Danmarks Tekniske Universitet

Bachelor Thesis

A Traffic Guidance System

By:

Nikolaj Birch – s072777

and

Christian Thomsen – s072836

IMM-B.Eng-2010-80

December 20th 2011

Abstract

Traffic guidance is a major problem in the modern society. Traffic should be guided through city centers and other areas without queues forming on the roads.

In this project a traffic guidance system is developed that gathers traffic data from the Android-based smartphones of car drivers, and use this to direct them away from heavily congested roads. It achieves this, by combining a more conventional GPS-based navigation app with positional feedback from the smartphone.

The system consists of a internet based server, which handles the pathfinding and traffic control, using the A* algorithm and map data from OpenStreetMap, and an android app that guides the user and provide the feedback to the server. Also, a dummy client simulator is developed as well as a visualization tool, that simplifies testing and demonstrates the system functionality.

We succeeded in developing a system that can guide car drivers along the fastest routes and re-route if necessary when traffic jams are forming.

Preface

This thesis was made at the Department of Informatics and Mathematical Modeling at the Technical University of Denmark. It is the joint project of Nikolaj Birch and Christian Thomsen, supervised by Christian W. Probst.

Because this was a joint project, and the fact that we have designed, written and tested the code as a team, it is not labeled with author throughout. The main author of each section of the report is stated at the beginning of that section.

Basic knowledge of software development is required by the reader.

Signature:

Nikolaj Birch (s072777)

Christian Thomsen (s072836)

Table of Contents

1 Introduction....6
1.1 Problem specification....6
1.2 Structure of this report....7
2 Requirements Specification 8
2.1 Purpose....8
2.1.1 Server....9
2.1.2 Visualization:....9
2.1.3 Dummy client: 10
2.1.4 Smartphone Application....10
2.2 Functional requirements....11
2.2.1 Use cases:....11
2.3 Non-functional requirements....13
3 Project management....13
4 Traffic Control....14
5 OpenStreetMap....16
5.1.1 Map areas used during the project....16
5.1.2 Challenges....17
6 System Design....18
6.1 System Overview....18
6.2 Client server communication....22
6.3 Components of our system....24
6.3.1 Server-side....24
6.3.2 Client-side....39
7 Testing....48
7.1 System test....48
7.1.1 Black-box test....48
7.2 GPS test....50
7.2.1 Performance test....51
7.2.2 Stress test....53
7.3 Use-case test....53
7.4 Path-finding compared to krak.dk....54
7.5 Platform tests....54
8 Discussion....55
8.1 Navigation versus our system....55
8.2 Deficiencies....56
8.2.1 Data....57
8.2.2 Navigation....57
8.3 Future possibilities....58
8.3.1 Improved positioning....58
8.3.2 Reduction of carbon emissions....59
8.3.3 CO2 based navigation....59

8.3.4 Integration of public transportation....62
8.3.5 Driverless cars....63
8.3.6 TMC integration....63
9 Conclusion....64
10 Appendix....66
10.1 Appendix-1: User's manual....66
10.1.1 Server....66
10.1.2 Dummy client....67
10.1.3 Android Application:....69
10.2 Appendix-2: Timetable....70
10.3 Appendix-3: Changes in JmapViewer....71
10.4 Appendix-4: Test results....73
10.4.1 Black-box tests....73
10.4.2 Use-case test results....80
10.4.3 Pathfinding comparison test results....93
10.4.4 GPS-fix test results....96

1 Introduction

(Nikolaj and Christian)

Few things are more irritating than being stuck in traffic jams. Each year more and more cars drive the roads, and heavy congested roads are the curse of the infrastructure. Traffic jams and queues are unpredictable and difficult to avoid for the car drivers. In this project, we will demonstrate that queues can be avoided to a large degree, using the smartphones that are becoming ever more common these days.

A GPS receiver is a stable part of most smartphones and so is internet access. By tracking the driver's smartphone, information can be gathered about the current speed on the roads, and thus of the degree of congestion. The smartphone can then be used to guide other drivers around the queues, thus minimizing impact of congestion to those. This system requires no extra hardware in the car and only an internet based server to function – no roadside counters, cameras, tracking hardware or anything; just the app, running on the drivers smartphone.

In this project we have developed a prototype of this system, and demonstrated its capabilities and shortcomings. The system is a horizontal-type prototype, covering the entire functionality of the system to an largely equal degree.

We have used the openly available map data from the OpenStreetMap project, which is an open-source, community based effort to map the entire globe and make it available to the public.

1.1 Problem specification

(Nikolaj and Christian)

This project will involve the development of a system for traffic control, which gathers traffic data for car drivers. The system should guide the drivers fastest from point-A to point-B by avoiding queues. The fastest route should be calculated using the A* algorithm. Apart from this calculation, the information from the server about queues must also be considered. If a heavy queue is reported from one of the other units in the system, the routes that leads through this queue should possibly be updated to an alternate route.

The following components are expected to be part of the system:

- A program for clients that can communicate with the server to get the route from A to B, show the route and the current position, and then send information about this route to a server. After this, it should be capable of receiving corrections from the server about possibly

selecting an alternative route. A dedicated application for a smartphone will be developed, as well as a “dummy” unit that can simulate a client.

- The server consists of a detailed map and receives information from clients about client's positions and speeds. Based upon this information, queues are identified and possibly routes are updated. Furthermore, it must be able to receive information from clients, reporting about increased travel times on roads.
- A program to visualize the calculations from the server and the units, so data can be represented visually, it therefore must show where the units are on the map, and the roads that has queues on them. The purpose of the visualization is mainly to test the system's functionality.

1.2 Structure of this report

(Christian)

This report contains a number of sections that together explain our system, and the process of making it. In parallel to how we actually did the project work, we start by specifying and narrowing the requirements for the system and how we decided to work on the project. We will then describe and discuss the central concepts of traffic control and OpenStreetMap, and how we have used these during the development. We will also thoroughly describe the system, how it is constructed and why it is made in the way it is. A section about the testing and benchmarking we have done follows, as does a section that contains discussions about the finished system, as well as future capabilities and developments.

The appendix contains a full user's manual as well as test results and other information, not kept inside the main text.

The description of the system is split into the individual components. We have chosen to write about the entire process from designing each component to the finished implementation in one go. This also largely reflect our working order. We did not design and plan everything first before beginning the implementation, but made the process in a number of steps, each time deepening each component and adding to its completion.

We have made a lot of diagrams for this report. These are not the classic uml-type of diagrams, often used in software engineering, because these tends to be way more detailed and specialized than we need. The diagrams are made with much of the symbolism, but not the stricter conventions of the uml, to provide an overview instead of a full model.

Not all part of the code is described to an equal degree in this report. We have

prioritized the parts that we find most important, and left out most of the trivial parts, like listing all variables and getters and setters.

2 Requirements Specification

(Nikolaj)

2.1 Purpose

Based on the project specification, we will specify the project and make a solution strategy.

The project specification defines a traffic guidance system, which can be implemented in smartphones. The project specification is specified and in this section, we will analyze the specification, determine the technical aspects and both functional and non-functional requirements. This will help us in the Project planning, because it clear out the parts of the project with highest risk of failing. This allows us to set extra time to these tasks, minimizing the risk.

Based on the problem specification we chose to divide the system into 4 parts:

- Server
- Visualization
- Dummy client
- Smartphone Application

The subdivision of the system gives a simple overview of the system and what's needed to be done. This insures a more simple approach to the requirements specification.

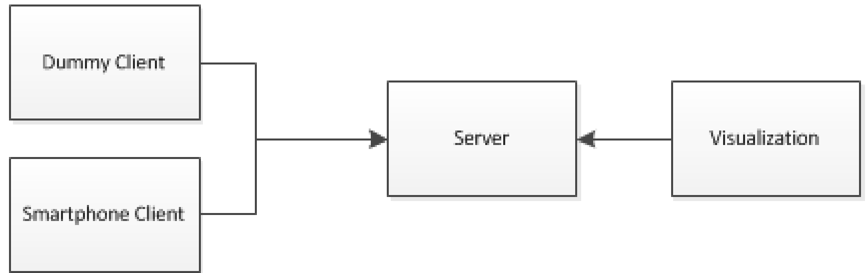


Figure 2.1: Simple overview of the system

The diagram in Figure 2.1 shows how the four components of our system should interact; with the server as a central unit, keeping track of all information from clients. The clients should communicate with the server trough Internet and the visualization should be implemented directly to the

server.

2.1.1 Server

The server should be the central part of the system; the server handles the path-finding based on the client's information. The server receives a start and end position from clients and should calculate the fastest route, based on the information. The calculation should be based on data from other clients on the road. If a client reports congestion on a way, the server should be able to determine the new transport time on the way. Furthermore it should be able to calculate new routes to clients which already has a route but is affected by the new congestion. The process of providing a client with a different route than the current one should happen without user interaction. This insures the user cannot overrule the system if he estimates the current route to be faster than the new route from server. The server will be designed and implemented as a prototype. Therefore we had to narrow the project down. Some aspects which are not implemented, are scalability to multiple computers and security. Our main focus is to develop a fully working server allowing users to obtain a route and receive new routes based on other user data. Although the scalability on a single computer, is in focus. Therefore we will analyze and implement algorithms and data-structures allowing as many users to use the system at the same time.

The map data the server is supposed to calculate routes from is the OpenStreetMap[1](OSM). This implementation of data from OSM should be parsed into the server from a XML file to objects in runtime.

The technology used for the server will be Java. Java is preferred because of the multiplatform support. Java has some good and common libraries which makes the implementation easier. Another pro for Java is that the creators of this software has been using Java for many years now and knows it well.

2.1.2 Visualization:

The visualization is used to generate an overview of the data, that the server transmits and receives. This data is mainly represented by the position of clients, and map data. The visualization of this will help us understand what happens with the clients. An example could be how the client's route looks like. The visualization will be implemented as a component of the server. It is supposed to run on the same computer and the same instance of the server implementation. The server and visualization is supposed to share GUI. This means that the information about clients connected from the server and the information about routes, junctions etc. from the visualization can be gathered

1 http://www.openstreetmap.org/

in one window.

The data the visualization shows is the map data from OSM with all details, ways, addresses and so on. Based on this data we will draw our routes, clients and junctions on the map. The data for drawing is information our server has, therefore we need an implementation which draws the changes in the data from server. This will end up with an event based visualization where the server informs the visualization when data are changed and tell what needs to be updated.

2.1.3 Dummy client:

The dummy client is supposed to work as our test client. The dummy's purpose is to simulate clients, this does not include a Navigation part like on the smartphone. The dummy client should connect on the same way as the actual smartphone application, this allows us to test the server the best way when the test clients works as the smartphone application. For the testing to be proper we need more that one client running – actually several hundred will be preferable for testing our server. Therefore it should be possible to connect many clients from same computer, preferable a kind of automation, which allows easy control of clients. Controlling the start point and destination are necessary for the dummy client, therefore we should be able to manually set in coordinates for start location and address for destination.

The dummy client need some kind of simulation, simulating the cars to run along the route. Therefore a simulation, which allows us to manipulate with the speed of clients. This way we can make new congestions and in this way simulate congestions.

The dummy client will like the server be developed in Java, but different from the Visualization, which will be implemented in the server. The dummy will have its own main class.

2.1.4 Smartphone Application

The smartphone application will be developed in Android, on a HTC Desire phone. The reason for the choice of android platform is that it can be programmed in Java, which our dummy client also being programmed in. This allows us to reuse some of the code. Furthermore the Android platform seemed easier to work with, partly because of Eclipse IDE, which we are familiar with.

The Android application will contain a way to type-in the destination address, and receive a route from the server. Then it shall display the route for the user and if the user confirms, start the navigation. The communication between server and client should be designed so that the communication happens in the background. This also allows the system to send new routes to client without

human interaction.

The navigation is supposed to guide the user from start position to destination. This will be implemented using arrows that will shows in which direction the user should drive. Furthermore the navigation unit will automatically change route if a new route from server is received.

2.2 Functional requirements

The functional requirements are the important basic requirements of our system being able to meet the project definition. The requirements is seen as a Use-Case Diagram in Figure 2.2. The reason we use use-cases as the basic functional requirements is because we have taken a user-centered development approach.

2.2.1 Use cases:

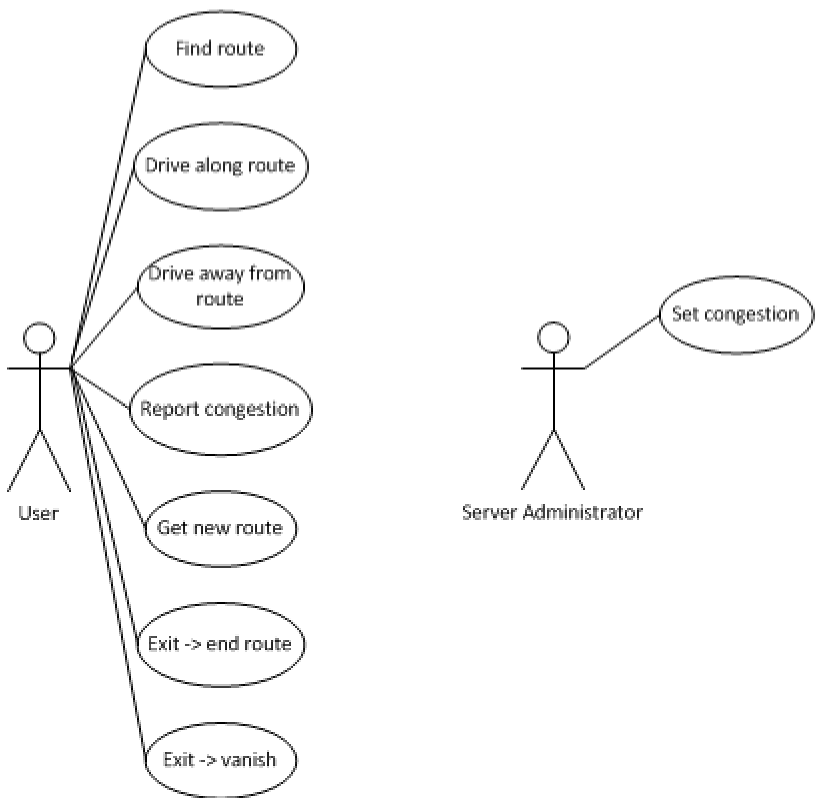


Figure 2.2: Use case diagram

The Use cases for the client is as stated above:

- Get Route:

The User should be able to Request a new route, the client should send the request to server and afterwards receive the route and start guiding the user to his destination.

- Drive Along route:

  After receiving the route the smartphone should guide the user all the way to the destination. The guidance should show which way to go next and what the name of the street is.

- Drive Away from route:

  The smartphone should be able to guide the user back on track. If the user leaves the route the smartphone should view the distance from the checkpoint he was supposed to reach.

- Report Congestion

  When driving along the route the smartphone should be able to report to the server when congestion is found. The smartphone will not calculate the congestion. The phone will send a update to the server each time the user reaches a checkpoint.

- Get New Route

  When the user drives along his route, another user can report a congestion, which may effects him, if the congestion is within his path. The server should calculate a new fastest route and send it to the client. Afterwards the client should guide the user to the destination with the new route.

- Exit -> end route

  If the user chooses to exit the program while he is on the route the client should disconnect and the server removes the client from the map.

- Exit -> Vanish

  If the system crashes, lack of Internet connection, no GPS signal or so. The system should remove clients when they have been inactive for an amount of time.

Server Administrator:

- Set Congestion

  The server administrator should be able to set congestions directly on ways.

2.3 Non-functional requirements

These define how the system is supposed to be, rather than how it does things.

- Speed, since it needs to support live queue-information, it needs to respond in near real-time.
- It needs to be portable, which means that most android-based smartphones, with network access and GPS, should be able to report queues to the server.
- All the reporting from client to server should happen without user interaction. Also the new route from server should happen without user interaction.
- The android app should be easy to use. No need for user manuals and instructions required should be minimal.
- Extendability and portability. The capabilities of the system should be easy to expand, or port to other platforms.

3 Project management

(Nikolaj)

This section deals with the project management of our project. We decided to make some milestones we could follow:

1. Product that works minimal.
2. Product that are working and fulfill the requirements.
3. Product that fulfills and works optimal.
4. Product and Documentation done.

These milestones are considered as iteration, this means an agile[2] development method is used. In each milestone we revisit the design and implementation, correspond to changes and new requirements for the product. Furthermore we have made a Gantt-scheme[3] (Figure 3.1), based upon the time-table in Appendix-2: Timetable which shows how much time we got to each task. As said in Section-2: Requirements Specification we divided the project into four parts to give a better overview of the project.

The Gantt chart is used to present the time schedule of the project. The chart normal contains the main components of the project, and the parts that is most

2 http://agilemanifesto.org/

3 Book: Operations Management, Russel & Taylor Page 364-367

critical to the project to finish. This chart is based on our requirements and our previous experience from software project on DTU.

The chart shows days scheduled to each part, but when more parts are scheduled at the same day, the work of the day is shared between the various tasks.

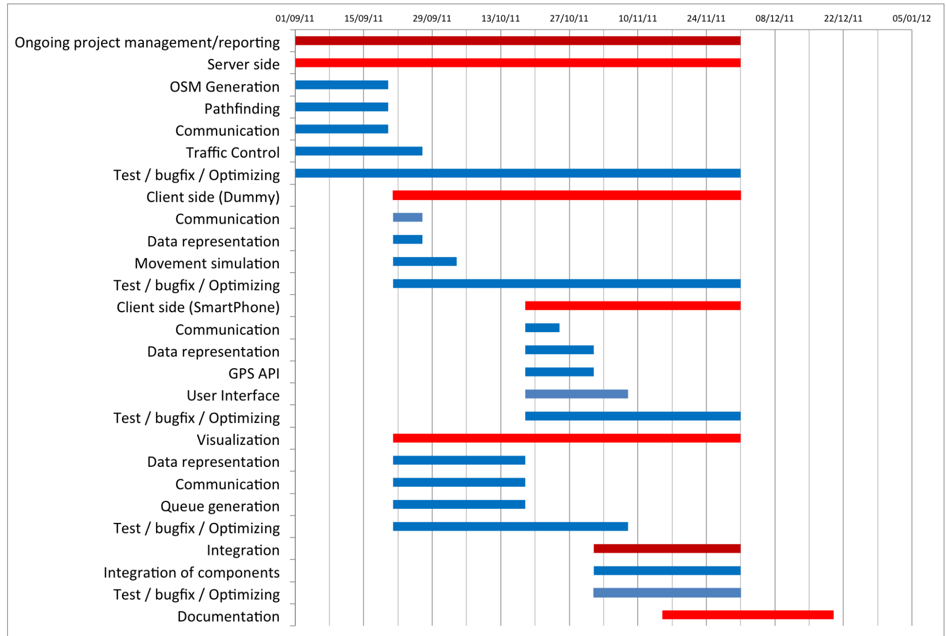


Figure 3.1: Gantt-Chart: y-axis Activity, x-axis amount of days of activity.

Our activities are not directly dependent at each other but its easier to develop the visualization after the server component has passed the first iteration. We chose to begin with the server, this part is the most important part of our system, and the most time demanding. Therefore we see this component to be the part with highest risk of demand more time than scheduled. There is much time scheduled to the integration of the components, this is important because of our iterative approach. The iterative approach means that even though we have developed all the sub-parts of a part, the part cannot be considered done. The part may be revisited in a later iteration and redesigned making our product smoother.

4 Traffic Control

(Nikolaj)

The traffic control aspect is important to understand how the system is going to

be designed. This section will explain the reason for our design choice and how we could have done.

When dealing with congestions there is several ways to calculate the queue. We chose a time based way, in this way we calculate the expected time between two points, this is done with the data from the map, supplying the speed limit on the way. The two points coordinates from the map data is used to calculate the distance between them. This gives us the expected time to travel on this way between two points. The client will report a timestamp each time it reaches a checkpoint or a node, which are the data from our map. After this implementation we consulted a traffic engineering student at DTU, he told us about how professionals observes queues[4]. The other way is to look at the way's capacity, this means how many cars pr. Hour the way can contain before it's a queue. If we take an example of a normal way with 1 lane in each direction the capacity of the way is approximately 1700 cars pr. Hour. If we had decided to implement this method we could have settled with the clients only reporting back their position. This way our server design had contained a counter for each way, when a client reported it to be on a checkpoint we could look up which way and set one more client at this way. The pros in the procedure would be the opportunity to caught the queues before they appear, in our way one client have to be in a queue before we can tell the rest of the clients that a queue has appeared. In this we could implement an algorithm which calculates a factor of how fast the cars on the road increases, in this way we could predict queues before they happens and guide clients around. The cons with this approach would be that every single car should use our system if not, queues will appear without our system noticing. In our way everyone need our system, but the more the merrier.

In junctions we have some problems because, you cannot drive through each junction with the speed as the speed limit says. Therefore we had to insert some delays when facing a junction. If there are two ways out from the users position, we added a delay of two seconds, this is based on our own evaluation of the time spend in junction in average. are there three or more ways out from a node we added a ten seconds delay. As before it is based on assumptions of time spend. These delays can give some problems. If its small ways that cross, it may not take ten seconds to get past them. But if it's two big roads crossing the time delay may be bigger than ten seconds. The map data from OSM supplies us with traffic lights, but because of the OSM is open-source, its not always the traffic light is indicated. Therefore its not possible to count on the data. But if the data was correct, we could implement a different delay in junctions with traffic lights.

4 http://vejregler.lovportaler.dk/ShowDoc.aspx?docId=vd-20101203131959405-full&q=kapacitet

5 OpenStreetMap

(Christian)

Our project uses map data from OpenStreetMap. Openstreetmap is an open-source mapping project. Adding and editing is a community effort, and as such data may come from anywhere. The project was launched in 2004, but only took off for real in 2007 with the initial map data collected by volunteers using hand held gps systems and manually entering the data. Later, many additional sources were added and this helped expanding the map greatly, to contain several hundreds of millions of entries. The open street map project is still however dependent on volunteers, even though they may be using aerial photography and satellite data in addition to their own experience and knowledge of the neighborhood.

The map uses the XML format (eXtensible Markup Language) for its data. An .osm file containing the xml data can be exported from openstreetmap.org for limited areas or downloaded for larger areas at a time from servers that extract the data from the entire map on daily basis or at other intervals. Alternatively, data can also be retrieved on a more specific basis, via http requests. We opted to work with with a pre-downloaded file, as this would provide us with the best insight in the possibilities of the data, and the easiest debugging, as well as not having to have an open connection to the openstreetmap database as a requirement for running our system.

The openstreetmap data is arranged into the three data primitives: nodes, ways and relations. Nodes represent points – road intersections, points along roads, addresses, shops etc. Ways are connected nodes; this may be two or more node, connected to form a linear feature – power lines, streams, roads, hiking trails and so on. Ways can also be bounded features such as parks, lakes, city blocks or coastline, in this case the string of nodes loops back on itself and form the area. Relations can be groups of nodes or ways and can denote things such as routes for bicycles or named motorway systems. Of our interest is mainly the ways that depict roads and nodes that depict addresses as well as those that are used to define the roads.

5.1.1 Map areas used during the project

We started out with a small area of the "Fuglebakken" area of northern Copenhagen. This was chosen because that area is generally made up of parallel roads, and right-angle intersections. This suggested that it would be relatively simple to work with and it proved to be right, although many of the challenges of a bigger map would also haunt this small one. Later on we moved to a bigger map, the largest to export directly as a sample from the data. This time we centered it on the familiar area around DTU in Lyngby – well not exactly centered, because half of the area would be the big Dyrehaven forest to the east of DTU, so the map has northern Lyngby, Brede, Virum and Nærum. This map was used throughout the development and testing phases, along side the biggest practical map we could use: one of the Greater Copenhagen area. This was more than half a gigabyte of xml data, so we could not read it with standard text-editors as we could the others, and it would take several minutes to parse the data each time we would have to debug something. So we used the smaller maps for development and debugging, and the big map for running and testing. These maps can be seen outlined in Figure 5.1.

Figure 5.1: Outlines of the three different map areas, we have used.

5.1.2 Challenges

Because the map is exported from a bigger collection of data, which has been cut at straight lines along the north-south and east-west direction, we are left with quite a few gremlins in the map. The most obvious is that along the edges, we have roads that leads to nowhere: ways that are in the map, but using nodes that are not. The exporter apparently does not take this into account, so we had to do that when reading and building our graph.

Another issue is that as our ways are directional, and some roads are one-way, some routes may begin or end at places that are impossible to reach. An example could be a motorway at the edge of the map. Motorways are

interpreted as two parallel roads, each of them one-way. Therefore, if you start somewhere after the last off-ramp on the out-going side, or have the goal at before the first on-ramp on the way in, it will not be possible to reach in our graph. Of course, motorway areas are not the most populated areas, and no addresses would be directly on the motorways themselves. Thus these scenarios are not very likely to happen under real conditions, but they were at times annoying during testing. Mainly the A* pathfinder were hit, because when it tries to reach one of these “black spots” and find that there is no direct route, it needs to root through most of the rest of the map to confirm that there is no indirect route either. There was not much we could do about this, except maybe do an iterative narrowing of the graph to ensure that no dead-ended ways were to leave the map, but then again, this would hurt the expandability of the graph, and make it harder to connect with a second area of the map, should we need that to happen. The issues would not harm the integrity or stability of our system, only the performance, as all that would happen would be a long-lasting calculation of the pathfinder that did not come up with a valid route. Therefore we chose to leave this in. A good example can be seen if one chooses a location near the Öresund bridge in the greater Copenhagen map. This has only two roads, and no connections until it gets to Sweden, which is not a part of the map.

There is a related issue but this time with entire areas that are unconnected with the rest of the map. Obvious examples would be islands which are not connected to the mainland by bridges. As our graph does not take into account ferry-lines, and some islands may not even have these, starting or ending at such places would of course yield no-route results. Other places that are more troublesome are mainly linked with the smaller maps, we have tried. The Fuglebakken map is crossed by a railway line that separates one part of the map from the other, the Brede map has the motorway and these renders smaller sections of the graphs unreachable from the rest, even though they both are valid parts of the map, and are actually connected in the real world.

6 System Design

(Nikolaj and Christian)

This chapter will present the design choices we have made. First we will introduce the combined structure of our system, and then go into of details it's individual components.

6.1 System Overview

(Nikolaj)

After we specified the requirements of the project we ended up with a design

containing a centralized server responsible for all clients seen in Figure 6.1

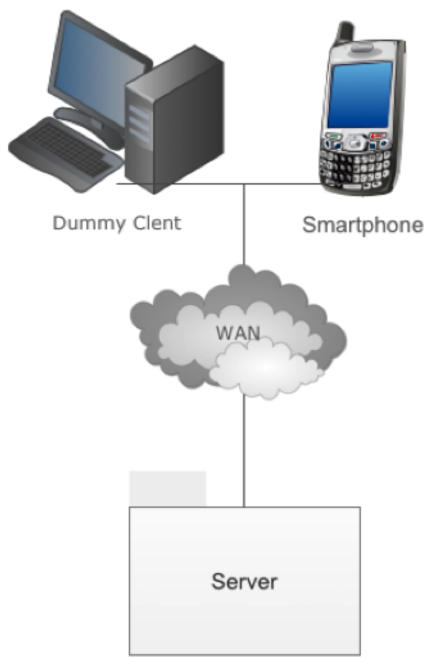


Figure 6.1:

The server is designed to handle all clients, their requests and their routes. Therefore no particular logic is placed in the clients, which ensures the system to be scalable when developing new clients for different platforms. The client's uses a "socket connection" connecting to the server. With the socket connection, standardization for the data was necessary. These design choices will be discussed in the server-client communication (section 6.2). The code for the dummy client and the smartphone client have been made as similar as possible, therefore the code can easily be used for both clients. Besides the communication the smartphone client also contains a navigation module. The dummy client do not include this since it's only for testing the server side and not the client. More about the client design later in this chapter.

The server design follows the Model-View-Control architecture, which gives the advantages of testing each component individually. The model part is the part containing the data, in our case stored in lists, its also here where we manipulate the data so it fits the requirements of our controller. The view part is the server-interface, our interface are combined between the visualization and server interface. It contains a map that provides information about routes, clients and congestions. It also contains a server GUI that supplies the

information about how many clients connected, shows if they send an update etc. The controller part provides the information to the view part, based on the data from the model part. Figure 6.2 shows how our system are split into Model-View-Control

The system follows the Model-View-Control architecture, Java Swing is used and therefore it is a 'Model-Delegator'-pattern, where the view and controller are combined.

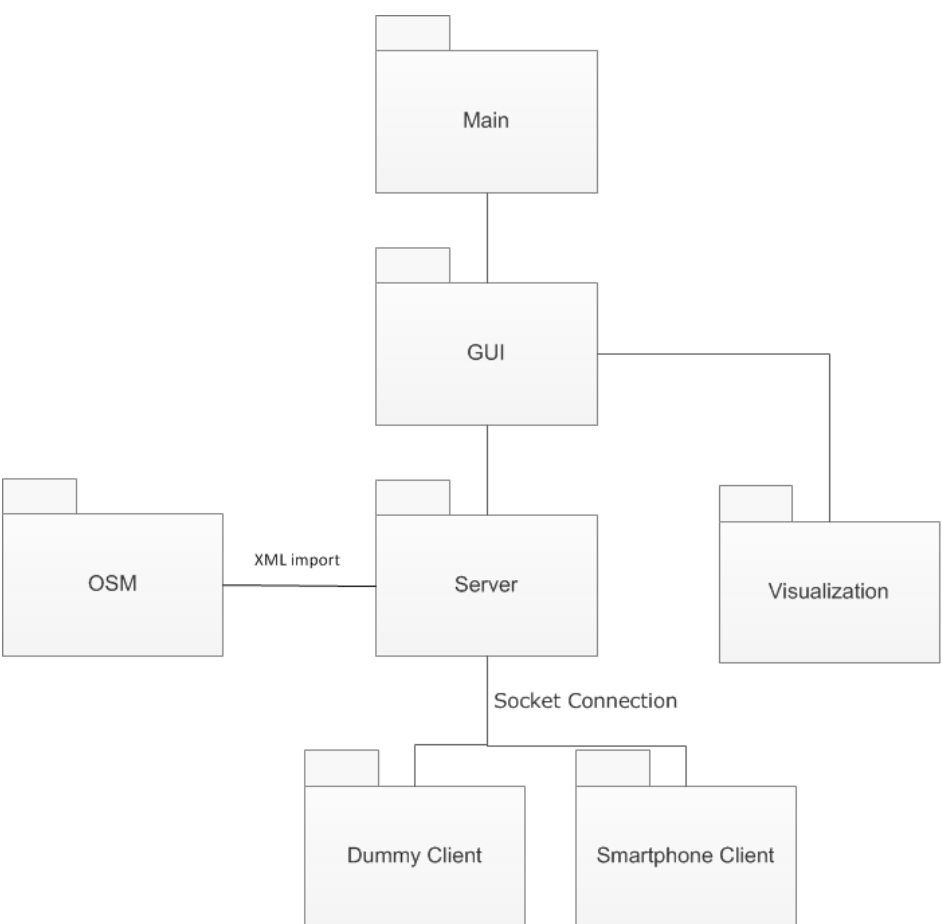


Figure 6.2: Package diagram of our implementation

The package diagram in Figure 6.2 shows how the components are implemented. The GUI implements both visualization- and server-GUI. This gives the Pattern stated above. The server package has the Controller and Model parts.

The model part consist the OSM-XML map which are read in to the system.

The XML data are the rare information we need. Our system reads-in the data and manipulate the data, putting them into data-structures. This ensures that the data we need is stored separately and ready to use for the controller part. Our controller part handles all the clients and based on information from them the controller extracts data from model and sends it to the GUI.

The overall design are as mentioned above the result of analyze stated above. Another approach may be to place more logic in the clients. This way the service would be more decentralized, this may have been done by coding an application, which requested the map each time, downloading the map data and displaying it. Just like the Google Map function works. But this would result in a lot of data traffic between the client and server. Another approach could be downloading the entire map to the smartphone. This would result in fast calculation of route, and only check the server for new congestion instead of getting the entire map. This way a limited amount of traffic between server and client are exchanged. This would cause a large amount of data placed on the smartphone and the need of from time to time update this map like we see in normal navigation for cars.

In the chosen approach the large amount of data is placed on the server, which is faster and have more memory than smartphones. In this way we use the advantage of the fast server to do calculations and contain the large map. The traffic between clients and server are also low. The only things, which are parsed, are strings and XML. The data is sent from clients when asking for a route and when they reach a waypoint. This means that the XML data is only sent one time for each route. And the string it sends at each waypoint only contains: Prefix, ID and timestamp (more about this in next section). This way the Internet traffic is brought to a minimum, and the resources are used in the best way.

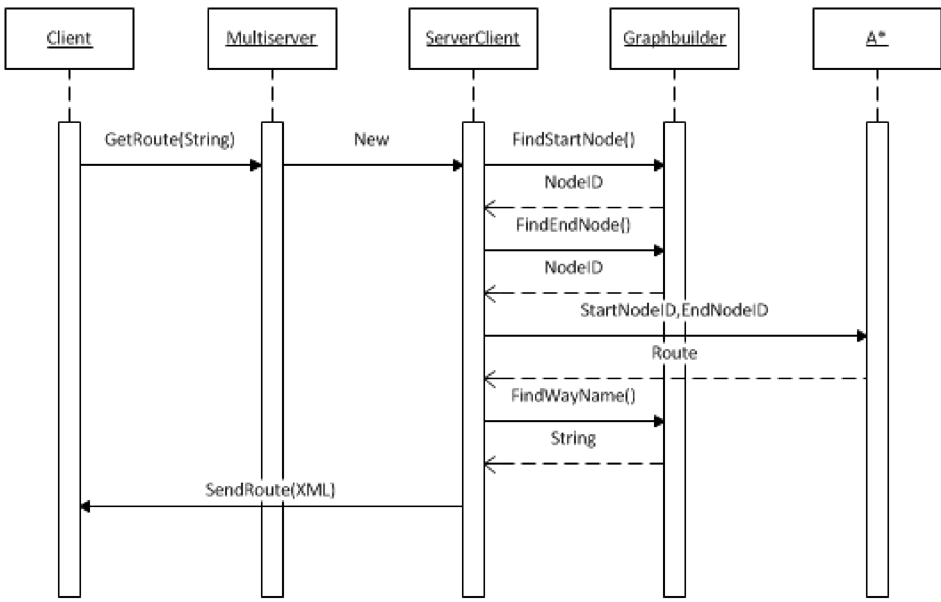


Figure 6.3: Sequence Diagram whole system

Figure 6.3 shows the flow of data in our system. This overview begins with a client requests a route that is send as a string to the *Multiserver*. The *Multiserver* splits the String up and creates a new Object: *ServerClient*. The server client finds the start- and end-nodeID(NodeID is nodes from the OSM map. We need to find the closest node the system knows from the user input) based on the information from *ServerClient.* Then it creates a new object; A* that it sends the Start and end nodeID ant then it calculates the route and returns it. Then the *ServerClient* sends the information as a string in XML format.

6.2 Client server communication

(Nikolaj)

The communication between client and server are as stated above socket connection, the technical aspect is discussed in the next section. Although a standard for the communication must be stated. There are two aspects of the communication, the XML code that returns the route and the three strings that are: Route request, Disconnect and Update position. The three strings are the communication from client to server, and the XML are the communication from server to client.

The composition of the strings is shown below:

The delimiter “%” is used to split up the string.

Route request:

prefix (1)%ID%Latitude%longitude%Postcode%City%Way%Housenumber

The prefix indicates which type of message the server receives: “1” is route request. The ID is a timestamp, this ensures a unique ID. Latitude and longitude are the current position of the client. Postcode, City, Way, House number are all part of the destination.

Disconnect:

prefix (2)%ID%Timestamp

The prefix indicates the message to be a disconnect message. The ID is needed for the server, knowing which client disconnects comes from. The last parameter is the timestamp, showing the time for disconnect.

Update Position:

prefix (3)%ID%Latitude%Longitude%Timestamp

Like before there is a prefix indicates an “Update position”. The latitude and longitude are the user’s position; this will always be a coordinate which the system knows as a checkpoint. The client calculates its distance from the checkpoint each time it gets a GPS fix. If it’s within a certain radius the client will send the request “update route” with the check point coordinates. The Timestamp are used to calculate, if there is any Queue between checkpoints.

The 3 strings insure the communication between the clients and the server. It’s only the clients that send these strings. The only thing the server sends are the XML containing the path. In Table 6.1 an example of the XML structure are shown.

Table 6.1: The XML structure of route

```
<path>
  <id></id>
  <node>
    <wayname> </wayname>
    <lat> </lat>
    <lon> </lon>
    <time></time>
  </node>
</path>
```

Table 6.1 shows the XML Structure, as mentioned it is the server that sends this out when a new route is calculated or a faster way for a client is discovered. The “path” element has the ID of the user, in this way the client ensures that it’s

a route designed for it. Each node represents each checkpoint on the route. The XML can contain any number of nodes, if no path are found it will only contain the destination node. A node contains a way name. This is used to display the way name, the user is supposed to follow on the smartphone. The lat and lon are the coordinate set that represents the checkpoint. This is used to calculate the distance from the clients position to the checkpoint, furthermore it is sent back to the server in the “Update position” call. The time is the time used int total up to this node, the last node contains the total time of the route, this is directly passed from the A* algorithm used in the server.

The XML structure could have been a string like our other strings but considering the big amount of information a path contains we decided to do it this way. The XML are actually parsed as a string to the smartphone, but instead of splitting the string up with delimiters as we choose in our client’s communication with the server. The advantages of XML are a simple robust format of our information. Robust because its based on a proven standard and can be tested and verified.[5] If the path should have been sent in a plain string only split by delimiters the string would be unnecessarily confusing and hard to implement in new platforms for clients. But in our 3 string methods it would be “overkill” to put it into a XML structure, but if we had decided to so. It would make our system more scalable if new features are implemented or more information from clients is needed.

6.3 Components of our system

6.3.1 Server-side

(Christian)

The server-side part of the system consists of a number of components. The communications server which handles networking, the map which is constructed from raw Open Street Map data, the A* pathfinder and the visualization and graphical user interface.

6.3.1.1 Communication

The communications server handles the network connections and manage the clients that are connected to the system. First of all, the server needs to be able to accommodate multiple clients. In our implementation, we settled on aiming for a few hundred clients at a time; at least the simulated clients. A real-world system would have larger capacity, hundreds of thousands perhaps, but that would require a very fast internet infrastructure and dedicated machines, much more powerful than our desktop and laptop computers.

5 http://www.w3schools.com/schema/schema_why.asp

The server therefore could be multithreaded, each thread handling all interactions with a single client and blocking the other threads when it is networking. Java however, has a non-blocking input/output API.[6] This makes it possible to multiplex the networking - to handle multiple connections in turn through a single entity.

We chose the non-blocking, multiplexed approach as this would allow us to avoid the usual problems connected with multithreading: deadlocks, thread safety etc., while benefiting from the infrastructure provided by the nio API.

Apart from managing the networking, our server would also have to keep track of all the client's data. Their route, their current positions and speeds, and use this data to determine whether roads are congested and how badly congested they are, as well as to alert clients about congested roads along their present routes, so they can get faster routes if possible.

We have designed our communications server as two classes: a server class (MultiServer) that manages connections and receives incoming data and a client class (ServerClient) that manages a single client's data and provides the sending of outgoing data to that particular client. Whenever a new client connects and request a route, a new ServerClient object is made and subsequent data and communications to this client are handled by that object. Incoming data are categorized and the appropriate action is taken by a number of processing methods.

The serverside classes and the structure is outlined in Figure 6.4.

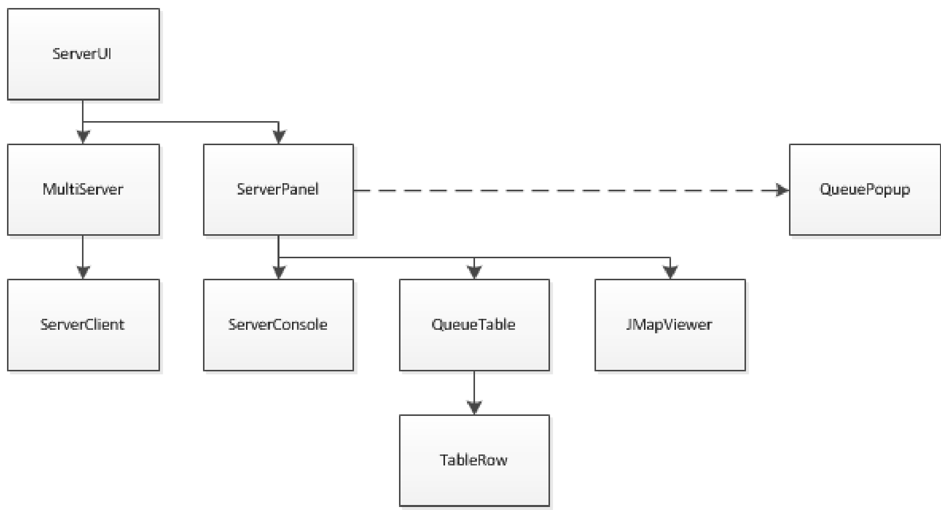


Figure 6.4: The structure of the server and ui

6 java.nio.*

ServerClient

The ServerClient class is the central data processing and outgoing communications object for each connected client. Because of the multiple data variables in this, there are a lot of setter and getter methods in this class, we will not describe these deeply, but concentrate on the more complex methods.

The first method invoked on a serverclient after the creation will usually be `makeRoute()`. This gets the right ids for the starting and ending nodes from the map, and then gets a route from a new Astar pathfinder object. It initializes the navigation variables and uses the helper method `sendRoute()` to build the string that is sent to the client and then send it. We chose to send the data as an xml-formatted string. This provides (reasonably) human-readable data as text, that is also easy to parse back on the client-side on various platforms if this should be necessary. The human-readability aids in debugging and later expansion of the system by other programmers. The xml string is wrapped in a bytebuffer and written to the serverclient's socketchannel.

`UpdatePosition()` contains the bread and butter of our traffic detection system. It is invoked from the server, when the client reports that it has reached a new route-point. It determines the route-nodes and way in question and calculates the time, the client has taken in traveling the road versus the time it should have taken according to the routing information. This is then used to check if the road is congested or if a queue has formed (the word queue is used throughout the code regardless). If the delay is larger than the fixed threshold of 110% of the normal time taken, then the road is seen as congested, and he map is updated with this information. The 110% percent was chosen to allow minor deviations: stopsigns, pedestrians crossing, cars in front parking and so on. If however, the client is faster than the current queue speed on the road, it would mean that the congestion has lifted. Either partly, in which case the new queue speed is set in the map, or fully, in which case the queue is cleared completely. If a queue was detected, the id of that road is returned, so the server can check with its other clients if it will affect them.

This is then done by invoking the `checkForQueue()` method. If this finds a queue on its present route, it will reset its navigation variables and make a new route.

MultiServer

This contains the multiplexed network server. The server relies on the java.nio concept of socketchannels to provide the communication channels and the selector to monitor the channel. A serversocketchannel is opened and the serversocket associated with the channel is bound to the port on which the server is run. The selector is then registered with the serversocketchannel. Both

the serversocketchannel and the selector is created via the static methods `open()`. This is used instead of a conventional constructor, and provides a platform-specific implementation and promotes the portability of the code.

The server class then progresses to its main run method. This is an infinite loop that start with the selector selecting. This gathers a set of selectionkeys that contains info about all the events that has been detected. This set is then cycled with the help of an iterator object and the type of event is determined. We only need to process incoming connections and incoming data events, so they are determined by AND-ing with a static bit-mask from the selectionkey class.

If a connection accept event is detected, the selector is registered to select read-events from this channel and a report is written to the server console.

If the event is a data read, the data is read into a buffer for processing. The socketchannel read operation can return a -1 if something is wrong, in which case we close the socket. If it really is a message as it should be, the data is interpreted as a string. Because we cannot be sure that a single read-event will not contain multiple messages from a client, the string is split at each newline character, and each processed in turn, be sure we catch all messages. The messages from a client is coded with an integer prefix. 1 means a request for a new route, 2 requesting to be disconnected or 3 a position update. Each type of message is processed in its own method which splits up and parses the data to fit its own needs.

Route requests are normally made as the first message from a client. They may be made subsequently if a new route is needed, but this is where we get the first exchange of data with a client. The server maintains a list of currently active clients, and this is checked to see if the client is already known or whether a new connection has been made. Then a new serverclient object is made and added to the list. Else, the existing serverclient has its starting position updated with the new position. In either case, the serverclient's `makeRoute()` is invoked and the result is reported to the server console.

Disconnect messages are rather simple: the active clients list is checked to confirm that the client is in fact there, and if so, it is removed.

Update position events are processed similarly. The data string is split and parsed, and the serverclient's `updateposition()` is invoked. This however, returns the id of a road if there has been detected a queue on it, so the server can notify other clients about it.

6.3.1.2 Map

The map is be the main basis of data for our system, used for pathfinding, locations queues and much more. The xml syntax of the OpenStreetmap

datafile uses a large number of different data types for describing locations, roads, relations and others, but the ones of primary concern to us are:

- <node> which are points in the map, some without any other information, and some which contains addresses.
- <way> with the additional highway-tag which connects strings of nodes into roads.

Our map would consist of something similar: a graph containing nodes and connected by edges, and addresses that could be the destinations for the users. In OpenStreetMap, roads are made from a string of nodes, which can be very long for example on a winding road, a large number of nodes will be needed to describe the curves of the road. For pathfinding, we are primarily interested in intersections between the roads, and roads that run a straight line between to road-intersections would be the optimum for the purpose of pathfinding and would have the smallest memory-requirement. We could achieve this by combining the non-intersection strings of nodes in a way of OpenStreetMap into a single edge, and get rid of all the nodes that were no longer needed. This simplified map, however would lead to a very rough visualization of the routes and queues on a map. As one of the main goals in this project was a visualization of what is going on in the traffic at any one time, we decided not to use this approach and just accept the increased pathfinding time and memory footprint of a less derived graph. Another argument against this sort of graph is that the routing has to be user friendly, it must direct the user as close as possible to their goal. A long road that only connects at its ends, would leave the user far from their goal at the end of their route if they wanted to go to an address around the middle of the road. It could be argued that a user should be able to find its way along a single road, but anyway – we have both seen this not being the case..

Instead, we have made the edges of our graph by splitting up the long roads into their individual small sections between the nodes. The graph must be directional because roads may be one-way or roundabouts may only allow the drivers to go in a single direction for example. We ended up with a scheme of a graph, made up of nodes, each containing a set of ways which has a reference to the node on which it ends. This would be the basis for the map data as used by our system. Addresses in OpenStreetMap are merely an extension of a primitive node, but with information about the postal address and often other informations as well. Thus an address-node located at “Byvej 5”, are usually not on the actual “Byvej” at all, but often alongside it. The only thing connecting the two, are the names of the road.

A multi-level map might be the best solution in the long run. We could have both the detailed graph with individual small sections of a road, and a less

detailed, intersection-to-intersection type of graph. The pathfinding could then be done along the detailed graph until an intersection was reached, then continue along the simplified graph until close to the goal, and finally go back to the detailed graph again to take the final steps towards the goal.

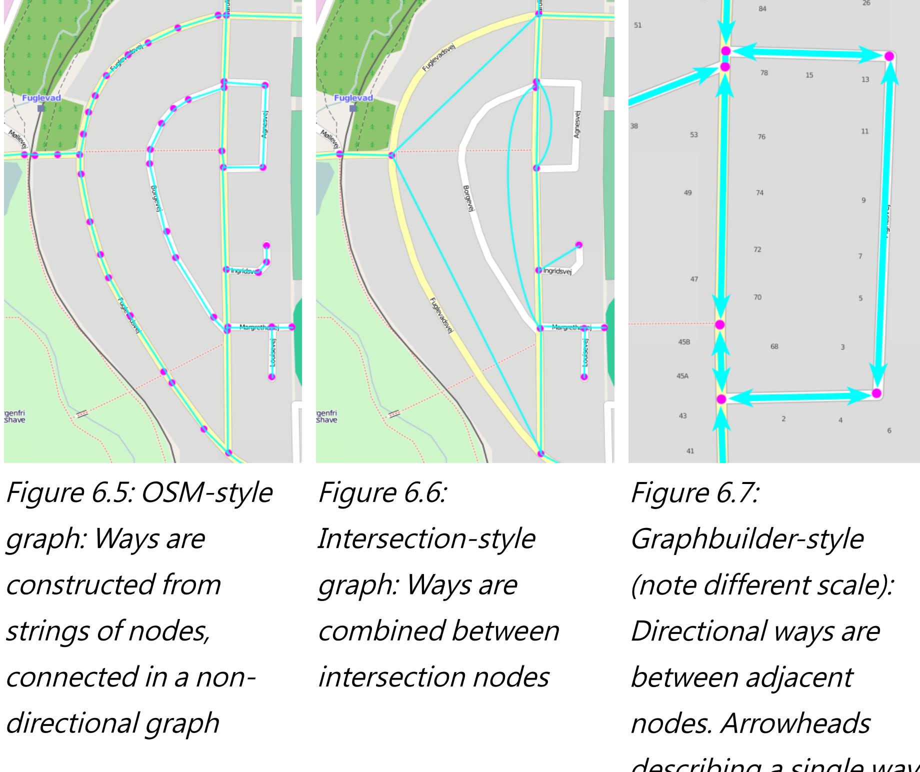


Figure 6.5: OSM-style graph: Ways are constructed from strings of nodes, connected in a non-directional graph

Figure 6.6: Intersection-style graph: Ways are combined between intersection nodes

Figure 6.7: Graphbuilder-style (note different scale): Directional ways are between adjacent nodes. Arrowheads describing a single way.

We made two separate classes for the graph: `MapNode` and `MapWay`. Addresses would be their own kind, as they would not be connected with the main graph as such, but merely provide a location. `MapAddress` does this. All three of them also implements Java's Comparable interface[7] that allows easy sorting of lists of these classes. The `compareTo()` method determines which object is the highest, when compared to another instance of the same class.

7 http://docs.oracle.com/javase/1.4.2/docs/api/java/lang/Comparable.html

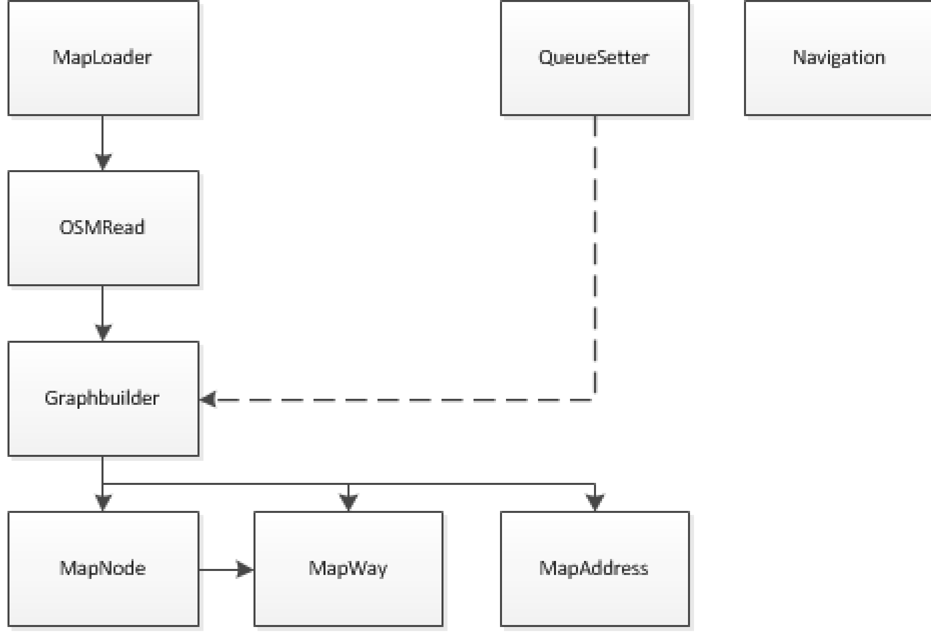


Figure 6.8: Structure of the map

MapNode

This contains the id-number and location of a point on the map, as well as a list of ids of all the mapways that leads out from this location. The compareTo method compares the mapnode's ids.

MapWay

Contains information and methods of a single section of a road: its speed-limit, the id of its starting and ending nodes, possibly the slow speed of a queue, and the name of the road it is part of, among others. A variable for a fixed time delay is also added, to take account of other slowing factors than speed-limits and road queues. This was added after some early trials had shown that we did not take into account that cars are slowed down when waiting at a traffic light, and that even in Hollywood, drivers to do not always go around corners on two wheels, tires screaming. A fixed delay seemed a good way to solve this issue, although some empirical trial-and-error would be needed to dial in on the appropriate size of this delay. CompareTo ranks mapways according to their starting nodes, and if they are equal, their ending nodes.

MapAddress

Has the location of a postal address.

These classes all contain a range of get-methods for their data and some setters

as well. Little would need to be changed after the creation of the nodes in the map (few houses move and few roads change names in this world...) but some of the data in a mapway would have to be editable; the queue-speed for example, and there are setter-methods for those. The CompareTo ranks these according to the name of the road, and then by the road number.

Graphbuilder

This class stores the full map, and contains the infrastructure to handle all this data. The main data are kept in three huge arraylists: nodelist, waylist and addresslist. There is also a list of the ways on which a queue has been detected. All of these gets built as the data are read from the xml-file from openstreetmap, and may be updated at run time. We needed to be able to randomly access these individual data as our system uses them for many purposes, so we chose ArrayLists as the collection of choice. Arraylist has constant or at least linear run times for most operations. The three data types would be referenced by either their indices in the lists, or by their name or the id, they have from openstreetmap. Thus we would have a number of access methods, taking different parameters to get the data needed. These are outlined in Table 6.2.

Table 6.2: Access methods are diverse. This lists the data types, and the parameters available to find them, as needed by our system.

Datatype	Parameters	Notes
MapWay	Index	General purpose access method
MapWay	Id	Search by OpenStreetMap Id
MapNode	Index	General purpose access method
MapNode	Address	Search for closest node
MapNode	Position	Search for closest node
MapWay	Index	General purpose access method
MapWay	Ids of start and endnodes	Search by OpenStreetMap Ids
MapWay	Name	Alphabetic search
MapAddress	Id	Search by OpenStreetMap Id

For the general purpose access methods, we use arraylist's standard get() method, but the others are more complex.

We need to know the indexes of a mapnode when we add mapways to the graph. A node in openstreetmap has a unique id, but we need to add the

mapway to the mapnode where it begins, and include a reference to the mapnode on which it ends. To find a mapnode's index in the list when knowing its openstreetmap id, we use a binary search. A binary search algorithm is the fastest way of finding an item, taking a “divide and conquer” approach that runs logarithmic time in the worst case scenario, O(log n) in big-o notation. However, it requires the data array to be sorted for it to work. In our implementation, we have included a flag that indicates if the arraylist has been sorted already, or if a sort must be done prior to doing the binary search. When adding a new node to the list, the flag `nodelistChanged` is set to true, and when the `getMapNodeIndex()` is called, it will first check this flag and sort the list if it is true. Usually, this will only have to be done once, as the map is built when starting the program, and the order is not changed subsequently. Thus we should get the full benefit from this fast search each time, but only face a single sorting, and also have a degree of failsafe, as the change flag is always checked. Both for sorting and searching, we use the java's Collections class, which has static methods for this purpose called “sort” and “binarysearch”, both requires the data to implement the comparable-interface which all of our nodes, ways and addresses does. The sort algorithm used is a modified mergesort which guarantees n log n performance.

We have to find the index of a mapnode closes to both a location and to an address when a client requests a route. The client supplies his own position and the address of where he wants to go, and we need to find the nodes that are closest to these two, for the pathfinder to make a route. If the location (latitude and longitude) is the input, we made it simple. The list of nodes is iterated, and the distance of each mapnode is calculated and compared to the smallest distance so far. When done, we will have found the node that is closest. The run time performance for such a search is linear O(n) and in our case, the performance is not the best, as there are a considerable number of nodes: 146.000 in the Greater Copenhagen map. It appeared to be the best solution when searching our arraylist, but other data-structures may have improved the performance.
Finding the node closest to an address works much the same. In fact we first find the addressnode, and then use the location of this as input to the method described above. The addressnode is found with a binary search for a wayname, but because there will be several addresses on each road, and the binary search returns as soon as it has found an addressnode with the correct name, we needed to expand the search algorithm. After finding a valid addressnode, it could be any house-number, so we start to count up through the addresslist until we find the correct house number, or if we reach the end of the road. If this does not result in a match, we count down through the list. House numbers in our addressnodes are in fact not numbers, but stored as strings. This is because they could be a combination of numbers and letters (52A for

example), some numbers might be missing, and other factors that might make a more systematic search difficult to design. Also a road usually has at most a couple of dozen house numbers, so this would not be critical to make a highly optimized search for house numbers.

When searching for a mapway, given the starting and ending mapnodes, we also use a binary search, but because the mapways and mapnodes are cross referenced, we can not just sort the list of ways at any time. Instead, we make a clone of the waylist and sort this, before searching. As in the other cases a flag is set to indicate when the list was changed, and so eliminates the need to sort the list when it is not necessary. But we need the index of the mapway in the original, unsorted list, so after finding the correct way, we go back and find its index in the waylist. This search for a mapway id is needed when a client reports its position and we want to check if there is a queue on the road. Because we have made the clients routes as a series of nodes, but not including the ways, connecting these nodes, we needed the ability to "go back" and get the ways. This is not in any way optimal, as this search is quite expensive in time, and could have been avoided to a large degree by including the ways in the handling of routing and communication.

Queuehandling

Handling queues is a very important part of our system. To keep track of all ways that has a queue on it, we made a list with references to those ways: queueways. When a queue is detected, we must set the mapway's queuespeed variable and add a reference to it in the queueways list. Similarly, when clearing a queue, the speed must be reset and it must be removed from the list. If it is already in the list, only the queuespeed must be changed. `SetQueue()` and `clearQueue()` takes care of these actions. We also have a bulk get-method that returns the entire list, as well as a list of the ids of all the ways in the queuelist. These are to be used by the visualization to get a list of all that needs to be drawn. The queues would be updated when a client drives down the ways, even faster or slower than the current speed, but because a very slow queue would lead to all clients being lead around the way by our pathfinder, we needed a some way to clear queues without a client reporting. We added a variable to all mapways that functions as a time stamp, being reset every time the queue-speed is changed. By comparing this time with the current time, it is possible to clear queues that are older than some delay. We have implemented this with a timed task, that clears queues that has not been changed in ten minutes. This ten minute timeout is a pure guess – it would necessary to measure or experiment with this in a real-world implementation. The more users in the system, the more likely it is that some of them would detect a queue dissolving, and thus clear the queue before the timeout. So if there is only a few users, the timeout delay would need to be bigger, and many users,

the timeout can be smaller.

MapLoader

The graphbuilder containing the map and infrastructure is built from openstreetmap data when the multiserver is started by invoking the static `load()` method in the class MapLoader. This will also save a complete graphbuilder object to disk to speed up future loading of the map. It will first try to load the graphbuilder object from a file, and if that does not succeed, it will read the xml from openstreetmap and save that to disk.

We use Java's objectinputstream[8] and objectoutputstream[9] to do the reading and writing, as this allows us to save and to load the entire object in one go. The only requirement is that the object must be serializable, and thus all of the classes used in a graphbuilder must implement java's serializable interface.

OSMRead

(Nikolaj)

The OSMRead class is used to import the XML-file containing the map. We use a SAX-parser[10] to import the file. The SAX-parser libraries are easy to use and effective when dealing with big files. We create a new `handler`[11] and start searching the file for strings we know. The search is based on start elements. It finds the next element in the text, therefore its not necessary to read the entire document into the heap-space. When the first element is called it calls the last element, this is used to give us the startelement (example: “Node “and the end element “/Node”). Between these elements we search for children, this is done by a simple equals statement. This way we get all the information we need and when the end-element appears we creates an object, in this case Node. This approach is used when we find: Ways, Way-Nodes, Address-Nodes. Ways consists of Way-Nodes.

The implementation is the one that works best. We began with a DOM parser. The problem with this approach was that it needed to read the entire file into the Java heap-space. The first couple of tries went well, but when the XML file containing the map increased it could not load the. The problem was the heap-space being filled before it could start processing it. Therefore we implemented the other way where it reads 1 line at a time without loading the entire file into the program.

---

8 http://docs.oracle.com/javase/1.4.2/docs/api/java/io/InputStream.html

9 http://docs.oracle.com/javase/1.4.2/docs/api/java/io/OutputStream.html

10 http://docs.oracle.com/javase/1.4.2/docs/api/javax/xml/parsers/SAXParser.html

11 http://docs.oracle.com/javase/1.4.2/docs/api/org/xml/sax/helpers/DefaultHandler.html

The OSM-read class is used as stated above to extract the information from the XML-file. Another important feature of this class is combining the data into objects and graph-structure to be read into the graphbuilder.

Navigation

(Christian)

Navigation is a utility class, meant to calculate distances inside the map. It takes inputs of latitude and longitude pairs, and uses these to calculate the distance. Alternately it can extract the latitude and longitude from a mapnode or an algorithmnode and use these, or it can be a couple of combinations between these. We added the combinations as we needed them. To be fully accurate, we would need the great-circle-distances, that is: the distance along the curvature of the earth. Searching the internet, we found a series of equations that should do just that, but somehow they were flawed. We then made our own distance-calculation. It is based upon the original definition of the meter, which was based upon the distance between the equator and the north pole, along a meridian through the city of Paris. This was then divided by 10 repeatably until a usable length was reached. Later on, the meter has been redefined several times, and since 1983, it has been based on the speed of light in vacuum. Never the less, the distance between the equator and the north pole is about 10 million meters, or 10 thousand kilometers as well as being equal to 90 degrees of latitude. Assuming that the earth is flat on the scale of our measurements, this makes for an easy conversion of degrees to kilometers in the north-south direction. If the earth was a complete sphere, this conversion factor would also work for longitude close to the equator, but not at higher (or lower) latitudes, because the distances between successive meridians narrows in, and becomes zero at the poles. We therefore multiply the longitude with the cosine of the latitude.

This calculation is an approximation alright, because the earth can not be both round and flat at the same time, and the fact that it is not actually spherical, but a so called “geoid”. Never the less, we measured some distances, and found only errors around 1% inside the area of our map of Greater Copenhagen, errors would tend to increase with increasing distance. This is fully acceptable, and errors in the length even tend to cancel out, because we use them to calculate differences in the length of routes. So as long as the errors are small, or at least consistent, we find no need to look for better approximations.

QueueSetter

QueueSetter is a small class, made to make it easy to add further control with the queues. Congestion may be detected by our our system, but only after some user has driven into it. Other congestion detection systems are already in use today, to help traffic control. Other external sources could include: changed

speed-limits due to roadside construction, accidents, planned sports events and the like.

Queues can be set from outside by creating an instance of QueueSetter and using it to set a queue by wayID or by wayname. We demonstrate this in the QueuePopup window that can be opened from the server UI.

6.3.1.3 Path-finding

Algorithms

Finding the fastest path through a map that consists of hundredth of thousands – even millions of nodes and edges, and doing so quickly requires an efficient algorithm. It is done using the A* algorithm (pronounced “A-star”), which is often used in video games and other applications requiring a fast way to do routing through a graph of nodes, connected by edges. Other options could have been Djikstra's algorithm which always finds the fastest route, but is computationally heavy on a large and complex graph, or a greedy best-first-search type of algorithm which would be very fast to run, but might not find the fastest route.

Djikstra's algorithm works as a broadening search and tries to keep the distance to the starting point (the cost of traveling) low, while a best-first algorithm would try to always aim at the lowest distance to the goal, disregarding other opportunities. A* is like a combination of these, keeping the cost low, while at the same time looking first for the lowest total cost. This saves it from having to look at lots of the irrelevant ways that djikstra would otherwise have to search, keeping the run time and memory footprint low.

A* pathfinding

A* uses a cost function and a distance-to-goal function added together to determine which nodes to visit next. The cost function called g(x) is the cost from the starting node to the current node, in our case where we want to find the fastest route, the cost is the time taken to reach the current node. The distance-to-goal function, h(x), in A* is a heuristic “educated guess” of the remaining cost to reach the goal from the current node. The h function must not overestimate the cost. Otherwise, the algorithm will tend to be closer to djikstra and little gain would be made. On the other hand, the h function must not underestimate the the cost too much, else the algorithm would be too greedy and we run the risk of finding a sub-optimal route. A straight-line distance to the goal seems to be the most often used h function. In our implementation it is a bit more complex; a straight line to the goal will be much faster if going on a motorway than if you are driving on a small residential road, and our nodes can have several types of roads leading from them. The best guess of the h cost function would be the straight-line time to the goal, when driving at somewhere

between the slowest and fastest speed limit. If we chose the fastest speed limit, the algorithm would “hope” to find a motorway starting at the next node, going directly to the goal, and thus any road that brings it closer to the goal would be preferred. This wouldn't be good, as actual motorways wouldn't get preferred. Almost the opposite is true if the other extreme is chosen: it would have to search too many opportunities because of “fear” that the next road might just be a bumpy living street.

Our implementation

We have chosen to set the h function to be the straight line time to the goal, given the speed limit of the fastest type of road from the current node. So in the event of an intersection of roads, the motorway would be tried first, in preference over the residential road, all things else being equal. This is calculated as the distance between the current node and the goal node, divided by the speed of the fastest road from this node, taking into regard that the speed might be lowered by heavy traffic.

Our A* pathfinder need to create its own partial graph consisting of nodes and edges between these. To satisfy the needs of the algorithm and at the same time cut down on memory usage, we have made a separate set of classes for these, instead of extending the classes of the big map. The are implemented as inner classes in the AStar class, and are called AlgorithmNode and AlgorithmWay similarly to the MapNode and MapWay classes of the big map, which also acts as arguments in the constructors.

The algorithm works by keeping a list of possible nodes to look at next – the “open” list, and a list of nodes that are considered visited – the “closed” list. The open list is is kept sorted by implementing it as a priority queue, ordered by the cost functions g and h as described above added together. This ensures that the most promising node in the path to the end-node is always at the head of the queue. In addition, a list of all the nodes that has been constructed is kept to keep track of all the nodes that has been made, as they are only constructed as they become needed.

At first, the open list contains only the start-node and the closed list is empty, because we have not investigated any nodes yet. Then a series of iterations are run until the end-node is found, that is: we have a connected set, all the way from the start to the end. In each iteration, the head of the open list is removed, investigated and added to the closed set. If the closed set contains all the nodes that has been investigated, it means that there is no longer any more possibilities to investigate, and we consider a route impossible and exit the path-finding there. To aid the infrastructure in the rest of our system, a “bogus route” is created, consisting of just the start- and the end-nodes. This will enable us to get some information as well as prevent unwanted artifacts to

appear. After these checks, we go trough all the ways out of the current node, which was previously at the head of the open set queue. Each one ends in another node, and if this node has not been made previously, we create it. If it is already in the closed list, it means that it has been investigated, and as such, we do not need to do anything further. Else, we calculate the results of the g and h functions. If the node is not in the open set (if we have not visited it before) it is added there. If it is, but has a higher g-score than the one we just calculated, we update the scores, because we have now found a faster way to get to that node than previously. Each node also has a reference to the parent node, the node through which it was reached. Once the iterations finish, we can use these references to travel back to the start, through what has now become a linked list, and thus we have our route. As the g-score for each node is the time taken to reach the node, we can find the total time to travel the route, by reading the g-score off the end-node.

The result is saved in a separate list of path nodes for retrieval by the system. They ways through which the nodes were reached is another thing that we are going to need in the further processing, and the path only contain the information about the order of nodes. Thus we made a method that returns an array with the ids of the ways that leads between the nodes along the path. In addition, we have made a route class for use by our ServerClients which is a lightweight data type class, consisting of and array-list of route-nodes, each with just the id, position and drive times, and their associated getters and setters.

6.3.1.4 Visualization

To visualize what is going on, and to enable the operator of the server some level of control, as well as facilitate the testing of the system, we have made a graphical user interface (gui) for the server side of our system. This is made up of three major sections: a console-like text output from the server. This displays the status of the server: who connects, who disconnects, do the pathfinder succeed in finding routes and so on. There is also a list of the queues that are currently detected, and allows the user to clear these manually. Buttons also enable clearing all queues in one go, and to add new ones. The third part is a large map. This displays data about all the clients, that are currently connected: their routes, including starting and ending points, and their current position. The map also shows all the current queues that are detected. Buttons enable the viewer to hide the queues and/or the client information.

The server console and the queue list are placed in a panel on the left side. The console is defined in its own class and consists of a scrollpane with a textarea inside it. A write method is made to append the incoming text and set the new caret position so the text will appear to automatically scroll down as new

messages arrive and are printed. A label below this displays the number of client that is currently connected. The table of queues is also in its own class with a scrollpane, but the contents are different. It is a standard jpanel which is updated regularly with a number of row objects, each representing a single piece of a road, a way, with the relevant information. The name of the road, the current driving speed of the congested traffic and the percentage of this, relative to the normal driving speed. There is also a button on a row that when clicked will clear the queue from the global map, and also remove the the row from the list. Below the table of queues is a button that clears all the queues on the map, and a button that opens a pop-up window that allows the user to manually enter a new queue. The pop-up has a number of textfields that allows the name of a road and the speed of the queue to be entered. This invokes a queuesetter that sets all ways of the road to have a queue of that speed. Alternatively, the id of a single way may be entered.

On the right side is a map. This was adopted from the OpenStreetmap java component “JMapViewer”.[12] This relies on internet based rendered tiles to display the map data. Various sources for these tiles are included in the demonstration project, but we decided to choose the one we found was best suited to our use. We needed a uncluttered view of all the roads, and not much more. Aerial photos are available as well as hiking maps and much, much more; both paid and free services, but we chose the the Mapnik tilesource as looking the most promising.

The full JMapViewer demo project was way too much for our needs, so we extracted only the classes we would need, and added a handful of others. The class JMapViewer is the central component that constructs the view from the tiles and adds some extra features like markers and control buttons, the rest being mainly infrastructure. Also, this class needed a few some changes to comply with our needs. Therefore we made some changes in JmapViewer.class, all of which are clearly marked in the source code. The changes and additions are listed in the table in appendix-2.

6.3.2 Client-side

(Nikolaj)

6.3.2.1 Dummyclient

The dummy client is used for simulating clients, instead of having a lot of smartphones we use dummy clients, which can act just like a client, but without any actually GPS locations. In this way we don’t need to have clients out on the street but instead simulating a user driving along a route, that makes the server act in the same way, and a lot of logic can be tested.

12 http://wiki.openstreetmap.org/wiki/JMapViewer

The "dummy client" (later referred to as client) needs to connect to the server using same procedure as a smartphone, which are trough "socket connection" in this way we can have multiple clients running on different computers. In our implementation we aimed for around 200 clients running at the same time for properly testing the server. Each client must be individually controlled; this means that the communication between client and server must be the same as with the Smartphone. Therefore the same procedure with client sending text strings and receives XML string is implemented. The simulation of a client driving must also be simulated properly using the "update route" call. Because of the lack of GPS unit, the client will simulate the route jumping from node to node in its path, being able to simulate traffic jams.

We chose to design the client as close to the real smartphone client as possible allowing reusing some of the code only needed to change few elements. Therefore are there two parts of this client; the actual client and the client handler.

The client needs to request a route, receive the route and process each step of the path, sending an "update position". Using real time simulation, meaning that we will be able to set the time it's supposed to use between nodes.

The client handler handles the dummy clients that are created, and provides a UI for setting start position of the client and destination. The handler keeps track of all clients created; this gives us the opportunity to create many clients in one instance of the program. The handler also controls the simulation of the clients, supplying the user with a list of active clients and the opportunity to change simulation time.

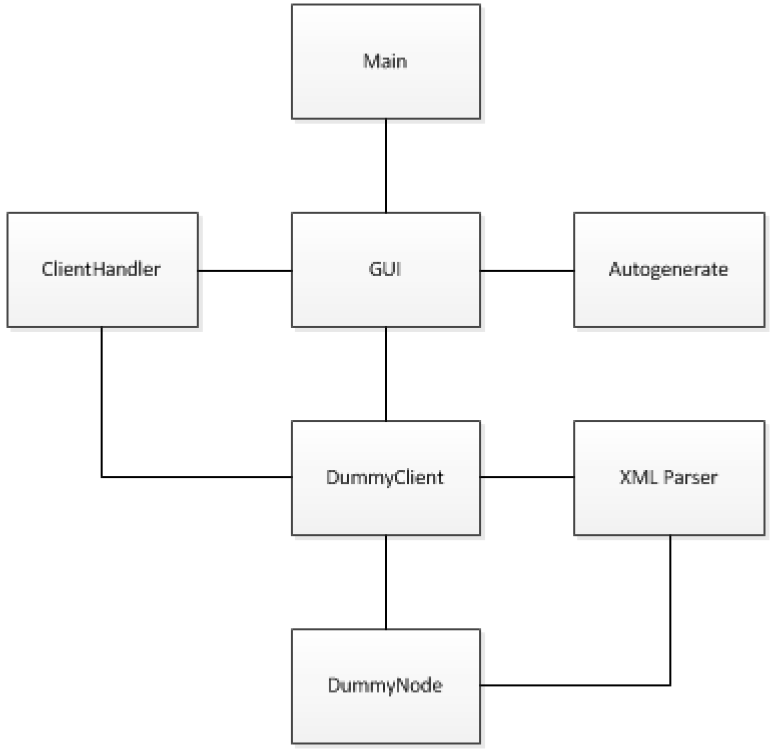


Figure 6.9: Overall design of the dummy client.

The GUI is used for creating new clients; in this term a new client is a new instance of the class DummyClient. The GUI has the AutoGenerate class which supplies the user with auto generated values based on the graph builder's map. This way it's easy to make a new client well knowing that the data it's based on data that exists in the map. The GUI then creates a DummyClient which has the data about start point, ID, destination. The DummyClient is inserted into a list in the Clienthandler.

The Clienthandler stores the list of clients, and contains the implementation of handling multiple clients. The list of clients is used by the GUI class to update each client, For example by disconnecting it. The communication between client and server happens directly; therefore the client goes into a state where it waits for input from the server. When a client gets a message from the server it receives the XML string and calls the XML parser class which handles the XML file and splits it up in DummyNodes which it puts into the client's route list. The route list is the list containing the path of the current route. The DummyNode contains: ID, Latitude, Longitude, Time and Way. As explained in the communication section this I the information we need for displaying the route on smartphones. In our dummy client it is not necessary to have the way

name. But if the client should be extended to actually navigate, it is implemented and ready to use.

The simulation of the client driving along the route are mainly handled in the ClientHandler, There are two possible ways to simulate the steps, either by setting a fixed time between each checkpoints, or just a run-method, which uses the time it is supposed to use between checkpoints and simulate real time driving. The Clienthandler goes through the list with clients and calls a method inside the class, which updates the position of the client. It's all based on timestamps, which it sends back to the server with the `Updateposition` method. Both simulation methods is based on the speed of the client, the speed can be set in the GUI where the user specify how much percentage of time you want the client to drive with. If a client sticks to the speed limitations it drives with 100%, if it drives half as fast as the limit it is set to 200%. This way of simulating gives a good simulation, which are very close to real driving. This way to simulate is the best implementation in our opinion, first we had a simulation, which was based on steps. When a client had to move from one node to another while crossing an intersection, we had to wait until we believed enough time was spent at that junction, for it to be passed. Another problem was driving in a queue. We didn't know how long time it should take driving on a way with a queue. Therefore the other way was implemented allowing us to do our simulation much smoother.

The design of the client was also redone. In the start, a single client was made. Doing that, required a lot of instances of the program be running at the same time. Therefore the ClientHandler was implemented and GUI extended with the DummyTable containing a list of Clients with the ability to change speed for each client. This extension allows us to test the system very well, and was necessary for our project. Furthermore the implementation of the communication between server and client has been changed. The first implementation involved a basic socket implementation where the ClientHandler was listening on a predefined port, and then it extracted the ID of the XML string and found which client it should pass the XML to. This implementation was problematic when multiple replies from the server was received, it was not able to read the buffer before the buffer was full and started replacing messages. Therefore a new solution was implemented, this time using channels in the port. Each DummyClient has its own channel assigned and when the server wants to communicate with the client it sends the string directly to the client. In this way the ClientHandler don't have to keep track of all incoming messages. And the problem with the buffer is almost non-existing. Theoretical the same problem could occur again but it would require a lot of messages to one single client are send at the same time. This is highly unlikely cause a lot of junctions should occur at the same time within the client's path.

6.3.2.2 Smartphone client

The Android client got the same basic functionality as the dummy client. The client needs to be able to write in an address, get a route from the server and, navigate the user to the destination. In the overview of the system we talked about our design choices regarding the more simplified clients and the more heavy server duties.

The Android client is therefore designed as simple as possible, this means that not much logic are used, the most this client actually does is the navigation part, where the other steps are exchanging data with the server and presenting them to the user. The navigation part is showing which direction you should go, way name and the distance to the checkpoint (This can either be a junction or just the row turning). Each new GPS fix will trigger the calculation of distance to next checkpoint.

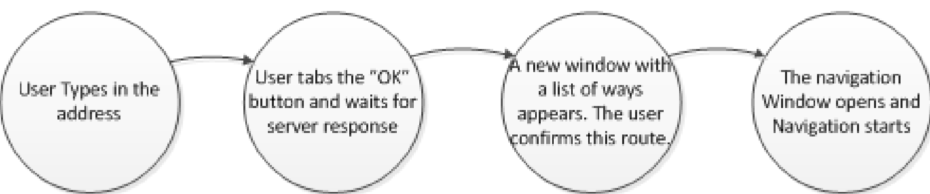


Figure 6.10: State diagram

Figure 6.10 shows the flow in our system from a user perspective. The user types in the destination, the client sends the data to the server. The server replies with a route, the client displays the route and allows the user to evaluate the route and then confirm it. The Navigation starts and when it receives the first GPS fix the first direction appears. Each new GPS fix the distance to next checkpoint is calculated, and if it’s within 8 meters, the client will send an “update position” with the new position, and then the distance to next checkpoint is calculated and updated on the view.

This process is based on our overall architecture decisions, it should be a very light weighted application, and on this prototype state, no fancy features are implemented

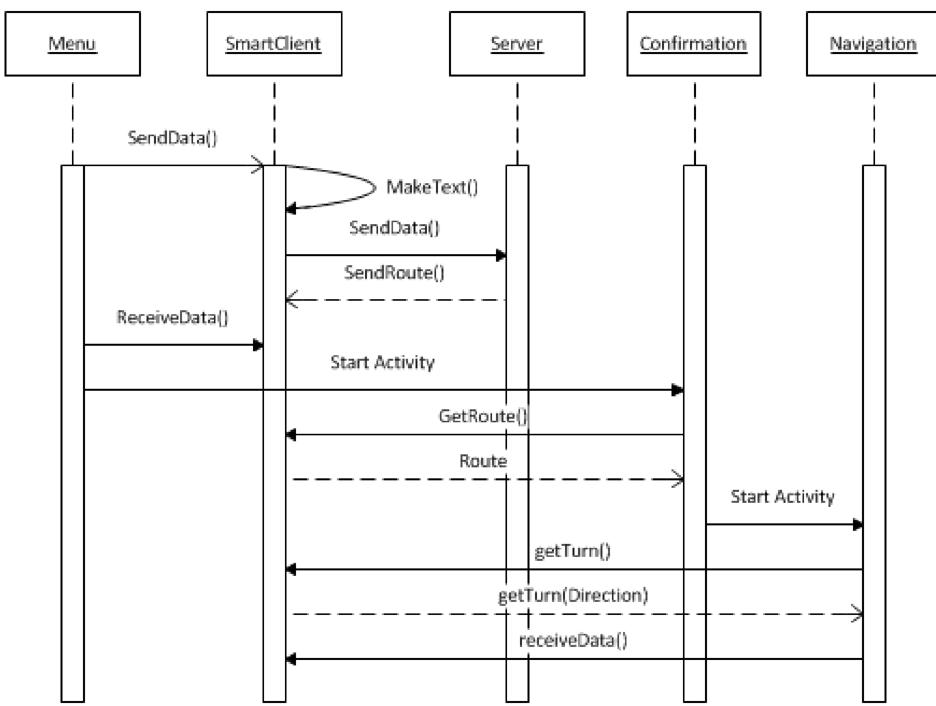


Figure 6.11: Sequence Diagram of getting a new route

Figure 6.11 shows the technical aspect of the flow in getting a route and navigating to the destination. The SmartClient is the heart in the android system; it holds the object for the data, e.g. destination address, ID, Route. This information are all stored in this object and therefore all the Activities; Menu, Confirmation and Navigation must get the data from it. The `receiveData()` is called from the Menu activity triggers the smartClient to read the buffer until a message from the server occurs then it parse the XML file and fills the *route* list with SmartNodes[13]. Afterward the call it starts the next activity: Confirmation. The Confirmation gets the list of SmartNodes and displays them to the user. This allows the user to verify the route before he begins the tour. If this is accepted he starts the navigation which is explained in Figure 6.12

13 Explained in the Class overview in Figure 6.13

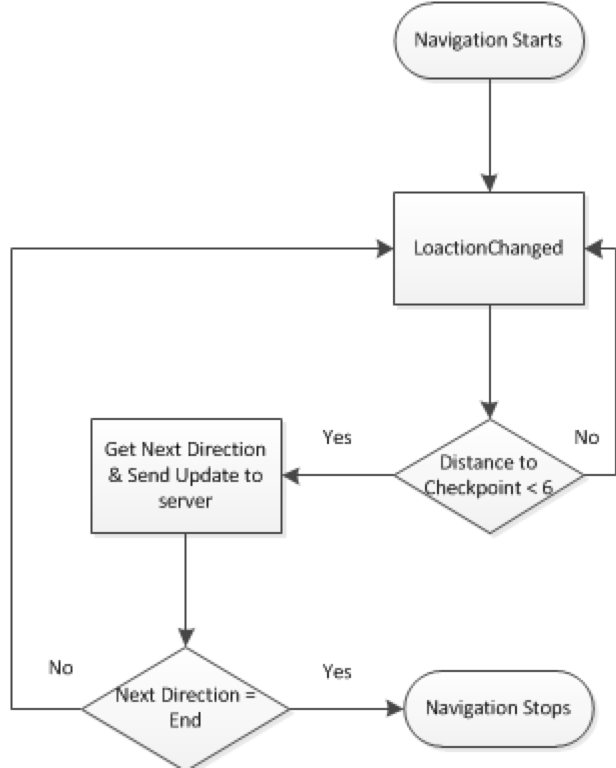


Figure 6.12: Flow chart of the navigation method.

The `Locationchanged()`[14] method is a Event which gives us the opportunity to obtain the latest Latitude and Longitude from the GPS. The method provides us with the functionality to decide how often the method should be called and how many meters the client should have moved before it counts as a `Locationschanged()`. We decided to make the client as sensitive as possible because of the importance of GPS data. The distance check is calculated in meters, this may give some errors, The GPS unit has indicated to be unstable when it comes to specific locations, although the buffer with 8 meters near the checkpoint should allow the Navigation to discover the checkpoint. When the distance is under 8 meters the Navigation calls the method `getTurn()` The get turn is used to determine which direction the user should go next. For that we need some vector calculations to determine this. It's not enough to now the coordinates of the next checkpoint, it is also necessary to know which way the user came from. We observe the ways as two unit vectors; this gives us the ability to calculate the cross product and hereby derive the direction.

14 http://developer.android.com/reference/android/location/LocationManager.html

When the direction is calculated, the Navigation activity changes the arrow picture according to the return value of `getTurn()`. The text field which shows the distance to checkpoint and way name are also updated. Then the `sendUpdate()` is called and sends the new position to the server.

In Figure 6.11 the Navigation calls the `receiveData()` this is the last thing that happens in the `locationChanged()` this gives an event based listening for new routes. This is necessary because of the server design, the server sends out a new route without any interaction if it finds a faster route that the current.

The application on the smartphone is based on activities; Activity[15] classes are an android class, which can interact with the user. This is done with the `setContentView(View)` method that allows to change the layout of the window. The layout is designed in a XML file which is referred to by the method.

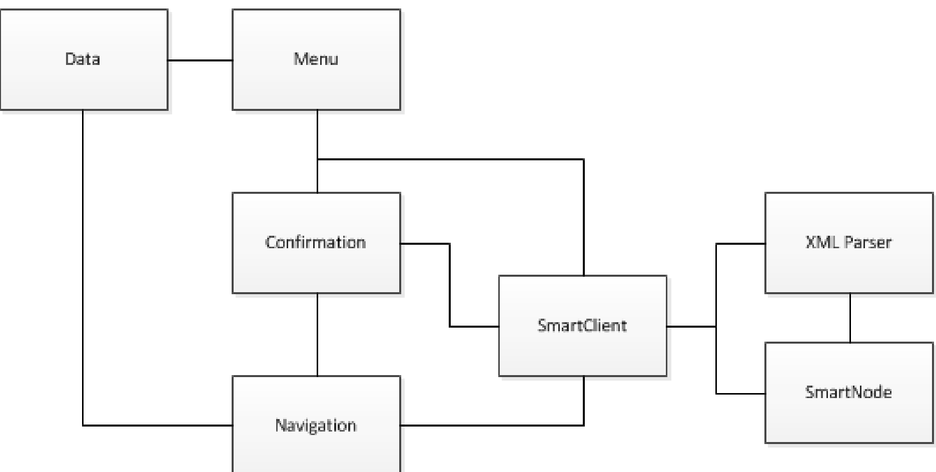


Figure 6.13: The classes of the Android Application

The Menu activity contains the main method, it's from here the program starts. The menu activity also contains the first view, where the user is presented to layout where the destination should be entered. The Clients ID is obtained from the Data class. Afterwards a SmartClient object is created containing the information about the client. When the address is send to the server, and the client have received the data (shown on Figure 6.11). The route is set in the SmartClient's the incoming message from the server is first parsed through the XML Parser and afterwards divided into SmartNodes. The Menu activity now starts a new activity; Confirmation. The confirmation class's only job is to show which ways the route is going through. This ensures that the user can

15 http://developer.android.com/reference/android/app/Activity.html

verify that it is indeed the right address (if he has been there before or knows which direction to go). If he confirms the third and last activity: Navigation is started. The Navigation activity contains the Navigation feature as explained in Figure 6.12. Navigation also makes a new Smartclient just like Confirmation. This is necessary because of our data is stored in this object. The data are in the two cases the need for our route data, stored in a list of smartNodes. This means that the SmartClient got static on all the variables allowing us to reach the data even though we create a new object. This is not a good design solution. The implementation should be revisited if we had some more time, it works because we only got one set of data in the client, and when the destination is reached we clear the static variables. If we had time for one more iteration we had passed the object between the activities. This could have been done with bundles and intents; this is a easy approach when dealing with simple data types, but when dealing with objects and list of objects it's much more difficult and time demanding. Another way to do it would be to only have one activity and then use the method changing the layout for each click. This would result in much smoother design containing a main class as a controller and a view and then the SmartClient as a model containing the data. The problem with this approach is a big controller class; the Navigation activity is easier to manage when it has its own activity. If we had 1 more iteration we would have kept to the plan. But extended the implementation with the data from SmartClient passed through activities.

The design of our `receiveData()` method which is event based on the `LocationChanged()`. A better way to implement it would be a Service[16] running in the background. A service is a way to tell the main class new things, which is exactly what we want to do, by running a service in a new thread. This service should listen on the incoming messages to the program, and when it receives a message it tells the Navigation activity right away. This would optimize the performance on the client. It is important to show the new route fast, cause of the risk of driving by a way where you should have turned. Another problem in our implementation is the ID, the ID is generated based on a timestamp which is most likely to be unique but not 100%. Another implementation would be with the smartphone IMEI number using TelephonyManager[17] The manager is a class that provides the developer with information about the device hardware and software.

The confirmation-activity's functionality is not fully implemented; in the

16 http://developer.android.com/reference/android/app/Service.html

17 http://developer.android.com/reference/android/telephony/TelephonyManager.html

prototype the confirmation only gives the user a view of ways on the route. If more time were added a Google Maps implementation would have been made. Such an implementation views the route on the map before the user accepts the route. The same implementation could have been done within the navigation method. Instead of showing only arrows it could show the map at the same time. This would present the user for a more transparent navigation.

7 Testing

7.1 System test

7.1.1 Black-box test

(Christian)

Testing the system turned out to be rather difficult when using a real-world roadmap. Unless you live in an American-style city in which the roads are laid out as a rectangular grid, it is nearly impossible to judge whether one route is shorter or faster than another route.

Black-box testing requires that the output can be predictable for a certain input, to evaluate whether the test returns the expected result. We cannot tell if the test was success or a fail if we do not know what to expect. To alleviate this, we have constructed a very simple map which we can use when testing the system. It was written by hand using the syntax of the OpenStreetMap xml files. This allowed the test to be conducted without changing anything apart from that map. It consists of three roads in the east-west orientation and three roads in the north-south orientation. The roads are laid out in a 3 by 3 grid which is 200 meters square, and is positioned within the area of DTU to provide a familiar setting. The roads are named “Avej”, “Bvej”, and so on, and each of the east-west roads has three addresses. These are offset slightly from the roads, as they would be in a real map. Because the test map does not reflect the real world data, the underlying map are displayed in the user interface, with our additional info on top, and this might be a bit confusing. Please refer to figure Figure 7.1 for a visual representation of the test map.

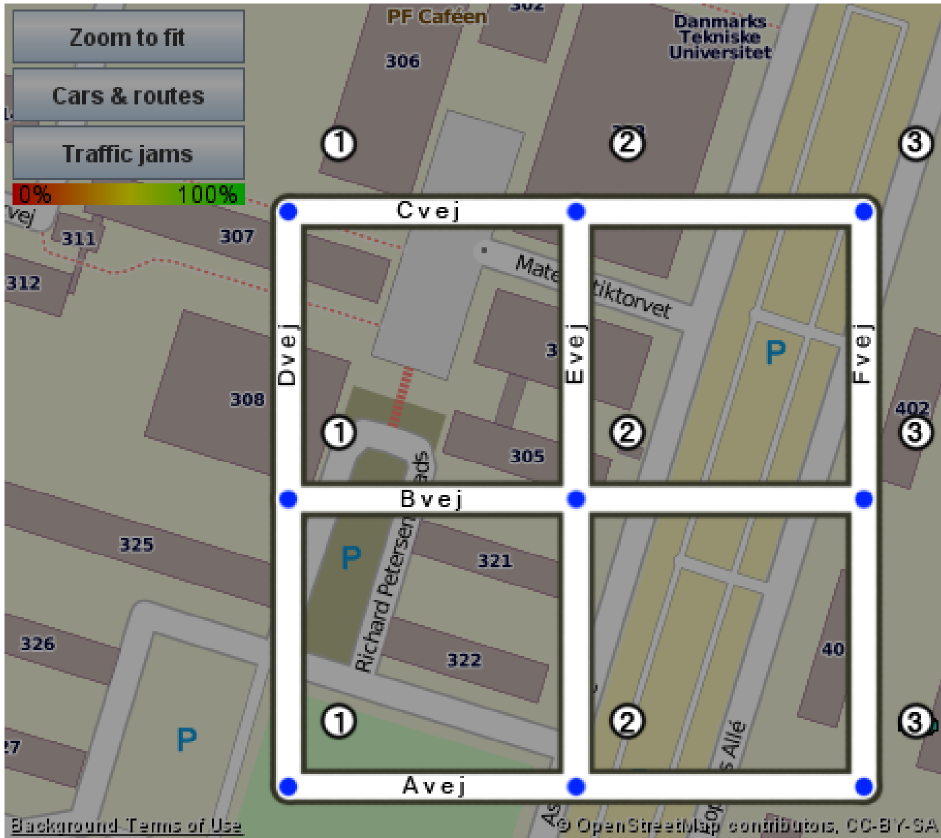


Figure 7.1: The map used in the black-box test. Roads are outlined, the mapnodes are blue dots and circles with numbers indicate the addresses.

Tests were designed to try out the various capabilities of our system: routing, detection of queues, re-routing of clients, removal and updating of known queues. Clients were added using our dummy client generator.

1. This tests the most basic functionality of our system: a client requesting a new route from its current position to an address.
2. This tests what happen if the client requests a route to its current position.
3. Tests whether the pathfinder avoids a congested road and takes another route. This is exactly the same as in test 1, but a queue is added on the route that was found during that test, suggesting that the fastest route is now different from the one found during test 1.
4. Same as in test 3, but with a new queue on the route found there. The

fastest route now, should be through the central 4-way intersection.

5. Tests if a client get re-routed around a queue, if this is added after the client starts driving.
6. Tests that a queue is reported and added to the graph, when a client move slowly along a road.
7. Tests that when a client passes a road which has a known queue, but at normal speed, the queue should be cleared.
8. Tests that when a client drives down a road which has a known queue, but at a speed different from the queue, but slower than the speed limit, the queue should be updated to the new speed.

The test results proved that in general the system works as it was planned, but the textual reports from the server when no route is found could be more precise. All test results are summarized in Appendix-4: Test results

7.2 GPS test

(Nikolaj)

This test is made to test the GPS unit in the smartphones. This test is performed with a HTC Desire, with our Navidroid application that is redesigned to count how many GPS-fixes it receives in 5 minutes.

A GPS-fix is when the `locationChanged()` method is called, when the method is called, it tries to get the fix of the location, it will not end before a fix is received. Its developed to send a location changed each second(assuming the GPS fix was available) and a change in the location in a meter. The one meter should be all the time because of the uncertainty of the GPS unit. Three test cases is examined: indoor, outside on a road in central Copenhagen and outdoor on a open field.

Table 7.1: GPS test results – see also GPS-fix test results in Appendix-4: Test results

Place	Fixes	Until first fix	Seconds per fix
Indoor	16	193 seconds	18 seconds
Small road in Copenhagen	91	86 seconds	15 seconds
Outdoor on a open field	151	11 seconds	1 seconds

This test shows that the GPS unit in a smartphone is not perfect to use. On a

open field it would receive a update each second when driving, this would be okay to use in a car driving. Inside Copenhagen where tall buildings interrupts the satellite signal it is only a single fix each 15 seconds. If you are driving on small roads 15 seconds could be okay, this test is performed standing still outside. Therefore the factor when driving and being inside a car may affect the result more. When used indoor its very hard for it to get a signal, which was expected because of the need of clear sight to the sky.

The test shows that the GPS unit in a smartphone still need a bit improvement to be very good as a car GPS. Unfortunately no data from normal car navigation was available. Therefore these assumptions are based on our own experience, including using a GPS-tracker app, while being passenger in a car[18]. But in smaller cities and on open road the Smartphone works just as good as a normal car navigation.

7.2.1 Performance test

(Christian)

This test was made to see the temporal performance of the system: how fast can it provide a route to a user and can it do so consistently? We made this test with the big greater Copenhagen map, and added dummy clients with the randomized functionality. 50 timed samples were made, and the time to return a route was logged. This is the time between hitting the submit button and the client receiving the route. No distinction was made between those returns that yielded a valid route, and those that returned a no-route-found message; the timing was the critical factor in this test. Also, the straight-line distance between the starting position and the goal was calculated and recorded to see if there could be any correlation between these, which could be used as a basis for predicting when a route should be ready for the client. The results are outlined in figures 7.2 and 7.3. Note that two outlying values of 408 and 410 seconds are not included in the figures, as that would make the details of the figures hard to make out.

18 http://www.sportstracklive.com/

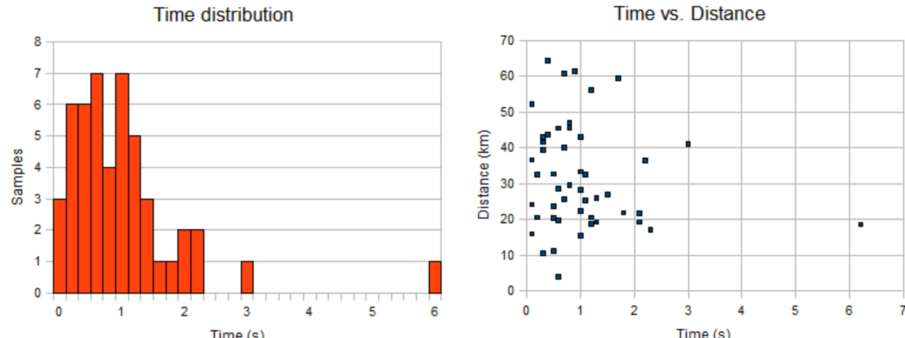


Figure 7.2: Histogram, showing the time distribution of route returns

Figure 7.3: The time plotted against the straight-line distance

From the histogram it is clear to see that almost all the measurements are below 2 seconds, with the median being 0.8 seconds. This is very satisfactory. Any user should have enough patience to wait 2 seconds. There are a few outliers: 3 and 6 seconds. These may be routes that are especially difficult to calculate, and the A* algorithm struggles to reach the goal, and never the less, are still acceptable. The two outlying values of 208 and 210 seconds are not. Neither of the cases return a valid route, and it appears that they had a goal that was located in one of the "black spots" in the map (See section 5.1.2 for details). The nearly identical times for these outliers, indicates that the pathfinder has had to travel the same a similar amount of the graph without finding a valid route, probably nearly the entire graph. This is done despite them being totally different: different starting and goal locations, and different distances of 11 and 51 kilometers.

An additional test was made by auto-generating 100 dummy-clients in one go. This took 1342 seconds, or 13.42 seconds per client. This is not acceptable, but looking closer at the results, revealed that 3 out of the 100 routings were the full-graph traversals as explained above. Subtracting these (409 seconds each) and calculating the average or the remaining 97 clients yields a values of 1.19 seconds per client on average. Much better, but we still need to be able to avoid those full-map traversals for the system to be really feasible. This may be done with a simple time-out function, so if a routing takes more than for example 10 seconds, the result could be assumed to be a no-route-found, and the pathfinding could be stopped. A better solution would be to tidy up the map data to ensure that there are no "black spots" or unconnected areas, making sure that no areas are left unconnected, and that all nodes and ways match up, which they do not do in the raw data from OpenStreetMap.

7.2.2 Stress test

(Christian)

This was to test how many clients, the system could handle at any one time. This is of course a matter of machine capability. More ram, faster processor, better internet will enable the system to handle more clients. This difference in performance has been very noticeable between the computers that we have used during the project. The laptops that has been the main workstations during development, start to slow down when adding as little as 100-200 clients simultaneously, but they can both handle the simulation with ease. A more capable stationary PC was able to keep up until 750 clients because of its better processor and ram figures. As with the laptops, it is the actual addition of clients and pathfinding that is the heavy part. Simulating and visualizing is not the limiting factor. If the system is to be used in a real setting, a dedicated server would help a lot and allow many more clients to use it. To expand the capability even further, several servers could be used, with some sort of load-sharing scheme that would have to be implemented.

7.3 Use-case test

(Nikolaj)

The use-case tests are conducted because of the importance of testing the most important functional requirements. These are based directly on the use cases developed in the requirements section. The actual tests can be found in Appendix-4: Test results (section 10.4.2) and in this sub-section we will discuss our results and show them if important.

Description of Use-case tests

1. The Find Route test is when a user want to get the route from A to B. The test failed one place when no internet connection are available, the try/catch statement does not catch the exception that is thrown. This results in a Crash of the application when a route is requested and there is no internet coverage.
2. The Drive along Route test is when the user drives along the route. All of the tests is passed.
3. Drive away from route test is when the users drives away from the route, then the application will not show the user back to the checkpoint but keep showing the next checkpoint. All tests is passed.
4. Report congestion test is when a car drives slower than the speed limit the server should set a congestion on the way the user is located. All of these tests are passed.

5. Get new route tests is when the user drives along the route and congestion appears on a way in its route. Then the server should calculate a new route and send it to the user. This test is passed.
6. Exit → end route test is when the user shuts down the application or the route ends, then the application should send a disconnect to the server and the server should remove the client from server. This test is passed.
7. Exit → client vanish test is when the application loses the Internet connection or battery power, then it should send a disconnect to the server. This test is partial failed, because of the android design the application closes before disconnect is send. But a feature on the server, which removes the inactive clients, is implemented. Therefore the client doesn't have to send disconnect.

7.4 Path-finding compared to krak.dk

(Nikolaj)

This test is a test of the difference between our product and a professional route description "krak.dk". The test scenario will be a short distance, and a long one. The short one is look at the route and see if they take the same ways. And the long to see if the calculated time matches.

As seen in Pathfinding comparison test results (section 10.4.3), path finding comparison the short routes are identical and with the same time to travel. The long route is almost same time only 4 minutes apart. They choose 2 different ways, both ways are possible and we were not able to see why there is a small deviation, it can be because of junction calculations, or speed – limits.

7.5 Platform tests

(Christian)

The goal here was to try the smartphone client program on as many different Android phones as we could get our hands on. We had two HTC Desires, we borrowed from DTU, Christian has the same model. In addition to this, friends and relatives with android phones were asked to help. The smartphone model, the installed version of Android and whether our app worked or not, as well as comments are presented in Table 8.1.

Table 7.2: Platform test results

Smartphone	Android version	Works	Comments
HTC Desire	2.2	Yes	
HTC Desire HD	2.3.5	Yes	Arrow graphics does not show
Sony Xperia X10 mini	2.1	No	Does not even install
HTC Sensation XL	2.3.5	Yes	

The reason that the app does not install on the Sony Xperia X10 Mini is probably that the android version is below the one specified in the source manifest file. This specifies which android api-level the app uses and ensures that it complies with the minimum capabilities of the handset. We may have been able to lower this level, but during the development phase, we had only access to HTC Desire phones, which has Android 2.2 and we could not have been sure that the app would be able to run on any lower api-levels without further testing. To make the app run on this phone may be as easy as changing the line in the manifest that specifies api-level to be "7", the one that represents Android 2.1.

The HTC Desire HD shows the direction arrows as the generic Android-logo icon. This handset (as the name implies) has a high-resolution screen, and the arrow graphics were made to fit a medium resolution screen. Providing a set of arrow graphics with a higher resolution should fix this easily.

8 Discussion

8.1 Navigation versus our system

(Nikolaj)

This sub-section will compare our type of navigation to the known professional navigation, which are used in cars.

The specification of a "normal" Navigation unit compared to our product is:

We used the most sold GPS unit in 2009/2010[19][20] The Garmin nuvi 265WT[21]

19 http://gpstracklog.com/2010/12/best-selling-gps-for-november-2010.html
20 http://reviews.cnet.com/8301-13746_7-10411699-48.html
21 https://buy.garmin.com/shop/shop.do?pID=13430&ra=true

Table 8.1: Select specifications for the Garmin navigator compared our system.

Feature	**Normal Navigation Unit**	**Our Project**
Internet	No	Yes
FM Receiver	Yes	No
Display size	4.3”	3.7”
Preloaded street map	Yes	No
CO_2 based route	Yes	Yes

The table above shows some selected specifications of the two products, its chosen based on giving an overview of the difference between these Navigation methods. The lack of internet in the Garmin compared to our system is compensated with the preloaded map and FM receiver, our product uses the internet to communicate with the map data, while the Garmin has loaded the map in internal memory. The Garmin unit has the advantages of not depending on a available internet connection. Our product also needs the Internet for receiving the TMC in XML-structure while the Garmin receives it through the FM-receiver. The Garmin screen is a bit bigger, this is the biggest advantage compared to our product. On a smartphone it sometimes can be hard to see where you should make the turn because of the smaller screen. This could be compensated with an integration, with voice telling the user where to go. Android has the text-to-speech functionality that could be used.[22]

Another big problem would be that our product is dependent on internet connection. But the fact that Internet gets more common, and that most of Denmark is covered with 3G, compensates for the lack of a map downloaded to the internal memory. Our advantage of not having the map preloaded makes changes to the map, easier to update. If a new road is built, we can just update the centralized server, meanwhile each Garmin unit needs to be connected to a computer, allowing the Garmin unit to update the map.

8.2 Deficiencies

(Nikolaj)

This sub-section will discuss the deficiencies of our project, this analyze will be based on our requirements specification, and what deficiencies we have

22 http://developer.android.com/reference/android/speech/tts/TextToSpeech.html

encountered.

8.2.1 Data

The implementation of our project is dependent on data. Data in this case is the information our system receives from clients, therefore we need a minimum of users of the system before it is usable. The way we implemented the traffic control, is the way you have to go, if we do not want to spend a lot of money on hardware, which can count cars at each way. In our implementation we do not need any hardware besides the servers. Therefore it is also important that the clients actually drive the route the navigation tells them. If they drive without the application turned on, the other users of the system cannot count on the data the system supplies. If we should give 100% information about how much time it takes to drive on each way, we could do two things. Make the smartphone to update the position each time its changed, in this way we know where all smartphones with the application installed are. This may give some moral and ethical problems because of surveillance of each user. Another approach could be installing hardware at all roads; this hardware could either count cars per hour or measure the speed of each car and send the data to the server. This way we rule out the user errors of forgetting to turn on the application, and start navigation even for short rides. The cons with this approach would be the price and time it takes to set all the hardware up to count cars. This is an expensive solution compared to the smartphone way where all data is supplied by the smartphone.

8.2.2 Navigation

Our navigation design may encounter some problem. The implementation only contains the navigation with arrows, way names and distance to the turn. Other GPS units show a map zoomed in to the position of the user. This way it’s clearer to the user which way they should turn. Our implementation with arrows works and gives the user the opportunity to follow the route, but it lacks on the user-experience. If we had some more time we would have improved the navigation with a drawn route on the already implemented Google Maps, which has an API for drawing. Our navigation unit is not capable to lead the user back to the last checkpoint if he is lost. Our solution will keep showing the wayname of the checkpoint and the distance to it. This way the user can see if he moves further away from the checkpoint or closer. But there is no arrows or an automatic recalculation of the route. If the user is lost he must start a new route, afterwards the server will send a new route to the user. Starting from his current position. An optimal implementation would keep track if the user is on the route, if he disappears from the route, the navigation should send a request to the server asking for a new route from the new position.

The map data we get from the OSM has some failures. We do not verify the quality of the data we get, but read it in to the system. A type of insurance that the data we get from the map actually works would be preferable. We have encountered some problems with the routes, because of the ways are not connected. This happens when we read the map into the system. Some ways are not connected but the data is still in the system. Therefore the algorithm sometimes has a long response time. A solution to this could be when the reading of the XML file, we make a path from each node, and if no nodes are able to connect then we remove the node from the data. This way the algorithm would always find a route. But it would take much longer time to read in the map if all these connections should be verified.

The data structure of our graphs could be improved. Right now the data lies in lists. If the data were put into some more high performance structures as trees, the performance would be better. Our solution is to all the data we got in graphs are sorted, this way we know exactly where the data is based and can make insertion and withdrawal fast. This is implemented with binary search as explained in section Graphbuilder on page 31.

8.3 Future possibilities

(Christian)

8.3.1 Improved positioning

One of our headaches was the GPS receivers in smartphones. The GPS signal is notoriously difficult to receive inside buildings, or in cities, where tall buildings blocks the view to the sky, and flat facades causes radio signals to bounce around, introducing errors and loss of positioning service. The lack of reception inside buildings, should not be an issue to our system, as we do not expect the users to be requiring directions for the parking spot inside their garages. Loss of reception can also be an issue inside tunnels, however, and that could be an issue. As the network service is probably also lacking inside tunnels, these would be double trouble. In open country, at least as open as a suburban areas, with only 1- or 2-story buildings, we have found no problems at all: first fixes arrive in seconds, and the resolution and precision has proven better than 10 meters in our tests.

In the very near future, new and promising ways of positioning are coming into use by the public. The European Union is developing its own satellite-based system, called Galileo. This is to have full control over the availability of the system. GPS is controlled by the United States Air Force, and as such, it can be turned off or encrypted at any time, the USA wants to limit the availability. The EU wanted better control, and also better accuracy, so started the Galileo

project which is very similar in structure to the GPS-system once it becomes active (planned for 2014-2019). China is developing a similar system also. All these systems, will improve the availability of a positioning system greatly. At least in times of calm and peace, the operators are planning to have signal available to public use that can have precision to within meter scale. A handset, or navigation aid in a car, that took into account all these systems, would have a much better chance of getting a valid fix, even among the tall buildings of a city. This is because the satellites are much more densely spaced in the sky, and a receiver on the ground would need a much smaller part of the sky visible to pick up the needed number of satellite signals.

Hybrid positioning systems are an emerging concept that could provide better positioning in urban areas.[23] These uses combinations of different technologies such as nearby wifi hotspots, mobile cell points and others to provide a positioning that can argument or replace the GPS-based systems.

8.3.2 Reduction of carbon emissions

The environment could get a lot of benefit from our system. Traffic is one of the largest contributors to CO_2-emissions (86% in Copenhagen in summer, 39% in winter[24]) Anything that can decrease the emissions from traffic may be an important factor in the present and future. Our system can not make sure to reduce the time it takes to drive and thus the amount of CO_2 emitted, but it can optimize the time. There is a lot of talk about carbon efficiency and A, B, C rated cars, Blue Motions and so on and so on, but a car that is stuck in a traffic jam drives 0 km per liter regardless, however advanced its technology may be (as long as its engine is running). Apart from total traffic jams, even a medium congested road will increase CO_2 emissions by the cars: start and stop and ever-changing velocities is much worse than cars driving at a constant velocity.[25] So a traffic management system that can avoid congested roads will help the environment. Our system does this and more: we both counteract traffic congestion, and decrease the time a driving trip takes, even if there is no congestion. All else being equal, a quick trip emits less CO_2 than a long lasting one.

8.3.3 CO_2 based navigation

An interesting addition to our system could be to be able to generate the routes

23 Google Maps for Mobile or openBmap.org for example

24 Towards a spatial CO2 budget of a metropolitan region based on textural image classification and flux measurements: Remote Sensing of Environment (October 2003), 87 (2-3), pg. 283-294 Henrik Soegaard; Lasse Møller-Jensen

25 A lorry union website claims 3 times more: http://www.iru.org/en_policy_co2_response_flowingtraffic

that causes the least CO_2 to be emitted during a trip: CO_2-based pathfinding. Technically, it would be extremely simple to change our pathfinder from finding the fastest route to finding the cleanest route. At the present, the A* pathfinder uses a cost value to find its way. This cost function is the time it takes to travel a way, based on either the speed-limit, or the slowed-down congested speed, deduced from the gps fixes and timestamps reported by the clients. This cost function could be changed to be the amount of CO_2 emitted while traveling a way, and the pathfinder would find the cleanest route – voilá. CO_2-navigation!

But it is not so simple unfortunately. The barrier here is data: how can we tell how much CO_2 is going to be emitted? Different motors have different emission levels and efficiencies, and even the same motor in a different car, or at a different speed or different gear will change the emissions. Perhaps it would be possible to split the cost function in two: one part focusing on the actual vehicle, and one part focusing on external factors.

External factors first. These is the factors that are equal to all vehicles.

- The speed of a way: high speed causes more aerodynamic drag, but decreases traveling time.
- The surface of the road: rough roads causes more resistance, but slick ice is also bad for the emission levels.
- The type of road: speed-bumps, and frequent twists are worse than a straight road.
- Incline: driving uphill needs more power than going downhill.
- etc

Internal factors are those that differ from vehicle to vehicle.

- Engine efficiency: this also depends upon the speed of the car.
- Tires: worn winter tires with too little pressure in a hot summer can be a significant drawback
- Car aerodynamics: Most motor vehicles are very dirty from an aerodynamic point of view, and there can be a lot of differences between vehicles.
- Driver: The driving style of an individual can have a big part.
- etc

All these factors and many others have a bigger or smaller effect on the CO_2 emissions, a vehicle may produce. Our system could perhaps be changed to receive some vehicle data from each client, when receiving a request for a new route and then combine these with other data, stored in its map to calculate a cost-value for the ways, and thus come up with a cleanest route for that particular vehicle. We have included a CO_2 cost for roads in our system, but we do not use it for other purposes than calculating the approximate emissions on

each route.

8.3.3.1 Implementation of CO_2 data in our system

The values are based on the quite sparse data we have been able to get. Although there is a lot of information on the fuel economy and therefore the CO_2 emissions by cars in general, these are meant as comparison values between cars, so a buyer can choose a car by the fuel economy. The values are typically a measurement of a fixed, benchmark situation, that is supposed to be representative of a normal, mixed, driving profile with accelerations decelerations and different speeds. What we would like instead, is data on how different driving speeds affect the emissions. These would then be used for the different types of roads in our map. Apparently, the car manufacturers keeps these values for themselves, and only publishes the results of required benchmark(s). Manufacturers are even rumored to design their cars to perform well in the benchmarks, instead of normal day use, and this may be the incitement to keep their data safely tucked away.

After some investigation on the internet, and asking a Volkswagen car dealer for data that was better suited, we had to give up and take a different approach. We found that the the car in most widespread use in Denmark is the VW Golf, though the specific model is not detailed. We were able to find slightly more detailed data about this, than most other cars[26]. The European standard fuel economy test is using the so called New European Driving Cycle, which is a combination of an urban drive cycle and an extra urban drive cycle. These are supposed to represent a typical driving profile inside a city and outside a city respectively. During these cycles, the emissions are collected and analyzed afterwards for the result. VW also specified the partial fuel economy during each separate part of the test. This allowed us to calculate the emissions during these driving situations. The data and results are listed in Error: Reference source not found. Although meager, we would now have at least some data...

Table 8.2: CO_2 emissions calculated from the fuel consumptions as detailed in the data for the VW Golf 1.4

Drive cycle	Fuel consumption	CO_2 emission	Relative emission
Urban	8.5 L/100km	197 g/km	1.32
Extra-urban	5.1 L/100km	118 g/km	0.79
Combined	6.4 L/100km	149 g/km	1

We have entered the relative CO_2 figures into our map, guesstimating which road types could be considered “urban” or “extra-urban”. Seeing that the VW

26 http://www.car-emissions.com/cars/view/38130

Golf could be considered an average car, we have chosen this approach to make the external factors as explained above. By counting up this relative CO_2 cost, and multiplying with the combined-cycle CO_2 emission figure, we can then calculate the total emission of a route. Assuming that other cars have the same relative factors, we can multiply with their combined value (which is available for all cars in the EU) and get their CO_2 emissions. We admit that that there is quite some approximations and assumptions involved in this approach, but at least it shows that it is possible to implement a CO2 based navigation system.

8.3.4 Integration of public transportation

An important factor in the transportation infrastructure is the public transportation forms like trains and buses. OpenStreetMap has provisions for including data relating to public transportation, such as bus stops and train stations, and even include some routes in its data. There are proposals up for more deep integration of public transportation into the map. This proposal does not cover timetables, and timetables would be critical to the integration into our system. It would not be of any benefit for a user to know that there is a train route, he can take to avoid a traffic jam, if the train departs in 9 hours from now.

In Denmark we already have a service that covers public transportation routing and timing very well. “www.rejseplanen.dk” is a cooperative effort by the main transportation companies to maintain an up-to-date route-finding service at all times. This, however, does not have any provisions for private cars on the main road network. What we would need is a merger of these.

There are some snags inherent to coupling personal and public transportation. People driving a car will do so when they *want*. People taking a train will have to do so when they *can*. A user who wants to go from point-a to point-b may be able to do so in his car all the way, or there might be a possibility to take a bus or a train for part of the route. Our pathfinder could then guide the user to a train or bus stop. But it would have to do so in time to park the car, walk to the ramp and maybe buy the ticket before it leaves. Another issue is parking spots, the user would have to be guided, not to the station itself, but to a nearby parking facility, and one with a free spot too. If the user cannot find a free parking spot, the detour would be wasted – time-wise anyway.

The way a user would normally use our system would be when he enters his car and wants to navigate around congested roads to a goal. This would mean that to provide a route that includes public transport, we would have to be tightly timed, as there would be only a very short window of time to take a train or bus, before the effort would be wasted anyway. There would also have to be an amount of luck, in that there must be parking available within reasonable time and distance. These challenges would make it very difficult to

make a usable coupling between the public grid and the private cars, which could consistently provide fast and efficient routes to the users.

8.3.5 Driverless cars

The ultimate level of traffic control would be to take the man out of the loop. Studies have shown that the main cause of traffic jams may be human factors.[27] Accidents and driver errors are also almost entirely a human affair. It is also apparent that even people who do use a satellite navigation system to find their way takes a wrong turn now and then. The user does not always follow the directions of the navigation aid, unwillingly or on purpose. A guidance system that provides a good and consistent route will with time make the user more confident about following the directions and thus make them more likely to use our system as it was meant to be used: following the directions blindly to avoid the congestion, even though it may not be apparent that it is the best solution in all cases.

These human decisions are nearly impossible to control – and whether they should even be tried to be controlled is very much an question of ethics. We have already made a system that would be a nice thing to have for a “Big Brother” - giving “him” total control does not seem right to us. We want to make something that can advise the users, not decide for them, even though those decisions may prove wrong, people should still have the right to do the wrong thing...

Driverless cars are no longer a thing of science-fiction, although they are still a thing of science, and not ready for everyday use. But we are getting there.

8.3.6 TMC integration

(Nikolaj)

Traffic Message Channel (TMC) is a service delivering traffic and travel information to drivers. Normally transmitted to the user using the FM-RDS system, allowing users to get the information with their radio. This service is used by radio stations to transmit traffic information to drivers and by the navigation companies to update speed limits on ways. This way the user of the navigation can avoid traffic incidents and roadwork. The navigators which uses this also got a FM receiver implemented to receive the TMC, due the lack of Internet connection. In some countries the TMC is also brought to the users in a XML-structure. This allows devices with no FM-receiver but a internet connection to receive the data. Trafikken.dk brings this to the users in Denmark. This service is unfortunately not free, and therefore we have not

27 http://ing.dk/artikel/86162-stop-and-go-paa-motorvejen-det-er-kun-bilisternes-skyld

been able to implement it in our system. We have created the class QueueSetter which can set queues on ways. With this implementation we only need the XML parser and interpreter to implement this feature. The TMC implementation will allow the users to avoid these zones if there is a faster way. Without it, at least one client must drive though the zone with roadwork in order to report of the zone with a lower speed limit. The TMC also transmits about car accidents that an implementation also would prevent cars from choosing this way. Especially roadblocks are something we want to avoid. If the client's checkpoint is on the other side of the roadblock and the client just holds still, the server will not know about the queue before the roadblock is gone and it reaches the checkpoint. This may result in many other clients also being guided though this way with a roadblock.

9 Conclusion

(Nikolaj and Christian)

This thesis presented a prototype of a traffic guidance system. We have not been able to find similar project that uses the smartphone infrastructure to gain traffic information and do traffic control based on these data. Smartphones are sold as never before supplying us with a cheap and effective infrastructure without the need of purchasing expensive hardware.

The system contains: Android smartphone application, Server, Dummy Client and Visualization. It was necessary to narrow down the scope of the project, because of the primary goal, which was a working implementation of the system. With the server as the central unit and smartphone application to present the data to the user. The dummy client and visualization is developed mainly for testing. To meet the project scope a horizontal prototype has been developed.

The test-results indicates that we have successfully implemented the traffic guidance system. The data-structures, algorithms and communication gives good scalability and, if enough users driving with the application running, the system could navigate users around congestions.

The scale of the system is the main challenge. A certain number of users are required to get sufficient data, if data about congested roads is not present, our system will be like a normal navigation system. Another problem is the ethical problem in our way to control clients. We always have control over where a client is located. This can be traced to a unique smartphone, in this way we are in possession of sensitive data.

We have shown the scalability of the system and future possibilities of the system. The integration with public transport could minimize the traffic

otherwise heavily congested arterial roads. This extension could also reduce the CO_2 emission from transport. Another feature partly implemented is the CO_2 based pathfinding, allowing the user to choose the most CO_2 economic route. All these, as well as future integration with TMC, makes for a promising future for this type of system.

10 Appendix

10.1 Appendix-1: User's manual

10.1.1 Server

(Christian)

The server-side does not need any involvement from the user to run as such, but the graphical user interface provides overviews of the status and the ability to enter data for demonstration purposes or for testing.

The user interface consists of three main parts:

1. A console that the server writes to.
2. The list of queues or congested ways.
3. The map

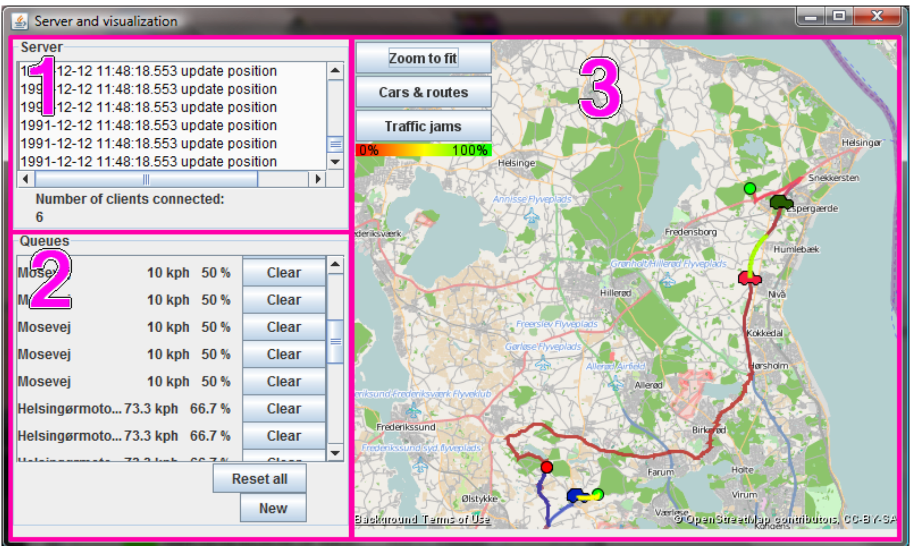


Figure 10.1:

The server console shows the incoming messages from all clients, prefixed by their id, which is basically a timestamp. dummy clients have their timestamps subtracted by 20 years, so it is easy to see whether a message comes from a real or a dummy.

The map allows the user to visualize the current clients, their routes and the queues that are currently detected. In the top left corner, there are buttons to control the view. The “Zoom to fit” changes the zoom level of the map to

include all markers, “Cars & routes” toggles the markers on and off, and “Traffic jams” toggles the congestion markers on and off. There is also a legend that shows the color scale of how much a queue slows the traffic.

The map can be zoomed using the mouse-wheel and moved by dragging the mouse while pressing the right mouse button.

Clients are shown at their last known position with a car-shaped icon with a color, specific to this client. The client's route is a line of that same color, with the starting location marked by a green dot and the goal marked with a red dot.

The Queuelist shows all the ways on which the traffic runs slower than normal. The list shows the way's name, the current speed and the relative slow-down as a percentage of the speed limit. There is also a button that clears the queue on this way, and resets the speed to normal. Below the list is a button that resets all the queues, and one that opens a pop-up window that allows the user to enter a new queue. This is shown in Figure 10.2.

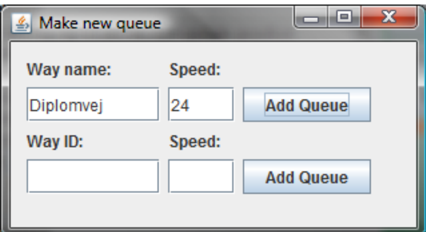


Figure 10.2: pop-up window for entering new queues.

To make a new queue, enter the name of the road and the speed of the queue (in kilometers per hour) in the top row and click the “Add queue”. This will set a queue on all ways that has that name. If you happen to know the ID of a way, a queue can be entered on a single way, using the lower row. The ID is the internal index as used by our graph.

10.1.2 Dummy client

The dummy client allows the user to add clients to the system for demonstration and testing purposes. The user interface has two tabs. The first for adding new clients, the second for managing and simulating these.

To make a new client, enter the latitude and longitude coordinates (in degrees) and the destination address, and click the “submit” button. Alternately the button labeled “autofill” can be clicked to provide a random starting position and destination address, generated from the map data. The location is within

the area covered by the map, but it may be far from populated areas. Below these controls, a panel labeled “multiple clients” allows the user to enter any number of random clients at once. Enter the number of clients to be added and click “ok”. This functions much like the autofill above, but allows the addition of dozens or hundreds of clients with one click.

The tab called “Active clients” shows a list of the clients that has been generated. Each is represented as a single line with a rectangle of the same color as that client has in the server gui. This provides for easy identification of the client. Also shown is the client's id, and how fast that client is driving. This is the percentage of time taken to drive down a way, relative to the time it would take, driving at the speed-limit of that road. To change this, enter the new value in the text box and click “change”.

Below this, the time can be simulated. This will simulate the clients driving along their assigned routes. To use these controls, enter the number of seconds to simulate, and click the “seconds” button to advance time a fixed amount of time. By clicking the “run” button, the time will advance continuously until it is clicked once more.

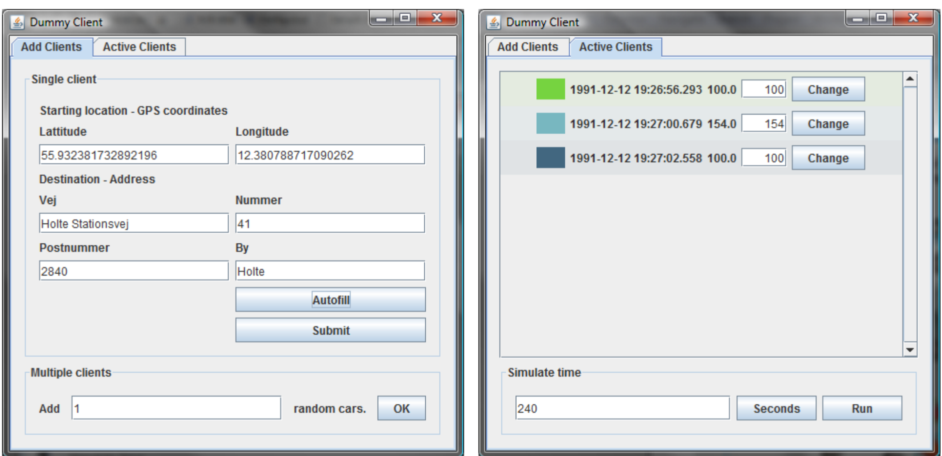


Figure 10.3: The two tabs of the Dummyclient gui.

10.1.3 Android Application:

(Nikolaj)

Installing the software:

You need a eclipse environment to build and install the Android Application ”Navidroid”. Also see “android developers”[28] for information on installing without Eclipse.

- Import the project from the zip file.
- Choose Android API 8
- Plug in your android based device and run as Android application
- The program is now installed.

Using the program:

- Open the Navidroid application
- Type in destination address
- It’s important to wait until the GPS signal has a fix. This is shown when the GPS logo at the top of the screen stops flashing.
- The ways your road is build on will now be shown on the confirmation page. Press confirm if it looks right.
- The “Navigation” screen is shown. It will tell you to wait until it got the right GPS fix
- The navigation starts navigate, and you can follow the arrows and distance with the name of the street you are supposed to turn at.
- The application will tell you when you are arrived to the destination.

28 http://developer.android.com/guide/developing/building/building-cmdline.html#RunningOnDevice

10.2 Appendix-2: Timetable

5 Points = 8 hours a week

Ressources	Hours
Resources in 13 weeks	832
Resources	***832***

Activity	Plan					Realized date				
		Start	Duration(days)	End	Estimate (Hours)	Start	End	Actual (hours)	% Actual	% Complete
Project Management		***09/01/2011***	***91***	***12/01/2011***		***09/01/2011***	***12/01/2011***			
Ongoing project management/reporting		09/01/2011	91	12/01/2011	10	09/01/2011	12/01/2011	10	100.00%	100%
Server side		*09/01/2011*	*91*	*12/01/2011*		*09/01/2011*	*11/08/2011*			
OSM Generation		09/01/2011	19	09/20/2011	43	09/01/2011	09/20/2011	55	127.91%	100%
Pathfinding		09/01/2011	19	09/20/2011	43	09/01/2011	09/20/2011	55	127.91%	100%
Communication		09/01/2011	19	09/20/2011	43	09/01/2011	09/20/2011	43	100.00%	100%
Traffic Control		09/01/2011	26	09/27/2011	43	09/01/2011	09/27/2011	50	116.28%	80%
Test / bugfix / Optimizing		09/01/2011	91	12/01/2011	35	09/01/2011	12/01/2011	45	128.57%	65%
Client side (Dummy)		***09/21/2011***	***71***	***12/01/2011***		***09/21/2011***	***11/08/2011***			
Communication		09/21/2011	6	09/27/2011	14	09/21/2011	09/27/2011	10	71.43%	100%
Data representation		09/21/2011	6	09/27/2011	14	09/21/2011	09/27/2011	14	100.00%	100%
Movement simulation		09/21/2011	13	10/04/2011	20	09/21/2011	10/04/2011	20	100.00%	100%
Test / bugfix / Optimizing		09/21/2011	71	12/01/2011	30	09/21/2011	12/01/2011	45	150.00%	100%
Client side (SmartPhone)		***10/18/2011***	***44***	***12/01/2011***		***10/18/2011***	***11/08/2011***			
Communication		10/18/2011	7	10/25/2011	14	10/18/2011	10/25/2011	14	100.00%	100%
Data representation		10/18/2011	14	11/01/2011	20	10/18/2011	11/01/2011	15	75.00%	80%
GPS API		10/18/2011	14	11/01/2011	20	10/18/2011	11/01/2011	40	200.00%	100%
User Interface		10/18/2011	21	11/08/2011	8	10/18/2011	11/08/2011	8	100.00%	90%
Test / bugfix / Optimizing		10/18/2011	44	12/01/2011	30	10/18/2011	12/01/2011	30	100.00%	100%
Visualization		**09/21/2011**	71	**12/01/2011**		**09/21/2011**	**11/08/2011**			
Data representation		09/21/2011	27	10/18/2011	30	09/21/2011	10/18/2011	45	150.00%	100%
Communication		09/21/2011	27	10/18/2011	30	09/21/2011	10/18/2011	15	50.00%	100%
Queue generation		09/21/2011	27	10/18/2011	30	09/21/2011	10/18/2011	15	50.00%	100%
Test / bugfix / Optimizing		09/21/2011	48	11/08/2011	30	09/21/2011	11/08/2011	30	100.00%	100%
Integration		11/01/2011	30	12/01/2011		09/21/2011	12/01/2011			
Integration of components		11/01/2011	30	12/01/2011	50	09/21/2011	12/01/2011	75	150.00%	100%
Test / bugfix / Optimizing		11/01/2011	30	12/01/2011	55	09/21/2011	12/01/2011	60	109.09%	100%
Documentation		***11/15/2011***	***35***	***12/20/2011***	***220***	***11/15/2011***	***12/20/2011***	***280***	127.27%	
Total					832			974		

10.3 Appendix-3: Changes in JmapViewer

Method	Description
JMapViewer	Constructor. Added initialization of variables
InitializeZoomSlider	Made the zoom controls. Removed the addition of most of the controls, added buttons to show/hide the markers, routes and queues. Added the queue color scale legend.
paintComponent	
paintPath	New: Paints a single MapPath
updatePathsAndMarkers	
updateQueues	New:
setMapQueuesVisible	New: Set the boolean variable and repaint
setMapQueueList	New: Set the entire list of queues and repaint
getMapQueueList	New: Get the list
addMapQueue	New: Add one queue and repaint
removeMapQueue	New: Remove one queue and repaint
removeAllMapQueues	New: Clear the entire list of queues and repaint
setServer	New: Sets the server from which to get the data about clients and queues

Table 10.1: Changes made to JMapViewer.java

Class name	Description
MapMarkerCar	Implementation of the mapmarker interface, paints a polygon the shape of a car
MapPath	Interface for displaying lines between a series of points on the map
MapPathRoute	Paints a colored path with a basic 3-point wide stroke
MapPathQueue	Paints a wide path with the color defined by a slowdown variable.
ScaleLegend	Paints a rectangle filled with a green/red gradient and labels

Table 10.2 Overview of the classes added:

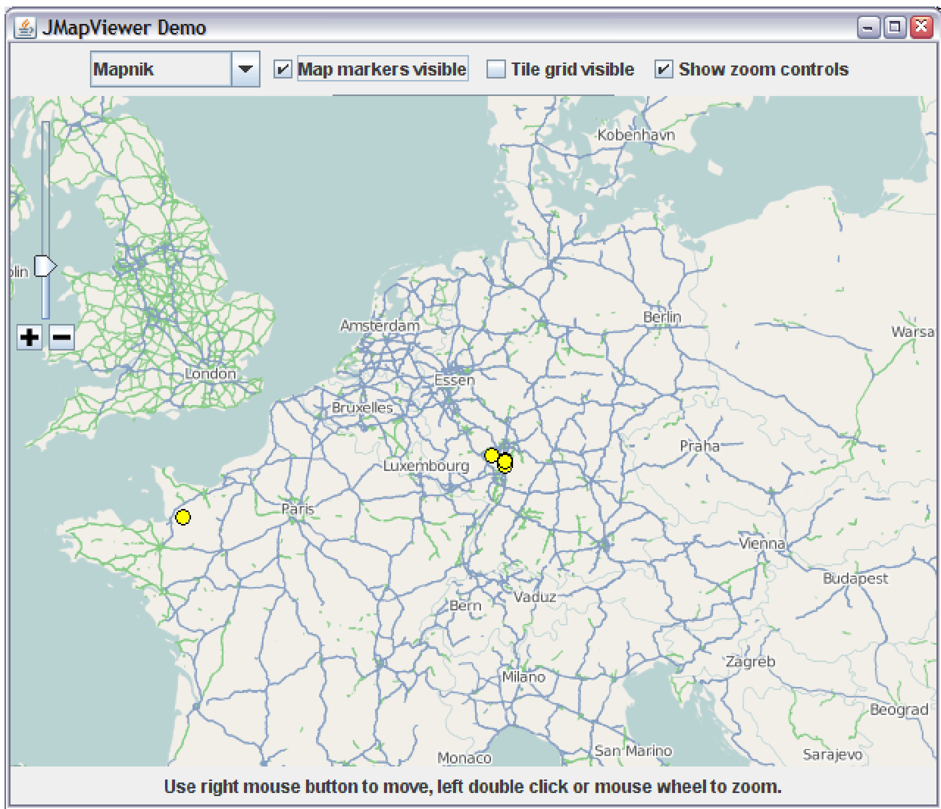


Figure 10.4: Screenshot of the JMapViewer demo from OpenStreetMap.org

10.4 Appendix-4: Test results

10.4.1 Black-box tests

Test Name **Tester** **OS** **Test no.**	**Black-box - routing** **Christian** **Windows Vista SP2** **1**
Scenario	Basic routing
Starting location	55.7832, 12.5215 (near Avej 3)
Destination	Cvej 1
Other preconditions	none
Expected outcome	System finds the fastest route
Result	Success. The center intersection is avoided because of the delay associated with 4-way intersections.
Screenshot	

Test Name **Tester** **OS** **Test no.**	**Black-box - routing** **Christian** **Windows Vista SP2** **2**
Scenario	Basic routing
Starting location	55.7832, 12.5215 (near Avej 3)
Destination	Avej 3
Other preconditions	None
Expected outcome	System doesn't find a valid route, because the starting and destination nodes are the same.
Result	Partial success. The server doesn't find a route, but reports “No route possible” - a better report would be preferred.

Test Name **Tester** **OS** **Test no.**	**Black-box - routing** **Christian** **Windows Vista SP2** **3**
Scenario	Basic routing, congested roads
Starting location	55.7832, 12.5215 (near Avej 3)
Destination	Cvej 1
Other preconditions	Heavy congestion on Fvej
Expected outcome	System finds the fastest route
Result	Success. The congested Fvej is avoided and center intersection is avoided as well
Screenshot	

Test Name **Tester** **OS** **Test no.**	**Black-box - routing** **Christian** **Windows Vista SP2** **4**
Scenario	Basic routing, congested roads
Starting location	55.7832, 12.5215 (near Avej 3)
Destination	Cvej 1
Other preconditions	Heavy congestion on Dvej and Fvej
Expected outcome	System finds the fastest route
Result	Success. The congested roads are avoided despite the intersection delay in the center
Screenshot	

Test Name **Tester** **OS** **Test no.**	**Black-box - routing** **Christian** **Windows Vista SP2** **5**
Scenario	Re-routing, congested roads
Starting location	55.7832, 12.5215 (near Avej 3)
Destination	Cvej 1
Other preconditions	After the user has started driving, a traffic jam is set at Cvej
Expected outcome	System finds the fastest route as in Test-1, but re-routes the client when the queue is added.
Result	Success. The tests starts as describes in Test-1. The new queue is inserted and as the client reaches the next waypoint, a new route is made, diverting him away from the jammed road.
Screenshot	

Test Name **Tester** **OS** **Test no.**	**Blackbox - queues** **Christian** **Windows Vista SP2** **6**
Scenario	Detection of queues
Starting location	55.7832, 12.5215 (near Avej 3)
Destination	Cvej 1
Other preconditions	After the user has reached the Bvej/Fvej intersection, his speed is lowered to 10kph
Expected outcome	Upon reaching the next route-point, a queue should be reported
Result	Partial success. The new queue is reported not upon reaching the next route-point but the next one over.
Screenshot	

Test Name **Tester** **OS** **Test no.**	**Blackbox - queues** **Christian** **Windows Vista SP2** **7**
Scenario	Removal of queues
Starting location	55.7832, 12.5215 (near Avej 3)
Destination	Cvej 1
Other preconditions	This setup is the same as after Test-6: a 10kph speed queue at Cvej, but the client doesn't get re-routed, and continues.
Expected outcome	When passing the queue, the client should report that the queue has gone, and it will be removed.
Result	Success. The queue is removed when the client has passed.

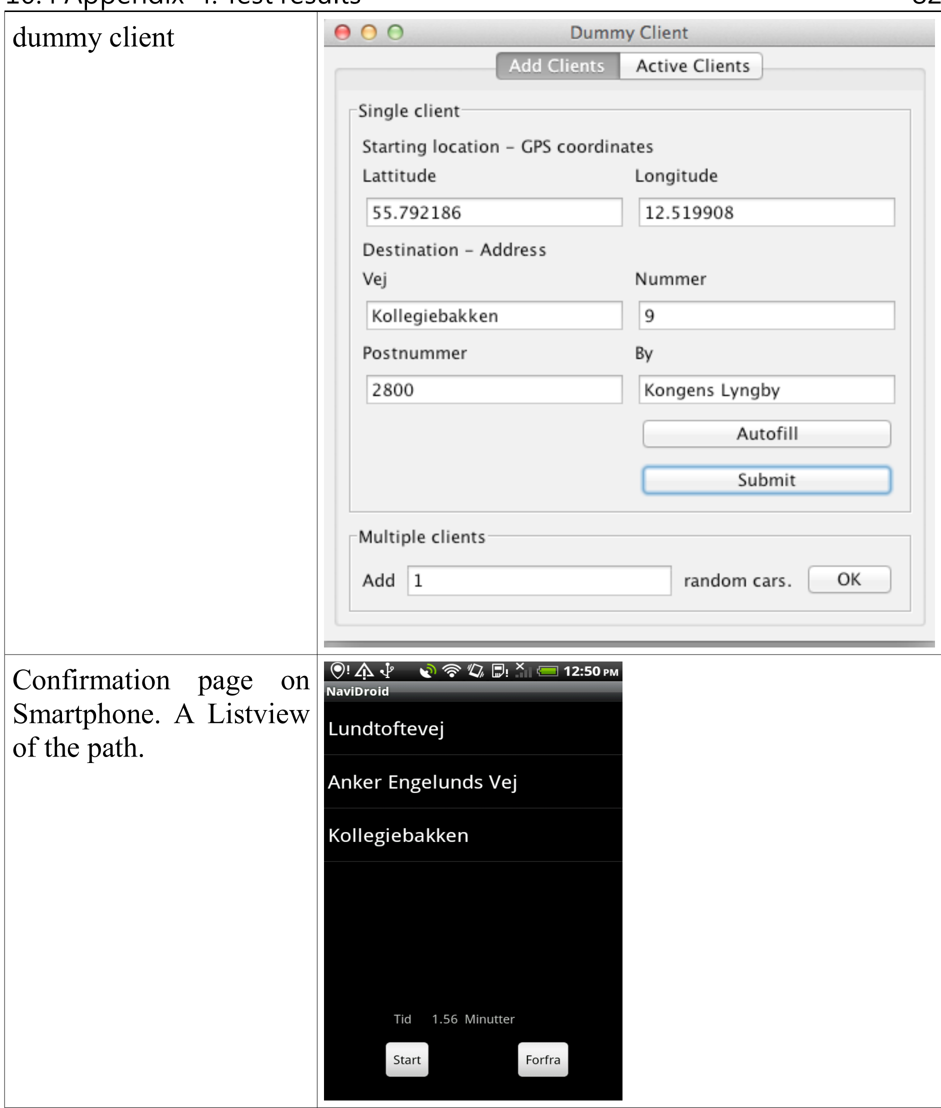
Test Name **Tester** **OS** **Test no.**	**Blackbox - queues** **Christian** **Windows Vista SP2** **8**
Scenario	Updating of queues
Starting location	55.7832, 12.5215 (near Avej 3)
Destination	Cvej 1
Other preconditions	This setup is the same as after Test-6: a 10kph speed queue at Cvej, but the client doesn't get re-routed, and continues, but at 40kph, instead of 50kph.
Expected outcome	When passing the queue, the client should report that the queue has sped up, and it will be updated to the new speed of 40kph.
Result	Success. The queue is changed when the client has passed. The list in the server UI is updated, and the color in the map is changed to a more green one.
Screenshot	

10.4.2 Use-case test results

Test Name **Tester** **OS** **Test no.**	**Use-Case test: Find Route** **Nikolaj** **Mac OS X Lion** **1**
Precondition	The user wants to get a route from current position to an Address.
Post condition	
Main path (M)	1: User type in the destination Address 2: The phone sends the information to server 3: Server processes the data and find the route 4: Smartphone receives the route
Alternative path1 (A1)	No internet on Smartphone then, the unit will not be able to send data .
Alternative path2 (A2)	No GPS signal, the Smartphone will not send the correct start location
Alternative path3 (A3)	No path found, If there is no possible path between start and end the Server should return an empty XML path.
Extra	Same setup on the dummy client

Table 10.3:

Test path	**Expected**
M	The Smartphone should receive the calculated route from server and start navigates to the destination. Same with dummy client
A1	The smartphone will not be able to get pass the first page in the application. The Dummy will not send any Data.
A2	The server will receive start Latitude =0.0 and start Longitude =0.0 because the GPS will not set the variables. The dummy are not dependent on GPS as it auto generates coordinates.
A3	The Smartphone will receive a path which only contains the start path and destination Same with dummy client
Screenshots	
Map Overview with route:	



dummy client	
Confirmation page on Smartphone. A Listview of the path.	

Table 10.4:

Test Name Tester OS **Test no.**	**Use-Case test: Drive along route** **Nikolaj** **Mac OS X Lion** **2**
Precondition	The user has received a route, and reached the first checkpoint.
Post condition	
Main path (M)	1: User moves towards the checkpoint 2: When the user is within 10 meters radius of the point, the smartphone will send an update. Including coordinates, ID and timestamp. 3: The server receives the update and sets the clients new position.
Alternative path1 (A1)	No internet on Smartphone then, the unit will not be able to send data.
Alternative path2 (A2)	No GPS signal, the Smartphone will not send the correct current location.
Alternative path3 (A3)	The user never reaches the start checkpoint.

Table 10.5:

Test path	**Expected**
M	The Smartphone will send data to the server with new position Same with dummy client
A1	The smartphone will not be able to send data to the server Same with dummy client
A2	The server will receive Latitude =0.0 and Longitude =0.0 because the GPS will not set the variables. The dummy are not dependent on GPS as it auto generates coordinates.
A3	The phone can not guide the user before he enters the first checkpoint The dummy are not dependent on GPS and therefore it always reaches the first checkpoint.
Screenshots	
Before first checkpoint is reached.	
First checkpoint reached	

Second checkpoint reached	
Client receives a route, and sends two updates to the server	1991-11-17 14:59:40.807 connected, new route found 1991-11-17 14:59:40.807 update position 1991-11-17 14:59:40.807 update position

Table 10.6:

Test Name **Tester** **OS** **Test no.**	**Use-Case test: Drive away from route** **Nikolaj** **Mac OS X Lion** **3**
Precondition	The user has received a route, and drives away from the route
Post condition	
Main path (M)	1: User driven away from route 2: Smartphone shows the checkpoint.
Extra	Same setup on the dummy client

Table 10.7:

Test path	**Expected**
M	The smartphone don't know where the user is. But keeps showing the checkpoint he is supposed to go to. The dummy client doesn't navigate
Screenshots	

Table 10.8:

Test Name **Tester** **OS** **Test no.**	**Use-Case test: Report congestion** **Nikolaj** **Mac OS X Lion** **4**
Precondition	The user has received a route, and drives on the route.
Post condition	
Main path (M)	1: User drives on a way 2: User drives to a checkpoint 3: Smartphone reports how long time since the last checkpoint. 4: Server calculates the time spend on way, and checks if its slower than normal. 5: If it's slower the server will set a congestion on the way.
Alternative path1 (A1)	The user never reaches the checkpoint
Extra	Same setup on the dummy client

Table 10.9:

Test path	**Expected**	**Result**
M	The smartphone should send the update to server, the server calculates a delay on the way, and set a congestion Same with dummy client	Pass Pass
A1	The smartphone never sends an update and therefore no congestion will be placed. The Dummy always reaches the checkpoints.	Pass
Screenshots		
Car Driving normal speed		

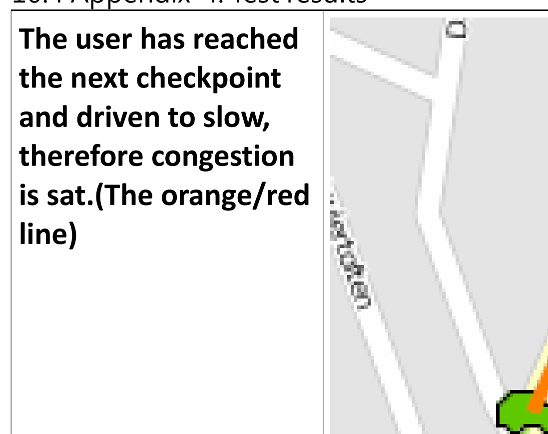
The user has reached the next checkpoint and driven to slow, therefore congestion is sat.(The orange/red line)	

Table 10.10:

Test Name **Tester** **OS** **Test no.**	**Use-Case test: Get new route** **Nikolaj** **Mac OS X Lion** **5**
Precondition	The user has received a route, and drives on the route. And a congestion occur on a way in the users path
Post condition	
Main path (M)	1: User drives on a route. 2: Congestion occurs on route. 3: Server calculates new route to user, and send it. 4:Smartphone navigates the new path.
Extra	Same setup on the dummy client

Table 10.11:

Test path	Expected	Result
M	The smartphone receives a new route from the server. And now navigates from the new information.	Pass
	Same with Dummy Client	Pass
Screenshots		
Route before congestion		
Green car is making Congestion and therefore Purple car now got a new route.		

Table 10.12

Test Name **Tester** **OS** **Test no.**	**Use-Case test: Exit → end route** **Nikolaj** **Mac OS X Lion** **6**
Precondition	The user has received a route, and reaches the destination on route.
Post condition	
Main path (M)	1: User drives on a route. 2: User reaches the destination
Extra	Same setup on the Dummy client

Table 10.13

Test path	Expected	Result
M	The smartphone sends the last position and the server removes the client's path from the map.	Pass
	Same with Dummy Client	Pass
Screenshots		
Before destination		
After destination reached.		
1991-11-20 21:19:39.873 connected, new route found		
1991-11-20 21:19:39.873		
Disconnect at 2011-11-20 21:20:41.466		

Table 10.14

Test Name **Tester** **OS** **Test no.**	**Use-Case test: Exit → Client vanish** **Nikolaj** **Mac OS X Lion** **7**
Precondition	The user has received a route, and drives on the route. But disappears.
Post condition	
Main path (M)	1: User drives on a route. 2: User disappears from route (No internet connection, No more battery)
Alternative path1 (A1)	The client stops the Application running on the smartphone.
Extra	Same setup on the Dummy client

Table 10.15

Test path	**Expected**	**Result**
M	The client will be removed from the server after an amount of time	Fail
	Same with Dummy Client	Fail
A1	The application will send a disconnect to the server. Same with Dummy Client	Pass
Screenshots		

Table 10.16

10.4.3 Pathfinding comparison test results

Test Name **Tester** **OS** **Test no.**	**System test – Path finding comparison** **Nikolaj** **Mac OS X Lion** **8**
Purpose	Determine how the Pathfinding algorithm works compared to a professional service. (krak.dk)
Post condition	
Main path (M)	Enter the same start position and destination and compare the route suggestions. Start: Kollegiebakken 9, 2800 Kongens Lyngby Stop: Solsikkemarken 34, 2830 Virum
Alternative path1 (A1)	Start: Kollegiebakken 9, 2800 Kongens Lyngby Stop: Gymnasievej 21, 4600 Køge

Table 10.17

Test path	**Expected**	**Result**
M		Same route:
	Krak:	Estimated traveling time: 5 minutes
	Our system	Estimated traveling time: 5,34 minutes
A1		Different route:
	Krak:	Estimated traveling time: 37 minutes
	Our system	Estimated traveling time: 33.87 minutes
Screenshots		
Krak : Kollegiebakken-Solsikkemarken.		
Our system: Kollegiebakken-Solsikkemarken.		

Krak: Kollegiebakken-Gymnasievej.	
Our system: Kollegiebakken–Gymnasievej.	

Table 10.18

10.4.4 GPS-fix test results

Test Name **Tester** **OS** **Test no.**	**System test – GPS-fix** **Nikolaj** **Mac OS X Lion** **7**
Purpose	Determine how well the GPS works in smartphones
Post condition	
Main path (M)	1. New Route 2. The application runs for 5 minutes 3. Counts each new fix of GPS (location changed)

Table 10.19

Test path	**Location**	**Result**
M	Indoor	16 fixes, 3.22 minutes until first fix. Equivalent to one fix each 18 seconds.
M	Copenhagen (Julius Bloms Gade)	91 fixes, 1.44 minutes until first fix. Equivalent to one fix each 15 seconds.
M	Outdoor on open field (Søndre marken, Frederiksberg)	151 fixes, 0.19 minutes until first fix. Equivalent to one fix each second.

Table 10.20