

Here are the actual SOAP requests and responses.

SOAP Request – GetBusStopList

```
POST /InfoTransit/userservices.asmx HTTP/1.1
Host: 10.0.2.52
Content-Type: application/soap+xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap12:Envelope
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap12="http://www.w3.org/2003/05/soap-
envelope">
  <soap12:Body>
    <GetBusStopsList
xmlns="http://xxxxxxx.xxx/infotransit">
      <auth>
        <user>string</user>
        <password>string</password>
      </auth>
    </GetBusStopsList>
  </soap12:Body>
</soap12:Envelope>
```

Figure 9.1 Requesting the bus stop list containing unique id numbers

In the SOAP request GetBusStopsList username and password needs to be specified. After receiving the request, the server will send a XML response which contains the new unique id numbers and the corresponding bus stop number.

SOAP Response – GetBusStopList

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap12:Envelope
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">
  <soap12:Body>
    <GetBusStopsListResponse
xmlns="http://xxxxx.xxx/infotransit">

<GetBusStopsListResult>string</GetBusStopsListResult>
    </GetBusStopsListResponse>
  </soap12:Body>
</soap12:Envelope>
```

Figure 9.2 Server respons to GetBusStopList

The information returned from the server is the list of unique id numbers and bus stop numbers.

Unique ID	Bus stop number
111111	16000001
111112	16000002
111113	16000003
……..	……..

Figure 9.3 The list generated the GetBusStopList reponse.

The GetUserReal-timeForecast request for real arrival times.

SOAP Request – GetUserReal-timeForecast

```
POST /InfoTransit/userservices.asmx HTTP/1.1
Host: 10.0.2.52
Content-Type: application/soap+xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap12:Envelope
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap12="http://www.w3.org/2003/05/soap-
envelope">
  <soap12:Body>
    <getUserReal-timeForecast
xmlns="http://xxxxx.xxx/infotransit">
      <auth>
        <user>string</user>
        <password>string</password>
      </auth>
      <busStopId>string</busStopId>
    </getUserReal-timeForecast>
  </soap12:Body>
</soap12:Envelope>
```

Figure 9.4 Request for real-time data

The user, password and busStopId nodes require values.

The response from the real-time system:

SOAP Response – getUserReal-timeForecast

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap12:Envelope
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap12="http://www.w3.org/2003/05/soap-
envelope">
  <soap12:Body>
    <getUserReal-timeForecastResponse
xmlns="http://miz.it/infotransit">
      <getUserReal-
timeForecastResult>string</getUserReal-
timeForecastResult>
    </getUserReal-timeForecastResponse>
  </soap12:Body>
</soap12:Envelope>
```

Figure 9.5 Server response containing the actual arrival time of buses given bus stop

The real arrival times at a specific bus stop:

Bus number	Arrival time
3	14:44
4	14:56
2	15:07

Figure 9.6 The real-time data

9.2 Android

One aspect of developing in android is the term 'Activity'. Activity is the window that is currently active on the smartphone.

9.2.1 Activity

The activity (task) lifecycle is defined by 4 different states:[8]

- Active or running. The activity is in the foreground of the screen
- Paused. It has lost focus, but is still visible. The activity is considered alive, but can be killed by the system in extreme low memory situations.
- Stopped. The application is no longer visible to the user, but retains all state and member information. Generally killed when memory is needed elsewhere.
- Killed. The system can drop the activity from memory by either asking it to finish, or simply killing its process. When displayed again, the program is completely restarted.

[8]Android Activity - http://developer.android.com/reference/android/app/Activity.html

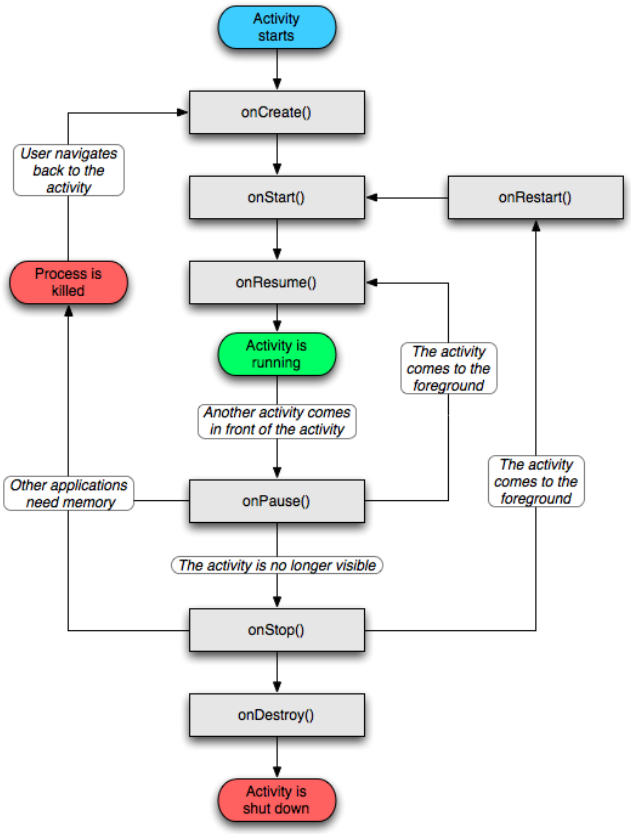

Figure 9.7 Activity life cycle

The MapActivity class (the applications' main class) contains some of these methods to make sure the application behaves properly. A quick overview of the different methods follows:

- onCreate() : This method is called when the activity is first created. This is where the application should bind data to lists, create views, etc.
- onStart(): This method is called when the user can see the activity.
- onResume(): This method is called when the activity interacts with the user. The activity is at the top of the activity stack and input from user is available.
- onPause(): This method is called when a previous activity is about to be the focus of the system. Here you can save data, stop animations etc.
- onStop(): This method is called when the activity is no longer visible to the user, because another activity is covering it. Followed either by onRestart() or onDestroy().
- onDestroy(): This method is called right before the activity is destroyed.

In this application

- onCreate() creates the views, sets the layout and binds the data from the bus stop XML file to a String array. The string array which contains bus stop id, name, latitude and longitude is then split so these values have their own string[x][y] position. This method also starts the GPS location manager, which configures the location service. An event listener for both location and button is created. This is also were the unique id numbers are downloaded and stored.

- onResume() set the restrictions on how often the locationManager should request location updates. In this project it checks if the position has changed more than over one meter, every second. If the position fulfills the criteria set by location manager, the location listener will trigger an event.

9.3 GUI

In this chapter a short description on how GUI modeling works on Android follows.

The GUI is defined by a hierarchy of View and ViewGroup nodes.

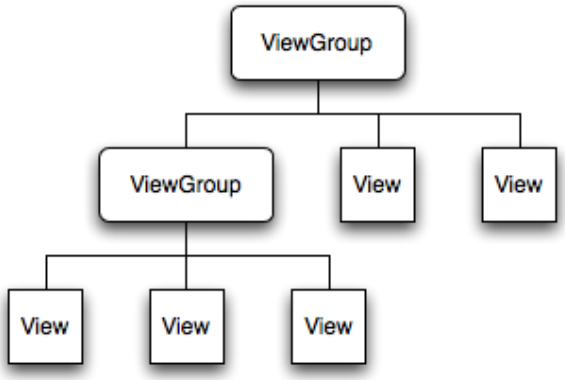

Figure 9.8 The view hierarchy [9]

The view hierarchy and is expressed with an XML layout file unique (figure 11.11)[10]. Each element in XML is either a View or a ViewGroup object (or a descendant thereof). View objects are leaves in the tree and actual visible objects. ViewGroup objects are branches in the tree and determine the layout of their leaves.

When implementing this view hierarchy as XML in the application, the result will look like this:

Figure 9.9 Description of layout

[9] View and Viewgroup - http://developer.motorola.com/docstools/library/Basics_of_Event_Management/images/view_hierarchy.gif/

[10] XML Layout - http://developer.android.com/guide/topics/ui/declaring-layout.html

The LinearLayout node is the root and splits off into two leaves (TextView and MapView) and one branch (RelativeLayout). The MapView is the visible map part at the bottom of the window, TextView is the black area which contains the text and the RelativeLayout branch contains two leaves (EditText and Button). The EditText is the input box which allows the user to generate a query. The button actually sends the query. In this application the view hierarchy is defined as such:

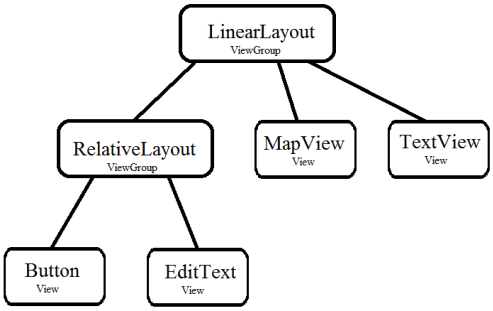

Figure 9.10 The applications view hierarchy

Layout file

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <TextView
    android:id="@+id/myLocationText"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text=""
    />
    <RelativeLayout
        android:id="@+id/widget61"
        android:layout_width="264px"
        android:layout_height="59px"
        android:layout_x="17px"
        android:layout_y="50px"
        >
        <Button
        android:id="@+id/Button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Send!"
        android:layout_alignTop="@+id/eText"
        android:layout_alignParentRight="true"
        >
        </Button>
        <EditText
        android:id="@+id/eText"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hvor vil du dra?"
        android:textSize="18sp"
        android:layout_centerVertical="true"
        android:layout_alignParentLeft="true"
        >
        </EditText>
        </RelativeLayout>
    <com.google.android.maps.MapView
         android:id="@+id/mapView"
         android:layout_width="fill_parent"
         android:layout_height="fill_parent"
         android:enabled="true"
```

```
        android:clickable="true"
        android:apiKey="XXXXXXXXXXXXX"
        />
</LinearLayout>
```

Figure 9.11 The layout file for the application

One of the benefits Google's MapView is that it can be used as a canvas. The pin and bus stop sign is drawn upon MapView to create a visual representation of the locations in question. The icons used are located within the application package and represented as a variable within the development environment. The only necessary operation before drawing is translating the geographical coordinates to screen pixels.

9.4 Hardware and OS

9.4.1 Smartphones

Smartphones are defined as a cell phone that offers more connectivity and advanced computing ability than a 'regular cell phone'[11]. With the ability to install and run more advanced applications, smartphones can be thought of as handheld computers integrated within a mobile telephone. These types of phones run a complete operating system which provides a platform for application developers.

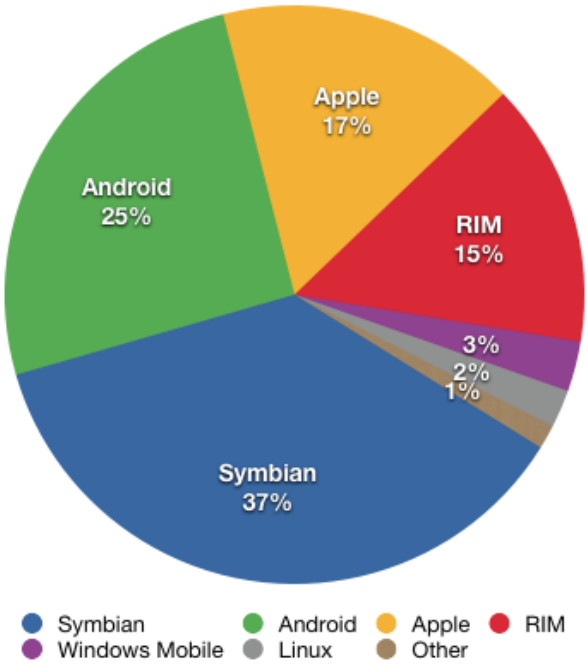

Figure 9.12 Sales of smartphones by operating system[12]

The demand for more computational power, larger screens and open operating systems has made smartphones the dominant product in the mobile phone market.

[11]General information about smartphones - http://en.wikipedia.org/wiki/Smartphone
[12] Research Canalys - http://www.canalys.com/pr/2011/r2011013.html

The smartphone used in this project was HTC Wildfire. It runs Android OS 2.1 on a Qualcomm MSM7225 528 MHz CPU, supported by 512 MB Read-only memory and 384 MB Random-access memory. Wildfire was chosen primarily because of its relatively low price and full GPS support[13].

Figure 9.13 HTC Wildfire

9.4.2 Android OS

Android OS is an open-source operating system initially created by Android Inc, but purchased by Google in 2005. It is based on a modified Linux kernel and consists of Java application running on an object oriented application framework on top of Java core libraries. These core libraries run on Dalvik VM (register-based virtual machine) featuring JIT compilation (just-in-time compilation. A method to improve the runtime performance of computer programs by translating high-level language continuously).[14] The core is generally written in C and the user interface in Java. Some third party libraries are written in C++.

[13] HTC Wildfire - http://www.htc.com/no/product/wildfire/overview.html
[14] Dalvik VM - https://sites.google.com/site/io/dalvik-vm-internals/2008-05-29-Presentation-Of-Dalvik-VM-Internals.pdf?attredirects=0

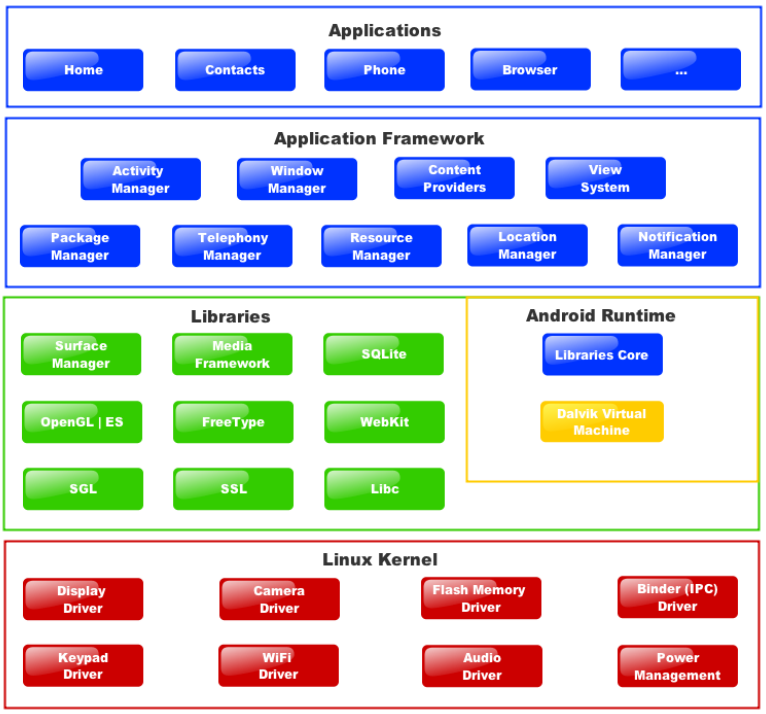

Figure 9.14 The Android OS Architecture[15]

It allows for developers to write application programs that extend the devices' standard functionality. The development occurs in Java, using Google's Java libraries.

Development on Android OS is quite popular and the official market for applications (Android Market) has over 200,000 applications[16] which extend the devices' standard functionality. Developers must use Android software development kit (SDK), which includes a comprehensive set of development tools. Some details on the Android debugger are located in the Implementation chapter.

[15] System architecture - http://androidteam.googlecode.com/files/system-architecture.jpg
[16] Hugo Barro, Product management director, Android - http://googleblog.blogspot.com/2011/05/android-momentum-mobile-and-more-at.html

9.4.2.1 Libraries

Figure 9.15 The available libraries

Android includes libraries written in C/C++ used by different components of the Android system. The functionality provided by these libraries can be used by developers through the Android application framework. The primary libraries[17] are:

- Surface Manager: handles access to the display subsystem and composites 2D and 3D layers of graphic from multiple applications.
- Media Libraries: support playback and recording of many image files, audio and video formats, including MPEG4, H.264, MP3, AAC, AMR, JPG and PNG.
- SGL: underlying 2D graphics engine.
- FreeType: vector font and bitmap rendering.
- SQLite: lightweight relational database engine.
- 3D libraries: use either hardware 3D acceleration or the included, highly optimized 3D software rasterizer
- System C library: a BSD-derived implementation of the standard C system library (libc), modified for embedded Linux-based devices.

The database (SQLite) was not used in this application.

9.5 Global Positioning System

9.5.1 General description

The Global Positioning System (GPS) is a satellite-based navigation system created by a network of (originally) 24 satellites[18]. The satellites were controlled by the U.S Department of Defense and GPS was originally intended for military applications[19]. The system provides reliable location and time information as long as there is an unobstructed line of sight to four or more GPS satellites. No matter what weather, hour of the day or where you are. In the 1980's the U.S Department of Defense made GPS available for everyone with a GPS receiver.

The satellites circle the earth in a very precise orbit twice a day and sends signal information to earth. This information is picked up by the GPS receivers who use triangulation to

[17]What is Android? - http://developer.android.com/guide/basics/what-is-android.html
[18] General information about GPS - http://en.wikipedia.org/wiki/Global_Positioning_System
[19] Trimble - http://www.trimble.com/gps/whygps.shtml

calculate the user's exact location. Triangulation works like this[20]: The GPS receiver compares the time a signal was transmitted by a satellite with the time it was received. This difference allows the GPS receiver to calculate the actual distance to the satellite. When combining these distance measurements from multiple satellites, the receiver can pin point the user's position. The accuracy of the position will naturally increase with the amount of available satellites. 3 satellites is enough to calculate a 2D position (longitude and latitude), but with 4 satellites you'll be able to get the altitude as well.

The network of satellites is orbiting the earth at about 20,200 kilometers in the Medium Earth Orbit. They make two complete orbits in less than 24 hours traveling at speeds of roughly 11,300 km/h. They gather energy from the sun, but carry backup batteries onboard in case of a solar eclipse.

Most receivers now have a parallel multi-channel design. [21]This means that they have 12 parallel channel receivers which are quick to lock onto satellites and maintain strong locks. They are considered extremely accurate even within urban environments with tall buildings.

The signals that satellites send out are called L1 and L2. These are low power radio signals. L1 is dedicated for civilians and has a frequency of 1575,42 MHz in the Ultra High Frequency (UHF) band. As mentioned the signals travel by line of sight, meaning they will pass through obstacles like clouds and plastic but struggles more with solid objects such as mountains and buildings. The signal contains 3 different elements. Ephemeris data, almanac data and an identification code[22]. The ephemeris data contains important information about the status of the satellite (healthy or unhealthy), current time and date. The almanac data contains information about every satellite, showing the orbital information for all the satellites in the system. The identification code indentifies which satellite is transmitting information.

9.5.2 Cell phones and GPS

After September 11 2001, the demand for GPS technology in cell phones increased. The U.S government pushed for implementing enhanced emergency calling which would show the location of the person in distress. There are basically two different ways of locating a cell phone. One is to use the towers and base stations which are arranged into a network of cells. A cell phone contains a low-power transmitter that allows it to communicate with the nearest tower. The base station then tracks your movement as you move from one cell to another by monitoring your phone's signal strength. So even without a GPS receiver, the location of the cell phone can be calculated based on its angle of approach to the cell towers, how long it takes the signal to travel to multiple towers and the strength of your signal when it reaches the towers. This is less accurate than GPS.

The cell phones with GPS usually have something called assisted GPS. This is a system which can improve the startup time of a satellite-based positioning system. It does this by using network resources to utilize the satellites faster as well as better in poor signal conditions. Signals bouncing off buildings, walls or trees are examples of poor signal

[20] How stuff works, Discovery - http://electronics.howstuffworks.com/gadgets/travel/gps2.htm
[21]NAV - http://www.navsoftware.com/info/how-gps-works/
[22]GPS Passion - http://www.gpspassion.com/hardware/explained.htm

conditions. This makes downloading the almanac and ephemeris data very difficult and time consuming because the receivers will only get fragmented signals.

Assisted GPS uses data available from a network to:

1. Quickly acquire satellites

The network can supply orbital (almanac) data for the satellites to the GPS receiver, which requires less transfer and quicker satellite locks. The network can also provide precise time.

2. Help calculate the position.

The server always has a good satellite signal and a lot more computation power than the cell phone. So it helps to calculate the position by comparing the fragmented signals it gets from the cell phone, with the satellite signal it receives directly.

The A-GPS also helps devices because the computational power required by the GPS device is reduced due to the fact that more calculations are done on the assistance server.

Cell phones also have the options to only use standalone GPS.

10. References

The sources used for the general knowledge of the underlying technologies are mentioned as footnotes throughout the paper.

Android

- http://developer.android.com/

BusTUC

- http://www.idi.ntnu.no/~tagore/rapporter/bustuc.pdf
- Discussions with Tore Amble

Real-time system

- Information provided by AtB

11. Figures

The first number indicates the chapter.