Distributed computing for the public transit domain

Degree Project of 15 credit points
Computer Science

Date/Term: 13-06-05
Supervisor: Andreas Kassler
Examiner: Thijs Holleboom
Serial Number: 2013:09
Distributed computing for the public transit domain
This report is submitted in partial fulfillment of the requirements for the Bachelor's degree in Computer Science. All material in this report which is not my own work has been identified and no material is included for which a degree has previously been conferred.

______________________________
Anders Andreasson

______________________________
Andreas Arvidsson

Approved, 2013-06-05

______________________________
Advisor: Andreas Kassler

______________________________
Examiner: Thijs Holleboom
Abstract

Traditionally each public transportation company have had their own services and software for transit information and travel planning. If you wanted to travel over a longer distance using public transportation you were required to utilize numerous different services to obtain all the information needed for your trip. This is both laborious, tedious and in some cases leads to misinformation and bad planning.

What our thesis contains is a description of the software system we have devised for the purpose of aggregating all available public transit information in Europe into one unified service.

The basis for this system is three components:

- The client applications (i.e. mobile phone apps, PC web pages etc.).
- The distribution system with its routing and load balancing algorithms.
- The calculation and logic system that manages the transit data.
Contents

1 Introduction ...........................................................................................................................1
  1.1 The application.............................................................................................................1
  1.2 Distributed computing...............................................................................................2
  1.3 Centralized management............................................................................................2
  1.4 Summary.......................................................................................................................3

2 Background ...........................................................................................................................3
  2.1 Bachelor project............................................................................................................3
  2.2 ITRACT........................................................................................................................4
  2.3 Similar work................................................................................................................6
  2.4 Preexisting software and formats................................................................................6
  2.5 Summary.......................................................................................................................13

3 Design ...................................................................................................................................13
  3.1 Design development....................................................................................................13
  3.2 General design.............................................................................................................16
  3.3 Automated server management..................................................................................19
  3.4 Request routing..........................................................................................................20
  3.5 Proxy class hierarchy.................................................................................................22
  3.6 Proxy event class hierarchy.......................................................................................26
  3.7 Server helper class hierarchy....................................................................................27
  3.8 Management web page design..................................................................................29
  3.9 Client web page design.............................................................................................30

4 Implementation ...................................................................................................................32
  4.1 Introduction.................................................................................................................32
  4.2 Automated server management..................................................................................34
  4.3 Request routing..........................................................................................................37
  4.4 Event management......................................................................................................45
5 Results and system evaluation

5.1 Introduction

5.2 Client web page result

5.3 Management web page result

5.4 System testing

5.5 Summary

6 Conclusion

6.1 Project evaluation

6.2 Problems and complications

6.3 Future work

6.4 Andreas personal reflection

6.5 Anders personal reflection

7 References
List of Figures

Figure 2.1 JSON example................................................................. 9
Figure 2.2: Open trip planner description................................. 10
Figure 2.3: One bus away description........................................ 11
Figure 3.1: System general design............................................. 16
Figure 3.2: Proxy design class diagram....................................... 23
Figure 3.3: Proxy event design class hierarchy............................ 26
Figure 3.4: Server helper design class diagram........................... 27
Figure 3.5: Client web page general layout............................... 31
Figure 4.1: Proxy implemented class diagram............................. 33
Figure 4.2: Routing graph table code.......................................... 37
Figure 4.3: Coordinate to index formula...................................... 38
Figure 4.4: Coordinate to index formula simplified................... 38
Figure 4.5: Swedish graph geographic bounds............................ 40
Figure 4.6: Ray casting pseudo code.......................................... 41
Figure 4.7: Routing graph table result........................................ 43
Figure 4.8: Coordinate to index formula simplified................... 44
Figure 4.9: Proxy event implementation class diagram............... 46
Figure 4.10: Proxy event callback example............................... 46
Figure 5.1: Client web page result............................................ 48
Figure 5.2: Client web page trip plan result.............................. 49
Figure 5.3: Management web page overview result.................... 49
Figure 5.4: Management web page graph result......................... 50
List of tables

Table 2.1: Open trip planner component description .......................................................... 10
Table 2.2: One bus away component description ................................................................. 12
Table 5.1: Test hardware description .................................................................................. 52
Table 5.2: Test bay area result ............................................................................................ 52
Table 5.3: Test Sweden result .............................................................................................. 52
1 Introduction

This is a dissertation written by Andreas Arvidsson and Anders Andreasson for our bachelor degree course in computer science from Karlstad university.

We did this project as a part of the European Unions ITRACT [1] project with Karlstad university professor Andreas Kassler as our supervisor.

In the following pages we will describe for you what our project was about, the choices we made, the problems we encountered and the result we got with a following evaluation.

1.1 The application

The goal of our project was to implement an interactive map application for the public transportation domain that offered augmented functionality and usability compared to existing travel planner applications.

The idea was that our application should through a graphical user interface centralized around a geographic map show both the static transit information from time tables and the dynamic transit information from real time updates in a user friendly and user interactive way.

The idea of a map centralized application was built upon the notion that it would be beneficial to the user experience if the user could get the information in two different formats:

1. The “classic” view where transit information is mediated to the user in a text style way through tables and lists containing arrival and departures times, planned itineraries and other transit data.

2. The “visual inspection” view where the user visually get the transit data from the map and can then extrapolate and estimate arrival and departures times based on the users own experience. Instead of writing in a schedule table that the bus will arrive in 5 minutes we graphically on the map show the users current location, all the adjacent bus stops and the real time position, speed and heading of all the vehicles in the area.

We believe that the combination of these two views of transit information will provide the user with an improved user experience, but what makes this project truly unique was the scale of which the project was to be realized. This application should be able to not just support and
plan trips between multiple public transportation agencies but even across multiple countries with the final goal of a unified travel planner for the whole European Union. Once this application is finalized the goal is that you should be able to plan you travel with such detail that you would get everything from the walk path between your house and the bus stop to which buses you should take where and when. The design even extends to more unique modes of transport like ridesharing and demand responsive transport.

This is the dream of a travel planner applications that is both easier to use and that offers extended information and functionality compared to existing applications. You could say that our final goal is to make all existing public transportation planning and scheduling services obsolete.

### 1.2 Distributed computing

Due to the scale of which the project was to operate the need to distribute the computational load is paramount for the performance and user experience of the application. To solve this problem we had to devise our own computer cluster consisting of a centralized master node and multiple underlying computational nodes. The master node accepts all the data request from the application and then routes these requests to its underlying computational nodes that then process the requests and return the requested data back to the application.

For this purpose we had to write a control and routing software for the master node and a server software for the computational nodes in which design we had to take into account aspects such as performance/load balancing, routing algorithms, availability/redundancy, security and much more. This drove the design of our computer cluster to become more and more sophisticated with both width in the form of load distribution nodes and depth in the form of redundancy nodes and to top it all of we had to write our own routing algorithm capable of distributing request according to load that also factors in redundancy.

### 1.3 Centralized management

With the distribution of data processing across multiple nodes a centralized management system was necessary. To this end we implemented a management web page that monitors all the nodes in the computer cluster and offers configuration functionality that enables the administrator to setup and manage the whole system from one centralized point in the system.
1.4 Summary

To summarize our project we have an interactive travel planner application covering the whole of Europe, a distributed computational system consisting of a computer cluster and a centralized management web page. To achieve this we have to implement:

1. A client application with an interactive graphical user interface that shows transit data in multiple formats.
2. Implement a master node control software for a computer cluster with our own routing and load balancing algorithms
3. Implement a computational node software capable of computing transit data to find the shortest itineraries for multiple modes of transport.
4. Implement a management web page capable of monitoring and configuring the computer cluster.

This is a monolith of a project to say the least and we hope you enjoy reading about how we worked towards realizing it.
2 Background

2.1 Project choice

The reason why both members of the group choose to work within the ITRACT project is that neither of us have ever done anything quite like this project. We saw the prospect of working within the ITRACT project as an opportunity to gain some new knowledge and experience in areas ranging from programming and software design to the intricacies of public transportation.

Beyond the aforementioned intellectual rewards of this project we also see a real life demand for this type of service and the potential for us to create a product that would be widely used and appreciated by the public is an attraction in itself. This combination of intellectual gain and real life applicability makes this project ideal for us and we will do our utmost to create a product that lives up to the dreams and expectations of the ITRACT project.

2.2 ITRACT

ITRACT is a project within the European Unions “The North Sea Region Programme”[2]. “The North Sea Region Programme” is a program which aims to improve the living and working conditions in the North sea region by increasing cooperation between the different countries in the North sea region.

The name ITRACT is an acronym for “Improving Transport and Accessibility through new Communication Technologies” and as the name implies the purpose of the ITRACT project is to improve the connectivity and accessibility of remote areas in the North sea region through infrastructure innovations in both areas of transport and communication.

The ITRACT project started in January 2012 and will continue throughout 2014. The budget for the ITRACT project stands today at 3,799,900€.
**Aim**
The final of the ITRACT project is to create sustainable regional economies throughout the North sea region by improving both the physical modes of transport as well as the communication and information technologies involved.

The ITRACT projects brings together people with the right knowledge and competence in different fields of technology together with the right people in the field of socioeconomic with the purpose to produce new and innovative applications to further the goal of improved connectivity and information sharing across the North sea region.

**Background**
Many of the areas in the north sea region is located at a distance from the main economic seats of their countries and thereby socioeconomic development progress in a lesser pace than the more centralized parts of the country. Some of the key components in the socioeconomic development is connectivity and accessibility and it is these areas that the ITRACT project works to improve.

**Partners**
To help achieve their goals the ITRACT project has several partners located in different countries in the north sea region. Most of the consists of Universities, municipalities and local public transportation agencies.

The Netherlands:
1. Hanze University of Applied Sciences Groningen
2. Gemeente Oldambt
3. OV Bureau Gr/Dr
4. University of Groningen

Germany:
1. Jade University
2. VEJ

United Kingdom:
1. DITA
2. Shuttledrive
3. Metro

Sweden:
1. Viktoria Institute  
2. Karlstad University  
3. Värmlandstrafik  
Norway:  
1. University of Stavanger  
2. Rogaland County  

2.3 Similar work

There already exist a large number of services and products with partially similar functionality to what this project will produce. The main difference is that our product implements a wider range of possibilities on a larger scale.

Almost every public transportation company have their own version of a trip planner in one form or another. Many of them have trip planning services for a wide variety of computer platforms ranging from desktop computers via a web page to applications for mobile operating systems. Most of these trip planning services exclusively supports that particular transportation agencies own lines and often in a somewhat narrow geographic area where that agency operates.

Traditionally this means that if you want to travel over a larger geographical area you have to use several different transportation agencies each one with their own trip planner and sometimes the transportation company responsible for a leg of your journey doesn’t even offer that kind of service.

What makes our product stand out from the crowd is primary that our product is not restricted to a single transportation agency, or a single country for that matter. This product will also support a large range of different modes of transportation including but not exclusive to: travel by own car, walking, demand responsive transportation and public transportation.

Our hopes is that somewhere in the foreseeable future our product will be able to plan a trip from the users home in one country to the users destination in another country. Including walking or driving to the first stop and changing vehicle and/or transportation mode along the ride until the user has reached the destination.

All this should be done with both static time schedule data as well as dynamic real time updates of the vehicles geographical position all to augment the trip planners precision. A
service with functionality like this doesn’t exist right now as far as we know. Some products offer part of this functionality, but not all in one package.

2.4 Preexisting software and formats

In the implementation part of this project we have used several preexisting software and formats as part of our product. To have a basic understanding of how these software and formats work is necessary before you can read and understand the following design chapter. Here follows a list with a short description of the most important software and formats we use.

**GTFS**

GTFS [3] is a text based, human readable, open standardized format for public transportation schedules and associated geographic data. The name GTFS is an acronym for “General Transit Feed Specification” but since the GTFS format was developed by Google and the Portland TriMet transportation company it was originally known as the “Google Transit Feed Specification“.

What the GTFS format does is give us a standardized format for storing and transmitting public transportation schedules and its associated geographic data. The reason why the usage of GTFS is beneficial for projects like this is that since our product will incorporate many different transportation agencies from several different countries we need a standardized format for receiving and parsing the data from all agencies. If every company has their own format we will not only need to parse each data stream differently but we will also get a different types of data with names and labels different for each agency. This would not be a very efficient way of working.

The design of our product is so that all transportation agencies supply their data in the same format which is GTFS. This is the input of all schedules and transit information for our product. If some of the companies can’t supply their data in this standardized format there is always the possibility of implementing a converter at either their or our end, but in that case the converter is a standalone component and not a part of the product we are making for this course.
The GTFS format specifies a set of very specific text files where some are required and some are optional. These text files consists of rows of text strings with comma separated values. There is also a GTFS specification for a real time data stream. This is used to update the real time geographic position of the vehicles and to inform about transit changes and alerts.

**OSM**
OSM [4] is an acronym for “Open Street Map”.
OSM is an open source map service of the entire world where information about terrain, political regions and boundaries, transit infrastructure and more is stored and this data is accessible free of charge. Since it is open source, users can help to update the world map to make improvements to it. OSM have an extensive tagging system for information about buildings, traffic signs, street lamps and more.

**JSON**
JSON [5] is a text based, human readable, open standardized format for data exchange. The name JSON is an acronym for “JavaScript Object Notation” and the format is derived from the scripting language JavaScript but despite this it is language independent with parsers available for many different programming languages.
JSON is used to represent simple data structures and associative arrays(i.e data objects) in a serialized format as text strings used for data exchange.

To use the JSON format we start with a data object. This object can more or less be any kind of data object. e.g. a single integer data object, an array of strings or a user defined class instance object.

Once you have the object you serialize it with your JSON parser of choice that produces a JSON formatted text string that represents your chosen object.

This text string is now readable by humans. i.e. any human that reads this string can now read and understand the entire structure of your object. If the object was a class instance you can now read what data objects is in the class with both name and value and if so is there an instance of another class in this one and the data objects for that class and so on.

Once you have your JSON string you can now exchange or transmit this to another system and when they receive it they can parse the JSON string which recreates the original object.
Using a JSON parser you can transmit large structures of data as human readable text and recreate the structure at the other end. Transferring data this way is much easier to implement and debugging than transferring the data as pure machine code. The human readable aspect of the JSON makes it easy to understand, troubleshoot and debug.

Example of a class object containing one instance of the class person and one instance of the class car.

```json
{
  "person": {
    "name": "Johan",
    "age": "37"
  },
  "car": {
    "manufacturer": "Volvo",
    "model": "V70"
  }
}
```

Figure 2.1 JSON example

**REST API**

REST [6] is a web service API (Application programming interface). The interface accepts HTTP request messages and responds accordingly. Mainly the API we use for our products consists of HTTP GET request with a few instances of HTTP PUT.

For our implementation all HTTP GET requests are responded with the return of a JSON string containing the requested data e.g. HTTP GET message asking for all stops on a route will return a JSON describing all the stops on that route.

**Open trip planner**

Open trip planner [7] is open source project used for public transport services. Open trip planner has since it was started up in 2009 gathered a large community of both users and developers that continuously work on making this project better in terms of performance and stability. In this short time span the project has already been deployed in 10 different countries.

The core of the Open trip planner is the trip planner API used to find the shortest path between two geographical point with support for walk, bicycle and different modes of public transportation. The program calculates several suggested itineraries based on different aspect such as speed, fewest transit changes, shortest walk path etc. The trip planner has support for
real time information to achieve more accurate information about where transit vehicles are at any given moment, but Open trip planner does not just support trip planning. Via Open trip planners RESTful API additional information about stops, routes, departures, server info and much more is available.

It is possible to load several graphs simultaneously into one Open trip planner server. In that case each graph, or router as Open trip planner like to call them when they are loaded, each have a unique ID.

Here follows a description of the main components in the Open trip planner platform:

![Figure 2.2: Open trip planner description](image-url)
<table>
<thead>
<tr>
<th>Component name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph file</td>
<td>File structure constructed by the GTFS and OSM data that contains all the necessary static transit data for the software to function.</td>
</tr>
<tr>
<td>Graph builder</td>
<td>Software that constructs a single graph file based one or more static GTFS sources and the OSM data related to the geographic transit regions for those GTFS sources.</td>
</tr>
<tr>
<td>Open trip planner</td>
<td>“Open Trip Planner” core software. This software reads data from the graph file and the GTFS real time feeds and uses this data to calculate fastest routes, departures for stops, stops on routes and so on.</td>
</tr>
</tbody>
</table>

*Table 2.1: Open trip planner component description*

**One Bus Away**

One bus away [8] is an open source project aimed to make life easier when using public transport.

The project was originally started up in Seattle/Puget Sound region and managed by University of Washington which still is working and improving the project in different periods.

One bus away project is currently used in several cities and is in development to be used by a few other.

One bus away have also have support for a variety of API functions that all is open source which can be easily change. With One bus aways RESTful API new functions can easy be developed to use information from the One bus away services. Information that can be gathered from the One bus away services is for example trips, routes, stop schedule, stops, transit vehicle and agencies.

The project use static and real time data to give up to date information about when a bus is going to arrive and depart a certain stop.
Figure 2.3: One bus away description

<table>
<thead>
<tr>
<th>Component name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bundle file</td>
<td>File structure constructed by the GTFS data that contains all the necessary static transit data for the software to function.</td>
</tr>
<tr>
<td>Bundle builder</td>
<td>Software that constructs a single bundle file based one or more static GTFS sources.</td>
</tr>
<tr>
<td>One bus away</td>
<td>“One Bus Away” core software. This software reads data from the bundle file and the GTFS real time feeds and uses this data to calculate departures for stops, stops on routes and so on.</td>
</tr>
</tbody>
</table>

Table 2.2: One bus away component description

**Open Ride**

Open ride [9] is an open source project for arranging ridesharing via smart phones. The system aims to improve spontaneous shared rides between people to protect the environment. Open rides main purpose is to make it easier for passengers and drivers to find each other. The system features functionality where drivers can inform the ridesharing system of single trip or schedule recurring trips. Passenger can then access the web page or mobile application to find a ride.
Open ride uses a smart search algorithm in real time where the driver and passengers can be on traveling foot and still find each other. When searching for a ride the open ride system finds a route and the driver and the passenger is contacted and connected to each other.

Trust between random driver and passengers can be an obstacle for many people the open ride offers a thrust building system where the thrust level is based on a two sided rating system where passenger and drivers can judge each other.

Open ride is going to be a component in the finalized version of this product but the entire part regarding ridesharing is implemented by other students working within the ITRACT project and is nothing we are going to work with for this course.

2.5 Summary

During this chapter we have looked into several different areas of information regarding the ITRACT project ranging from the background and history of the project to preexisting software like One bus away and Open trip planner to formats like GTFS and JSON. These software and formats are a vital part of our project since they are what our product will be based upon.

Now at the end of this chapter the you the reader should have an understanding of why we choose to do this project and maybe more important you now possess the necessary knowledge needed to understand the following chapters of design and implementation.
3 Design

3.1 Design development

At the start of this project we had close to no information about the desired final product. All the information we got was that it should be an “interactive map” for public transportation, but we had no predefined specification for what we should do or how we should do it.

During the first week of the project we had a meeting with our project supervisor Andreas Kassler where he outlined the basis of the interactive map. The information we got was that the interactive map should be based on two pieces of existing software, Open trip planner and One bus away. Based on this information we made a design where Open trip planner and One bus away where used together with an overlying shell that combines the functionality of both these software into one API for a web page containing the map service to access.

So our initial design was the already existing Open trip planner and One bus away software running alongside a shell/web-page all on one single machine. The implementation of this shell/web-page would be our product for this project.

As it turned out this initial design was further from the final product than any of us knew at this point. Over the entire time span of the project the design had to be incrementally changed and updated due to new requirements on our product from other people working within the ITRACT project.

Meeting with local public transport agencies

A few weeks into the project we had meetings with the local public transport agencies “Värmlandstrafik” and “Karlstadbuss” to get their input on what they want for the final product. Below we will try to summarize the different desired aspects and functions they had on the final product.

1. Interactive map. This is the base of the graphical user interface. Users should see the transit data on a map with the functionality to click on stops and vehicles to gain additional information.
2. Display all stops in an area. Preferably automated by GPS or IP look-up.
3. Click on a stop should display information about that stop, departures, routes, operation status etc.
4. Be able to display all stops on a given route.
5. Display the real time position for vehicles connected to a given stop or a given route on the map.
6. Trip planning. Both by clicking on the map and by entering a station name and/or a street address. The trip planning should be multimodal. i.e it should support different transport modes like bus, train, subway but also support walk and car.
7. Support for ridesharing/carpooling except for the traditional modes for transport we would like a function where you can announce you interest in ridesharing and also plan trips based on other peoples announced trips.
8. Support for demand responsive transport. For some public transport routes you need to order the vehicle to pick you up.

These are all functions and features of the web page interface and since that was what our product was about at this stage that is exclusively what we talked about with the local public transport agencies. How we achieve these functions and what we do in the back end(i.e behind the user interface) is once again up to us to figure out and to decide.

So after meeting with both our project supervisor and the local transport company partners the design and implementation specification was still up to us. This gives a both a lot of freedom to do as we think is best but also a lot of pressure to make a good design since we have to decide every single aspect of this project by our lonesome.

The demand responsive and ridesharing parts we was informed would be taking care of by other student groups working with the ITRACT project so we wouldn’t have to care about that.

**Weekly meetings with project supervisor**

During the entire run of the project we have had weekly meetings with our supervisor Andreas Kassler to update him on our progress and on most of them he have instructed us to implement some new functionality that he wanted for the product. Functionality like:

1. Support for multiple servers each each running multiple graphs covering different geographical locations. Some servers may have redundant information and the calls should somehow be distributed among the servers according to load. This
implementation should be seamless and totally transparent for the user of the shell i.e a client or client programmer.

2. Route client request explicitly based on geographic coordinates.
3. Route client request among redundant servers based on request round trip time(request delay) for each server.
4. Be able to manage servers from the shell: deploy and undeploy graphs, build new graphs and delete existing graphs, deploy graphs based on free memory on the servers.
5. Integrate a new existing software called Open ride for the ridesharing part. The routing and API for this is to be done through our shell.

The result of these weekly meeting with Kassler was that almost every week we had to make some changes to our design and implementation to suit this new functionality but with increasingly demanding requirements and growing intricacy of the code we soon realized that our existing design was inept to handle this kind of complexity and had to be completely redone.

Remember that our original design was one web page/shell running on one single machine. To go from that to a management system running multiple software on multiple networked machines is a big difference in terms of code complexity.

So halfway through the project we had to completely rethink and redesign the entire product which of course led to a major rewrite of both this dissertation and the already implemented code.

This the final design is going to be described in the following chapters.
3.2 General design

This is the overview of the main components in our design:

![System general design diagram]

**Figure 3.1: System general design**

MMP = Multimodal planner, OR = Open ride, OTP = Open trip planner

**Proxy**

To achieve all the features mentioned in the previous chapter with good performance we choose a design consisting of computer cluster with a centralized master node we call the proxy. The purpose of the proxy is to route client requests and aggregate data from several underlying computational nodes/servers each running multiple software with the explicit purpose of combining the features and functions of all these software into one unified API.

All requests from the clients are directed to one single API at the proxy which routes the requests to the correct server or servers and then return the information from the servers back to the client.

With this design we should be able to serve multiple clients making multiple requests to multiple servers simultaneously and the client doesn’t even need to know that there are any underlying servers. All they see is the proxy API.
Transit platforms
What we call a transit platform is the preexisting public transport software that we are running on the computational servers for the transit data management and logic. These are software like Open trip planner, One bus away, Open ride etc.
For this the first version of our project we are not going to be using One bus away since we can get by with only Open trip planner for most of the desired functionality. There may be a point in the future where One bus away is added but not during this course.
Open ride is going to be used, but all the implementation for that platform(and everything else related to ridesharing) is done by other students working within the ITRACTS project so we just need to have it as a part of our design for future integration with our system.

Server helper
To help us with some of the more intricate functionality of the management system we require a software running on each computational server that we call a server helper. This software automatically register itself to the proxy and help with functionality like sending updates of the server status to the proxy, building new graphs files, delete existing graph files, register and unregister existing graph files with the different transit platforms but maybe even more import is that all incoming requests to the transit platforms go through the server helper. The reason for doing this is so that we can filter the responses to better suit our needs without putting that resource drain on a centralized node like the proxy.

Multimodal planner(MMP)
The multimodal planner is a special type of computational server in that sense that it isn’t running a transit platform. The purpose of this server is to aggregate data from several different transit platforms running on several different servers into on trip planning API. Normal trip planning requests go directly to an Open trip planner server but when a client wants to plan a multimodal trip the request is forwarded to a multimodal planner server(multimodal in this context means transport modes stretching over multiple transit platforms).
The only difference between the multimodal planner and the other servers besides the lack of transit platforms is that the multimodal planner servers are explicitly registered as an multimodal planner with the proxy through the management interface.
All this logic is going to be implemented in the standard server helper software, but for now it is developed but other students(since it has to do with ridesharing) in a freestanding software that we have to integrate later.
**Graph repository**
The computer cluster has a centralized repository that contains all the graphs for all the different transit platforms. This is an arbitrary network resource that is reachable from all the nodes in the computer cluster. Whenever one of the transit platforms start up the graph file is loaded into memory from the graph repository.

With this design we can easily update a given graph on all the servers by rebuilding one single centralized graph file and then instruct all the server to reload that single graph file. There is also plans for the future of a revision control system to keep track of different build versions of the same graph file in the repository.

**OSM repository**
The OSM repository is much like the graph repository a centralized network resource that contains all the OSM data. OSM stands for Open Street Map and contains geographic and infrastructure data used for the build of new graphs for multiple transit platforms.

**Management web page**
The proxy is going to have a management web page where the system administrator can monitor the status and manage the different servers and graphs.

On this homepage all the underlying servers will be shown along their status and monitored data such as memory usage and status of the deployed platforms and their graphs.

There will also be functionality to modify the proxy content, remove servers, build new graphs, deploy and undeploy graphs, reload graphs and so on.

All this should be done through a user friendly graphical interface.

**Client web page**
Aside from the routing and management system we’re also going to implement a prototype of our initial interactive map web page. The focus of our product have been moved from the web page to the management and routing system but we are still going to implement a prototype web page. Mainly for the purpose to have something to display to the local public transport agencies we are working with but also as a test of our proxy. Not to test if the proxy actually works. That we could more easy do with a test suite but more to check if the data the proxy API return is sufficient and in an efficient format. Once you actually start to implement the client you suddenly realize that you are going to need data the proxy doesn’t return and vice
versa you send data that you thought you would need but as it turns out is just a waste of bandwidth.
For these two reasons we are going to implement the client web page side by side with the proxy and the server helper, but with a lower priority on the client web page.

### 3.3 Automated server management

For this project we have designed what we think is a very sophisticated server management system. We wanted a system which distributes status update data from the servers to the proxy with the following criteria:

1. **Low manual maintenance.** The management system should run by itself with low or no human input or interaction required.
2. **Low latency for updates.** Changes in the system must be reported to the proxy instantly so that actions can be taken if needed. E.g. the proxy shouldn’t route request to a server after it gone offline.
3. **Low resource drain.** The management system shouldn’t have to send or parse data if it is not necessary.

With these criteria in mind we designed a system where the server helper periodically collects all the changed (i.e. that differs from last update) data from the transit platforms, aggregate this data into one single JSON string and then sends incremental updates to the proxy.

The advantage of this system compared to a simpler system where the proxy periodically query each server for its current data is mainly two things:

1. **Shorter delay for updating changes.**
   
   Since the server helper and the transit platform are running on the same machine the resource cost of querying its current status is much lower than if the proxy should do it. This means that we can query the transit platform for data much more frequently. Ergo the delay for reporting any changes on the server to the proxy will be much lower.

2. **We save resources on both the proxy and the server.**
   
   If the proxy had periodically asked the server helper for the data the server helper had to query the transit platform, parse the data, aggregate the data into one JSON string and then return it to the proxy. Then the proxy first had to parse the JSON string and
then update its current data objects with the new information and in most cases to find out that nothing has changed on the server.

With our design we only send updates from a server to the proxy if something on the server has changed and even then we only send incremental updates i.e. we only send the data that has changed. If the data from Open trip planner has changed but not the data from One bus away we only send the new data from the Open trip planner to the proxy.

Keeping in mind that the proxy is already routing every client request through it so we want to use the available resources as efficient as possible which means no unnecessary use of resources. Once the servers are up and running there should be a long time between each change and needed update since it is very seldom that the status of a server change.

Also with this design we don’t need to know the URL to each server and we don’t need to instruct the proxy to use a new server. All we have to do is deploy the server helper on a new server and the server helper will automatically send data with its current status to the proxy that will register it as a new server.

Finally the proxy then needs some sort of backup system to store the information about all the servers and graphs in so that if the proxy restarts it knows all the servers and can query each of them for their current status data. Note that this is only done once when the proxy starts. All other updates are sent from the server to the proxy at the servers time of choice. As a backup system a simple text file may suffice at the beginning, but we probably should implement the backup system with a real database later.

3.4 Request routing

Routing by geographic location
Now that we have our server management system that keeps track of all the servers with all their different graphs we need an efficient way to route the incoming API request traffic between these servers.

The design we came up with is based upon geolocation where the client sends a geographic coordinate with each API request and that coordinate is then matched against the geographic boundaries of each graph. If the given geographic coordinate is inside the geographic
boundaries of the graph the request is routed to the server that contains that graph. The requested is routed to all the servers that contains graphs where the coordinate is a match i.e. if the given coordinate is inside the geographic boundaries of multiple graphs then the request is routed to the servers of multiple graphs.

**Design choices for the routing algorithm**

Next design choice was how we should perform this geographic bound matching as efficient as possible. Our first idea was just to iterate through all the graphs registered in the proxy and check each one if the given geographic coordinate is inside the geographic boundaries of the graph, but this would be a time inefficient solution for two reasons:

1. The routing would have a linear average time, \( O(n) \), based upon the number of graphs in the proxy. Double the amount of graphs and the average time it would take to find the correct ones would also double.
2. To check if a given geographic coordinate is within the geographic boundaries of a graph is a time and resource consuming operation.

What we wanted was a design that would take more or less a constant average time, \( O(1) \), to find the correct graphs not matter if we have 10 or 10 000 graphs in the proxy and also a design that didn’t have to make those time consuming boundary matches.

**Final design for the routing algorithm**

The design we finally chose consists of a precomputed geographic table representing every possible geographic coordinate on the planet where each graph is placed in the cells that corresponds to its geographic boundaries.

Then when a request is made to the proxy API the geographic coordinate of that request is used to find the correct cell in the geographic table. If there is a graph present in that cell then we route the request to that graph.

With this design we can have a huge amount of graphs in the system without adding any delay to the request routing since a table look-up takes the same amount of time to perform no matter if the cell is empty or if it contains multiple objects. The only difference is that if we find multiple graphs in the cell the proxy routes the request to each of them, but since each call to a graph is done in a separate thread the delay for the client should be unnoticeable.
The reason why we send the request to multiple graphs is so that we get all available data for the given geographic coordinate. Many public transport agencies travel across borders so if we want all the transit data from a given coordinate we need to request it from both graphs of the adjacent countries.

With this design if you request all departures from Oslo train station as an example you get all the Norwegian transit companies departures but you will also get the Swedish transportation company SJs departures from Oslo to Sweden.

We think that this design give the maximum flexibility and usability for the clients using the proxy API. If a client don’t want the information from the neighboring countries it is their prerogative to discard it at their end.

### 3.5 Proxy class hierarchy

Due to the complex nature of the proxy we have decided to have a strict hierarchy of classes where each class have a specific function. Below is a class diagram with descriptions of the classes in the proxy.
Figure 3.2: Proxy design class diagram

Call manager
The call manager class contains all the @PATH notations for the RESTful API and are therefore the entry point of any API request to the proxy.
Whenever a client makes a HTTP request to the proxy API the context path of the request is matched with the @PATH notation on each method in the call class and the corresponding method will be executed.

The reason we have chosen to have the entry point of each API call in the same class is to get a better overview of the entire API in code format. All the logic for each call is done by a
variety of other classes but the start of each of the API calls are all located in the call manager class.

**Database**
This class have the sole purpose of containing all the static data in the proxy. The database class contains the only instances of the server and graph manager classes as well as data object with necessary information for the operation of the proxy. Information like paths to the backup file and the graph repository as well as the context path to the server helper running on the underlying servers. The database class has functionality for storing the current data to a backup file whenever vital data changes.

All calls from the call manager class goes through the database class to either the server manager class or the graph manager class.

**Error message**
The database class contains a list of all the error messages related to the proxy and its underlying servers.

Every time something unexpected or faulty happens on either the proxy or its underlying servers a message with the type of error, its source(proxy or server URL) and a time stamp is created.

**Server manager**
This class contains and manages the list of servers objects representing each underlying server. The server manager offers functionality to modify, add new servers, delete existing servers, change type of servers and receive status updates from the server helpers.

**Server objects**
Each server object is a representation of an underlying server. This object contains the URL to the server and each connection to a server goes through its corresponding server object.

The server object also maintains a list of all the different transit platforms running on this servers and their current status.

**Platform info**
Each server object has a list of platform info objects each representing a transit platform running on that server and that platforms current status. The status consists what type of
platform it is (Open trip planner, one bus away etc.), its online status and its version number, but no graph info is stored here.

**Graph manager**
This class contains and manages the list of graph objects representing each graph running on an underlying server.

The graph manager offers functionality to build new graphs, delete existing graphs, deploy and undeploy graphs on underlying servers, and receive status updates from the server helpers.

The graph manager also contains the list of graph tables and the functionality to find a specific graph based on a API request which is used for the proxy routing. All transit API requests go through the graph manager.

**Graph objects**
Each graph object is a representation of a graph running on an underlying server. This object contains a list of all the servers (server objects) where this graph is deployed and also maintain the statistics of the graph, calls per hour, round trip time etc.

**Server statistics**
The graph object contains a list of instances of the server statistics class that each one contains one server object instance.

Each graph object has a list of all the servers that have that graph deployed. The server statistics class is a wrapper for those server objects with additional data related to the graphs performance on that server i.e. the server statistics class contains the data for the number of calls and the average round trip time a given graph have on a given server.

**Graph table**
The graph table class is a geographically sorted representation of all online graphs running on all the underlying servers for a certain transit platform and is used for the request routing for that transit platform.

Each transit platform has its own graph table in the graph manager class. This is done to increase performance since we don’t have to check the type of each graph when we do the routing since most of the API functions only use one specific transit platform.
Connect thread
Whenever the graph table look-up returns more than one graph the graph manager needs more threads to be able to make all the calls to the underlying servers in parallel. Each new instance of the connect thread class receives a graph object and a parameter string from the graph manager class and then continues to connect to the server of that graph object with the given parameter string.

Once the connection is complete the new thread is joined together with the original thread again and the data can be returned to the client as soon as there is only the original thread left.

3.6 Proxy event class hierarchy
Omitted from the previous proxy class diagram is the event management system of the proxy. Here follows a descriptions of the different class types in the event management system.

![Figure 3.3: Proxy event design class hierarchy](image)

Event manager
The event manager contains a listener list of all the “listener classes” that are interested in updates from the event system. Whenever a “trigger class” calls a callback method that same callback method is called on each of the listener classes in the listener list.

Trigger classes
Whenever a trigger class has an update to report it calls the matching callback method in its event manager instance.

It can be the server manager that wants to report that it has added or removed a server or a server object that wants to report that it has changed the server type or that the server has gone offline.
Listener classes
A listener class is a class that receives updates from the event manager.
Whenever the event manager calls an update callback method the listener class execute the
logic related to that callback.
Lets say that the server offline callback is triggered then the graph manager would remove
that server from the routing table and set all the graphs related to that server to offline.

3.7 Server helper class hierarchy
As with the proxy we have decided to have a strict hierarchy of classes on the server helper as
well. Below is a class diagram with descriptions of all the classes in the server helper.

Call manager
The call manager class contains all the @PATH notations for the RESTful API and are
therefore the entry point of any API request to the server helper.
Whenever the proxy makes a HTTP request to the server helper API the context path of the request is matched with the @PATH notation on each method in the call class and the corresponding method will be executed.

The reason we have chosen to have the entry point of each API request in the same class is to get a better overview of the entire API in code format. All the logic for each call is done by a variety of other classes but the start of each of the API calls are all located in the call manager class.

**Database**
This class have the sole purpose of containing all the static data in the server helper. The database class contains the only instances of the graph manager and platform managers (Open trip planner manager, Open ride manager etc.) classes as well as data object with necessary information for the operation of the server helper. All calls from the call manager class goes through the database class.

**Platform managers (OTP manager, OBA manager)**
Each transit platform running on the server have their own manager responsible for the logic of the API calls to that specific platform.
All requests that require data to be fetched from a transit platform running on the server goes through that platforms manager.

**Graph manager**
The graph manager contains the logic for graph management e.g. building new graphs, deploying and undeploying graphs on the server platforms.
This class contains a list of all the current graph build threads so that a current build can be aborted and the server can be notified when the build is done or has failed.

**Build thread**
The graph manager class contains a list of build thread instances where each build thread object is a new thread running on the server helper with the sole purpose of building a new graph.
Once the build is done this tread should also run the new graph file through a test suite to see if the build was successful. If the graph file passes the file should be moved to the graph repository and if the graph build have failed an error message should be sent to the proxy.
**Connect**  
This class contains a static method for connection to a given URL. This class is used by all the other classes to connect to the server platforms with client requests, fetching current status and to connect the proxy with status updates and error messages.

**3.8 Management web page design**

**General layout**  
For the management web page we wanted a user friendly graphical interface which doesn’t require the user to read a manual before using. All the monitoring and management functions should be available with clear indication of what each of the are representing and their functionality.

To achieve this we went with the “divide and conquer” approach where each specific problem or function in this case is separated from the others with clear distinction. Our implementation of this philosophy is going to be a tabbed web page where each tab contains the graphical elements related to a specific function or group of functions.

**Main tab**  
The main or “start” tab if you like should be an overview of the current status of all the underlying servers to the proxy.

From here an administrator will be able to monitor the status of all the servers with everything from the server URL and server type to if they are online and what transit platforms they have running. We are going to display this data in the form of a table where each tuple is a representation of the status of a single server.

Below the server status table there is going to be an additional table showing error messages sorted by the time in descending order.

This way whenever a problem occurs on either the proxy or on one of its underlying servers that error will be shown on the management page. We are going to color code them so that errors that are recent are highlighted.

**Transit platform specific tabs**  
Each of the transit platforms are going to have their own tab where an administrator can monitor and manage the graphs for that specific platform.
We are going to implement this as a table where we show all the graphs with their list of servers that graph is deployed on. It should be possible to add a server to a graph (i.e. deploy the graph on a new server) and also to remove a server from a graph (i.e. undeploy the graph from a given server).

Additional functionality would be to remove an entire graph (i.e. undeploy the graph from all servers and remove the graph file) and to reload the graph (i.e. redeploy the graph on all servers it is currently deployed on). All these functions should be controlled by clicking on graphical elements in the graph table.

The final functionality would be to build new graphs. This part isn’t possible to do directly in the graph table since it requires text input from the user so it is going to be implemented as a separate part below the graph table.

3.9 Client web page design

Due to the fact that the main focus and priority of this project is on the computer cluster we wanted the client to be as easy and quick to implement as possible without sacrificing functionality. Since neither of us have any experience on developing software on mobile platforms the obvious choice was a web page.

Map service
The main feature of our part of the ITRACT project is the interactive map so the first thing we decided was how we’re going to implement the map.

Almost immediately our thoughts fell on the for us most famous map service, Google maps. On further study it turns out that Google maps has a free to use license for their API up to a limit of 25,000 request per day which works fine for our prototype. If we in the future want to release a version of our product which still uses Google Maps there also exists paid licenses without the 25,000 requests restriction so that is no problem.

Page layout
Once we decided on the map service we looked into the graphical design of the homepage.

We went with a classic look with two visible panels. On smaller column along the left side for the controls and one on the right side that fills out the rest of the screen with the map.
The map panel should just be the Google Maps API map set to fill the entire panel and the control panel should have the from and to field where you can enter the address of a street or the name of a station to set the start and destination of the trip planner.

The control panel should also have field for entering date and time, set different modes of transport etc.

![Control Panel Layout]

*Figure 3.5: Client web page general layout*

**Search fields**

For the from and to fields we have two options. Either we our self do a search function that searches through all the names of the stops. This option have two drawback. One is that we have to implement it our self and that would take time that we rather spend on the proxy and two is that we will then only be able to search for stop names. Since this trip planner support modes like walk and car we would like the option to enter a street address in the to and from fields so that you can plan a trip from your doorstep if you like.

With this in mind we once again turned to Google, specific to their Places API. This API can give you a list of geographic addresses based upon a given input string. This means that when you type it automatically will give you options matching what you are typing.

So if we use the Google Places API we can search on specific addresses and we get character by character suggestions when typing. The Places API has a free license with an upper limit of 1000 request each day which also is fine for our prototype.
So we now have our map and our search functions. We have the basic design for the web page and we have the specifications for the proxy API. That is all we need to start implementing the web page.
4 Implementation

4.1 Introduction

The primary function of the proxy node is to aggregate data from several different computational servers with the explicit purpose of combining the features and functions of all these servers into one unified API for the clients to make their requests against. Inherent in this design is the need to configure, monitor and route request between multiple servers without adding a noticeable decrease in performance or usability for the clients.

How we have decided to implement this functionality will be described in this chapter. Due to the complexity of the management and routing system the description have been divided into several sub chapters each describing a key component in the implementation chain. We have chosen to omit some of the more basics features of the implementation to leave room for a more explicit description of the more sophisticated parts.

Used programming languages and software

For the different parts of this project we have used varying programming languages

1. The proxy and server helpers are written exclusively in Java with JRE version 1.6 compatible code.
2. The Client and management web pages are written in a combination of HTML, JavaScript and CSS.
3. Netbeans 7.3 were used as our only IDE(Integrated Development Environment).
4. Tomcat 7.0 were used as our only web server software.
5. Git-Hub were used for revision control.
Class structure diagram

Figure 4.1: Proxy implemented class diagram
4.2 Automated server management

Proxy data structure
The data related to the current status and management of all the servers is stored on the proxy divided into two different parts:

1. The server manager class contains a list of all the servers used by the proxy. Each server object in the server list is a representation of a given server and each server object contains the data related to that server.
2. The graph manager class contains a list of all graph object from all platform manager class used by the proxy. Each object in the graph list is a representation of a given graph running on one transit platform and each graph object contains the data related to that graph. Graph manager class also contains a list of graph table where every graph table represent one transit platform supported by the proxy. Each graph table object contains information used to find graph object which uses the same transit platform.

Maintaining an updated version of the server and graph lists
We have chosen a system where the primary mode of update for both the server list and the graph lists is that the server helper transmit incremental updates about the servers current status to the proxy with the minor occurrences that the proxy query the server helper for the current server status.

These are the different scenarios where update data needs to be transferred between the server helper and the proxy for the management system and our implementation of them:

1. The server starts when the proxy is offline
   When the server helper starts it transmits its current status to the proxy. Since the proxy is offline the connection fails. This puts the server helper into an infinite loop where it periodically tries to transmit its current status to the proxy This continues until the proxy is online and the connection is successful.

2. The server starts when the proxy is online.
   When the server helper starts it transmits its current status to the proxy. Since the proxy is online the connection will be successful. This puts the server helper into an infinite loop where it periodically check for changes and transmits those changes as
incremental updates.
On the proxy side the status update from the server will be received and the proxy will update its current server and graph object to match the servers reported status. If this is the first time the server sent data to the proxy a new server object will be created based upon the reported server status.
The proxy will save the changes to a backup file.

3. The proxy starts without any servers being online.
The proxy reads the backup file and recreates all the previous server and graph objects and sets the status of all of them to offline. The proxy then request status updates for all the servers in the server list with a HTTP PUT request to each of the servers. Since no servers are online the proxy will receive no status updates and all the server will remain in offline status.

4. The proxy starts with a server online.
The proxy reads the backup file and recreates all the previous server and graph objects and sets the status of all of them to offline. The proxy then request status updates for all the servers in the server list with a HTTP PUT request to each of the servers.
When a server get the update request it will send its current status to the proxy and when the proxy receives the status update the proxy will update its current server and graph object to match the servers reported status.

5. A server goes offline when the proxy is online.
If the server, more precisely the server helper, is shut down in a proper way it will automatically send a shutdown update to the proxy. When the proxy receives the shutdown status update the server and graph objects corresponding to that server are set to offline status.
If the server is shut down in an improper way (i.e. system crash, power outage etc.) there are one way to detect this. Whenever any request is routed to a server from the proxy and the connection fails the proxy set the server as offline and then request a status update from the server.

6. The proxy goes offline when there are servers online.
If the proxy goes offline the whole system is down. As it is right now the proxy is a
single point of failure. To fix this we propose a solution with multiple proxies either running in parallel (which would also reduce the load on each proxy) or one at the time with the other ones as backup to take over if the primary proxy fails.

**Proxy status update management**
The proxy is the centralized node that keeps track of the status information on all servers. When the servers transmit status updates the information is processed by the proxy that compares the incoming status information against its current status of the server. Once the status have been compared the information about the servers current status (is server online? which transit platforms are running? what graphs are loaded? etc.) is updated on the proxy. For most parts the proxy just registers the servers current status, but when it comes to the graphs that are deployed on each server the proxy is the master and decide what graphs that should be deployed. If the reported graphs on the server doesn’t match the expected graphs the proxy will send a HTTP PUT request to the server helpers API to deploy or undeploy graphs on the transit platforms so that the deployed graphs on the server match the expected graphs from the proxy.

**Minimizing data flow**
We want to minimize the status updates sent from the computational servers to the proxy by doing incremental status updates. The server helper detects changes in the server and report those changes to the proxy.

For the implementation of the incremental updates on the server status we choose to split the status information into multiple categories: memory status, graph builder status and one status for each transit platform. The memory status consists of the servers total and free primary while the graph builder status is the list of all graphs currently being built by the server. The status for each transit platform consists of that platforms name (i.e. Open trip planner, One bus away etc.), the software version of the platform and the list of all the loaded graphs on the platform.

To detects changes in the status the server helper check for status updates very frequently and if there is a change in one or more of the different status categories that status category is transmitted to the proxy. How the server helper does this checking is quite simple. It keeps tracks of its current status (i.e. the last status update it transmitted to the proxy) and periodically query each transit platforms for its current status as well as checking its own
status for memory usage and graphs being built. If any of these status categories doesn’t match the last update that was transmitted to the proxy the changed status categories are concatenated into one status update and transmitted to the proxy.

The server helper transmits updates for the slightest changes in status for all categories except memory. Since the amount of free memory changes very frequently we only send updates if the change is greater than 10% of the last reported value. If any of the other status categories have changed we always update the memory status as well since it requires so little bandwidth and computation resources.

4.3 Request routing

Graph table
As mentioned in the design chapter the request routing is done by table look-ups in a pre-calculated table. Here we are going to explain the structure of the graph table and how it is used for the routing.

All graphs running on all the computational servers for a given transit platform are stored in that transit platforms instance of the graph table class.
Each graph table class contains a three-dimensional graph table object that represents every geographic coordinate on the planet and the graphs for that coordinate.

For our implementation the graph table object is constructed by a two-dimensional array of ArrayList objects:

```
ArrayList<GraphObject>[]][] graphTable;
```

Figure 4.2: Routing graph table code

The reason for mixing Arrays and ArrayList is that the two first dimensions together make up a geographic table where the first dimensions is and index representation of a longitude value and the second dimension is an index representation of a latitude value and both of these dimensions together represents every geographic coordinate on the planet. This table always has a fixed size since the size of the planet(or more exact the geographic coordinate system used to describe it) is constant.

The third dimension is the list of graph objects with transit data for the geographic coordinate given by the two first dimensions and that list changes in size hence the use of an ArrayList.
So even tho in reality the graph table is implemented as a three-dimensional vector we choose to view it and describe it as a two-dimensional geographic table where each cell represents a geographic area. That cell then has a one-dimensional list of all the graphs for that region.

**Convert geographic coordinate to table index**
The geographic coordinate system defines each location on the planet by a longitude and latitude coordinate where:
Longitude ranges from -180° to +180°.
Latitude ranges from -90° to +90°.

For our geographic table(i.e. the two first dimensions of the graph table) we have chosen a cell size of 0.5°. This means that each cell represents a geographic area with the dimensions 0.5° longitude by 0.5° latitude. This is an arbitrary number and further studies have to been done to determine the optimal cell size.

To convert a coordinate to a table index we use the following formula:

\[
\text{Index} = \frac{(\text{coordinate}° - (\text{min}°))}{(\text{cell size}°)}
\]

*Figure 4.3: Coordinate to index formula*

The formula can be simplified as:

\[
\begin{align*}
\text{Longitude index} &= ((\text{coordinate longitude}°) + 180) \times 2 \\
\text{Latitude index} &= ((\text{coordinate latitude}°) + 90) \times 2
\end{align*}
\]

*Figure 4.4: Coordinate to index formula simplified*

That gives us that the following:
1. Coordinate (longitude -180, latitude -90) have table index: [0][0]
2. Coordinate (longitude 180, latitude-90) have table index [720][360]

Before we explain the process of adding graphs to and retrieving graphs from the geographic table we need to explain a few prerequisite knowledge’s.
**Polygon bounds**
Each graph object has a sorted list of geographic coordinates. These coordinates are sorted as a closed cycle that outlines the geographic boundaries of the graphs transit data i.e. the graph only has transit data for the geographic area inside the geographic bounds.
We call these bound the polygon bounds because if you draw a straight line between each coordinate you would get a two-dimensional figure constructed of a closed chain of straight lines i.e. a polygon.
The geographic polygon is always convex i.e. every internal angle is less than or equal to 180°.

**Box bounds**
The box bounds is just like the name suggest a box, or rectangle if you want, surrounding the utmost geographic coordinates of a graphs transit data. The box bounds are extrapolated by the graph table class from the minimum and maximum longitude respectively latitude of the polygon bounds.
Below you can see an image of the geographic bounds of the Swedish graph.

![Swedish graph geographic bounds](image)

*Figure 4.5: Swedish graph geographic bounds*

**Ray casting algorithm**

To test if a given coordinate is inside a given polygon we use an algorithm called “Ray casting”.

This algorithm says that a given coordinate is inside the polygon if a ray passing from the exterior of the polygon to the coordinate has an odd number of intersections with the polygon and if the number of intersections are even the coordinate is outside the polygon.

To use this algorithm we essentially need five components:

1. The coordinate to test
2. A coordinate we know is outside the bounds.
   We use the a coordinate with latitude 91 since the latitude scale only go to 90 that coordinate is always going to be outside the polygon.
3. A polygon to test against.
4. A line intersection/collision method.
5. Integer that keeps track of the number of intersections.

We check for intersections between the “outside coordinate > test coordinate ray” and all lines in the polygon (A line in our polygon array is a pair of adjacent coordinates) if the total number of intersections are odd then the test coordinate is inside the polygon.

Pseudo code:

```plaintext
num_intersections = 0;
ray = outside_coordinate, test_coordinate;
for each(coordinate pair in polygonBoundsArray) {
    line = polygon_coordinate1, polygon_coordinate2;
    if ( isIntersection(ray, line) ) //True if there is an intersection between the ray and the line
        num_intersections++;
}
return (num_intersections % 2 == 1); //Returns true if the number is odd.
```

*Figure 4.6: Ray casting pseudo code*

**Finding matching table cells for a graph**

To find which cells in the graph table the geographic bounds of a certain graph correspond to we use the “getCellIndexList(GraphObject)” method in the graph table class. This method takes a graph object as an argument and based upon the geographic bounds of that graph an ArrayList containing the index values to all the matching cells are returned.

The following procedure is used to construct the ArrayList of matched cells:

1. Create an empty ArrayList that will contain the indexes of all cells that match the geographic polygon bounds.
2. The box bounds are extrapolated from the polygon bounds.
3. Convert the box bounds coordinates to table indexes.
4. Loop through all the table cells within the box bounds matching each cell against the polygon bounds. The matching is done by the following procedure:
   1. Calculate the center position of each cell and convert that value to a geographic coordinate.
2. Call the graph objects method “isCoordinateInsideGraph(Coordinate)” passing the center coordinate as an argument. This method does the ray casting algorithm and return true if the given coordinate is inside the polygon bounds.
   If true add the index values for this cell to the ArrayList of matched cells. Then continue to test the next cell within the box bounds.

3. If the isCoordinateInsideGraph method return false test each of the four border lines in the box bounds for intersections against the polygon lines with the graph objects “collision(Coordinate, Coordinate) method. This method return true if the line created by the given coordinate set intersects with any of the lines in the polygon bounds.
   If any of the four tests return true add the index values for this cell to the ArrayList of matched cells. When all four collision tests are completed continue to test the next cell within the box bounds.

5. Once all cells within the box bounds have been tested the loop is done. Return the ArrayList with the matched cell indexes.

Below on the left side of the page is shown an image of the Swedish graph with its geographic bounds and an overlying grid. Each cell in this grid is roughly 0.5° longitude by 0.5° latitude and is used as a representation for the graph table.

On the right side is an image of the same graph but with color coding to show the results of the cell matching done by the getCellIndexList method.

The blue cells are all the cells within the bounding box that was not within the polygon bounds and are therefore all the cells that have been tested but that will not be returned in the ArrayList.
The red cells are all the cells that were within the polygon bounds and are therefore all the cells that will be returned in the ArrayList.

**Figure 4.7: Routing graph table result**

**Adding new graphs to the graph table**
Whenever a new graph object is added to the graph managers graph list or whenever an existing graph object changes status from offline to online that graph objects is added to the graph table for the corresponding transit platform through the following procedure:

1. The graph manager calls the “addGraph(GraphObject)” method in the graph table class passing the graph object that is to be added as an argument.
2. The addGraph method calls the getCellIndexList method passing the given graph object as an argument.
3. The getCellIndexList method return an ArrayList of all the index values for the cells that match the geographic bounds of this graph.
4. Loop through the cell index ArrayList and add the graph object to each one of the cells in the graph table described by the cell index ArrayList.
Using the graph table for request routing
Whenever a client makes a request to the Proxy API that requires data to be fetched from one of the graphs on the underlying computational servers the graph table is used to find the correct graph/graphs.
Since the table is pre-computed whenever a new graph is added there are no bound matching or resource consuming algorithm required to find the correct graph. All that is required is that the API request comes with a geographic coordinate. The “connect” method in the graph manager class calls the “getGrapsFromPos(Coordinate)” method in the graph table class passing the request coordinate as an argument. That geographic coordinate is then converted to table indexes with the following formulas:

\[
\begin{align*}
\text{Longitude index} &= ((\text{coordinate longitude}°) + 180) \times 2 \\
\text{Latitude index} &= ((\text{coordinate latitude}°) + 90) \times 2
\end{align*}
\]

*Figure 4.8: Coordinate to index formula simplified*

Once the indexes for the matching cell is calculated the entire ArrayList in that cell is returned to the graph manager.

Multiple graphs in one table cell
As previously stated the entire ArrayList for the matching cell is returned from the graph table to the graph managers connect method. There is a possibility that this ArrayList contains more than one graph object. This happens when two or more graphs contain transit data for the same geographic region which isn’t unusual for neighboring countries or regions that have traffic going across the border.

If the returned ArrayList does contain more than one graph object the following occurs:
1. The “main” connect thread creates a new thread for each graph object in the ArrayList except the last. These new threads are created by creating a new instance and passing the graph object as an argument to the “ConnectThread” class which implements the runnable interface to create a new thread for each instance of the class.
2. Each instance of the ConnectThread class is stored in a list called “connections”.
3. Each new thread calls the “connect” method of the given graph object which connects to a server that holds the graph that the graph object represents.
4. When all the new threads have been created the main thread connects to the last graph.
5. Once the connection to the last graph is terminated the main thread joins all the newly created connect threads back to the main thread. This can only happen if the connection of each graph is done so the main thread have to wait for each of the connect threads to terminate.

6. Once all the connections have finished the answers from each connection are stored in a list paired together with the graph ID of the graph that returned the answer.

7. The list of answers is serialized into a JSON string and returned to the client.

The import thing to understand is that the client gets back a list of answers. One answer for each graph that have transit data for the given geographic coordinate.

**Redundant servers for one graph.**

Since each graph can be deployed on multiple servers each graph object has a list of all the servers where this graph is online.

When the connect method of a graph object is called a connection is set up to one server that contains that graph. Which server to connect to is chosen by a weighted randomizer function. A weighted randomizer function is a function that randomize a value where not all available options have an equal probability of being chosen. The algorithm is constructed such that servers with a lower round trip time will have a higher probability of being chosen than a server with a higher round trip time. There is still a possibility that a server with a higher round trip time is chosen, it is just not that probable.

In most cases the server with the lowest round trip time will be chosen. This is the server that has the average shortest response time and therefore has the highest probability of giving the client the answer the fastest.

This is done both to even out the load of the servers and to keep the request time a short as possible for the clients. The reason that we sometimes choose a server with a higher round trip time is so that we don’t instantly overload the server with the lowest round trip time. With this design we still spreads the request among all the server, but with a weighted probability for the servers with a lower round trip time.

The round trip time is a weighted moving average that is updated for each routing request.
4.4 Event management

The event driven part of the proxy was introduced to give a more streamlined flow of data through the program architecture.

![Event management class diagram](image)

**Figure 4.9: Proxy event implementation class diagram**

**Event manager**

The event manager class have incorporates the server event and graph event interfaces. These two interfaces represents all events that can be triggered from trigger classes. The event manager then keeps lists of the listener classes that wants updates for each interface event. Whenever a trigger class triggers an event the respective method on the event manager is called and this method in turn calls the corresponding method on all listeners classes.

Example code for the server added event callback:

```java
serverAddedEvent(ServerObject server) { //Trigger class calls this event
    for each (ServerEventInterface listener) {
        listener.serverAddedCallback(server);
    }
}
```

**Figure 4.10: Proxy event callback example**
**Event trigger classes**
Triggers classes are classes that use the event system to inform that an event has occurred that other classes need to know about.

When an event is triggered the changed data (could be an updated server or graph object) is passed along to all the listeners to inform them of the change that has occurred.

Every class that wants to initialize an event callback needs to keep an instance of the event manager typecast with the event interface that keeps the callback that is gonna be used.

**Event listener classes**
Event listener classes are classes that requires updates when the data on the trigger classes change. Each event listener class needs to incorporate an event interface and then register as an event listener to a that event interface with the event manager.

Event listener classes wait on events that is pushed out from the event manager class. When an event callback method is called the data related to the event is passed as an argument with the callback method.
5 Results and system evaluation

5.1 Introduction

At this stage all the implementation is done and we are going to show you the results of our labor.

First out we have our two web pages. These we are going to show through a graphical perspective where we show you an image of the web page and the then explain all the different graphical elements on each web page.

Last out is a testing of the distributed computing system. Here we are going to show you that our outing and load balancing algorithms work and their performance.

5.2 Client web page result

The client web page is our prototype client application based upon the idea of an “interactive map”.

Figure 5.1: Client web page result

The main graphical feature is the map. The map shows all the stops in the area and if you click on a stop an information window will be shown with the upcoming departures for that stop.
On the left side we have the controls for the trip planner. On the top we have the departures and arrival locations. These search fields accept everything from stop names to street addresses. Once a location have been chosen the map automatically pans to that area and displays a marker that indicates that the area is used as either the departure or arrival location.

Below the search fields or the options for when you want to depart or arrive at your location. The maximum length you are willing to walk, your average walk speed and at the bottom is the button for planning the trip.

Once the proxy returns a planned trip there are multiple itineraries to choose from with the “cycle itineraries” button.

This is an example of what an itineraries may look like. If you click on one of the stops you will get the planned arrival and departure time for that stop.

Figure 5.2: Client web page trip plan result
5.3 Management web page result

This is our management web page where we can monitor and configure the system.

<table>
<thead>
<tr>
<th>URL</th>
<th>Type</th>
<th>Online</th>
<th>Free RAM (MB)</th>
<th>Total RAM (MB)</th>
<th>Platform</th>
<th>Version</th>
<th>Edit</th>
<th>Remove</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://192.168.1.65:8080">http://192.168.1.65:8080</a></td>
<td>Proxy</td>
<td>true</td>
<td>7490</td>
<td>12231</td>
<td>OTP</td>
<td>0.9.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="http://192.168.1.65:8080">http://192.168.1.65:8080</a></td>
<td>Routing and build</td>
<td>true</td>
<td>7475</td>
<td>12231</td>
<td>OTP</td>
<td>0.9.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="http://192.168.1.64:8080">http://192.168.1.64:8080</a></td>
<td>Routing</td>
<td>true</td>
<td>2811</td>
<td>11930</td>
<td>OTP</td>
<td>0.9.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="http://192.168.1.68:8080">http://192.168.1.68:8080</a></td>
<td>Routing</td>
<td>true</td>
<td>1122</td>
<td>7636</td>
<td>OTP</td>
<td>0.9.2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time</th>
<th>Server URL</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-05-05</td>
<td>PROXY</td>
<td>Failed to delete graph file with ID: newGraph_otp</td>
</tr>
</tbody>
</table>

*Figure 5.3: Management web page overview result*

On the top of the web page you have the tabs to choose different parts of the system to manage. The idea was that the first tab should be the management of the proxy and its servers and that the following tabs should be one for each transit platform in the system.

**Proxy manager tab**

If we start with the “proxy manager” tab, the main feature is the table of all the servers in the system. From here we can monitor the data and status related to each server. What are their address? what type are they? are they online? how much memory do they have? which platforms are online? and so on. The status of the servers and its transit platforms are color coded. Any problem with the system will show in red.

From here we can also edit the type of each server and remove unwanted servers.

Below the server table is the table of error messages.

Here are all the reported errors that have occurred in the system listed in descending order based on the time when the error occurred. From we can see when the error happened, “who” reported it and what the error was about. The timestamps are color coded so that new errors are noticed.
Transit platform manager tabs

This is an example of a transit platform tab. This is the tab for the Open trip planner manager.

![Figure 5.4: Management web page graph result](image)

From here we can monitor all the graphs in the graph repository related to this transit platform.

Each graph in the repository is shown as a table with the graph ID listed in gray. To the right is the total average calls per hour and the total average round trip time for this graph. Further to the right are the controls for removing (i.e. undeploy the graph on all servers and remove the graph file) the graph, adding servers to the graph (i.e. deploying the graph on new servers) and to reload the graph (i.e. redeploy the graph on all servers it is already deployed on).

Below the graph header is the list of servers where this graph is deployed. These servers are color coded to show if the graph actually is online or not. An orange server is a server that is currently building this graph file. To the right of the server URL is the average calls per hours and average round trip time for this graph on that specific server.

At the bottom of the web page we have the option to build new graphs. Enter the URL to the GTFS source and the desired graph ID and press the “Build graph” button and the command to build this graph will be sent to the proxy.
5.4 System testing

Here we are going to display the measured performance of the system. Since this product isn’t yet finished and we at this point don’t have access to a server environment capable of housing our product the data we have is somewhat limited.

Test setup
We constructed a test web page that request all the graphs and all their stops from the proxy API. The test page then chooses two stops at random from each graph and plan a trip between the geographic coordinates of these stops. This procedure is then repeated until the program is terminated by the user.

We have two different performance parameters that once the test is done are read from the management homepage.

1. Average calls per hour: This is the total amount of requests divided by the total test time.
2. Average round trip time: This is a weighted moving average round trip time calculated from the time that the request was routed to the time answer has been returned. i.e. this is both transmission and computational delay.

Computer hardware
For this test we was running and testing our product on a makeshift computer cluster consisting of the computers we had at our disposal capable of running these software.

<table>
<thead>
<tr>
<th>ID</th>
<th>CPU</th>
<th>RAM</th>
<th>OS</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workstation1</td>
<td>Intel Ivy Bridge i5 4.5GHz</td>
<td>16GB DDR3</td>
<td>Windows8</td>
<td>Proxy Test web page</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Server helper OTP</td>
</tr>
<tr>
<td>Workstation2</td>
<td>Intel Sandy Bridge i7 4GHz</td>
<td>16GB DDR3</td>
<td>Windows7</td>
<td>Server helper OTP</td>
</tr>
<tr>
<td>Laptop</td>
<td>Intel Core2Duo 2.66GHz</td>
<td>8GB DDR2</td>
<td>OSX 10.8</td>
<td>Server helper OTP</td>
</tr>
</tbody>
</table>

*Table 5.1: Test hardware description*
Results
First we start with a small graph covering the San Francisco bay area. This graph has approximately 200 stops.

<table>
<thead>
<tr>
<th>Computer ID</th>
<th>Average calls per hour</th>
<th>Average round trip time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workstation1</td>
<td>968</td>
<td>98</td>
</tr>
<tr>
<td>Workstation2</td>
<td>885</td>
<td>85</td>
</tr>
<tr>
<td>Laptop</td>
<td>95</td>
<td>464</td>
</tr>
<tr>
<td></td>
<td><strong>649</strong></td>
<td><strong>216</strong></td>
</tr>
</tbody>
</table>

*Table 5.2: Test bay area result*

Next is a larger graph covering the whole of Sweden This graph has approximately 54000 stops.

<table>
<thead>
<tr>
<th>Computer ID</th>
<th>Average calls per hour</th>
<th>Average round trip time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workstation1</td>
<td>1950</td>
<td>2195</td>
</tr>
<tr>
<td>Workstation2</td>
<td>1866</td>
<td>2286</td>
</tr>
<tr>
<td>Laptop</td>
<td>1326</td>
<td>2659</td>
</tr>
<tr>
<td></td>
<td><strong>1714</strong></td>
<td><strong>2380</strong></td>
</tr>
</tbody>
</table>

*Table 5.3: Test Sweden result*

About 36% of the planning request was successful i.e. the trip planner found an available itinerary between the given locations about 36% of the time.

Result evaluation
Due to the limited amount of data available it is impossible to make any final evaluations on the system performance. What we can do is to say that the routing and load balancing algorithms appear to be working. All trips planning request are made to the same proxy API with just the coordinates of two different geographic locations and we can clearly see that all the graphs on all the servers have been used.

When the test page used two geographic coordinates in Sweden the request was routed to one of the servers containing the Swedish graph and when the test page used two geographic coordinates in the San Francisco bay area the request was routed to the bay area graph. We can also see that the redundancy works because for both graphs all three servers were utilized.
So the geographic routing and the redundancy is working. What about load balancing then? This is a little trickier since we are lacking data about the resource utilization on each machine. What we can see is that the two workstations that have more or less equal available hardware resources have about the same calls per hour and about the same round trip time and that the laptop which have significantly less available hardware resources have fewer calls per hour and higher round trip time. As it is right now you can clearly see that the load balancing algorithm is running since in both graphs the server with the least available resource got the least amount of requests.

The goal should be to have the same round trip time on each server so the algorithm probably needs some fine tuning in the future, but that can only be done once a real test suite is created with much more available data.

### 5.5 Summary

All the primary parts of the system is up and running.

1. The client home page is working. You can see all stops in a given area. If you click on a stop the departure times for that stop is displayed. You can plan a trip both by clicking on the map and by typing a stop name or a street address.

2. The management web page is working. The status of each server is monitorable from the proxy manager tab. You can change the type of a server and remove unwanted servers. The graphs and their status for each transit platform is monitorable from that transit platforms tab. You can deploy and undeploy graphs, build and delete graph files.

3. The routing and load balancing algorithms appear to be working. All transit calls are directed to one API and based upon the geographic coordinate in the request the routing is done. Redundant graphs have their request distributed among the servers with load balancing based upon the servers round trip time.
6 Conclusion

6.1 Project evaluation

Now at the end of the course we look back at all we have done during the length of this project and evaluate how we think it went by asking our self questions like: Did we achieve our goals? Where there any unforeseen complications and if so how did we overcome them? What have we learn during this course? During this chapter we will try to answer all these questions and more.

This project turned into such a huge undertaking relating to so many different facets of computer science that it is hard to grasp all the different aspects of the project. We feel that to really understand what we wanted to do and what we did we would have to write twice the amount of pages that we did for this dissertation but there just was not time or space for it. We hope that what we did write showed the core and the essence of the project and that you the reader now have an understanding and hopefully an appreciation for the work we did.

Just the sheer size of the project makes it hard to write this chapter without it turned into a dissertation of its own but we will give it our best to keep it to the essentials.

We are personally really happy and satisfied with the work we have done. We adapted to radically changing circumstances and solved any complication that arose during the time span of the project with what we think turned into an elegant and sophisticated design.

There is still a lot of work still to do, but that was expected. Even tho both members of the group put in more time than expected for this project there was just too much to do. There was no feasible way that the two of us were going to get it all done with the time and resources we had at our disposal, but the general design and the framework for future development is in place and we think that the work we did do is of high quality and up to par with what is required for the finished goal to be realized.

6.2 Problems and complications

There occurred a lot of different problems and complications during the time span of this project for various reasons. Some of them occurred because we lacked the knowledge and
experience needed to do the task we tried to do and some of them was due to changing circumstances and dependencies on other people and some was just plain bad luck. Here we are going to go through some of the major problems that occurred and explained how we overcame these predicaments.

**Changing circumstances**
The largest inconvenience and drain on resources of this project was probably the radically changing requirements and circumstances regarding our product. We went from a project focus on makes one homepage running on one machine to implementing our own distributed computing network with our own routing algorithms. This radical change was not feasible with our original design, and halfway through the project time span we literally had to discard and rewrite both our already implemented code and this dissertation. This set us back approximately five-six weeks in our time plan but with hard work we managed to get it all done and ready for a demo before Värmlandstrafik and their IT-partner Hogia in just under three weeks.

So it took three weeks of really hard labor to get the new design made, implemented and documented but it was in our perspective worth it and we are very satisfied with how the new design and its implementation turned out.

Also about halfway through the project the ITRACT team(i.e. all the students working with ITRACT related projects and their supervisors) decided that we all should use a common revision control system for our source code. We had earlier been using a SVN server which we are used to and now needed to switch to Git-Hub which we had no prior experience with. This caused some problems with how we were going to interact Git-Hub with our IDE and how to move our current projects from SVN to Git-Hub. This problem was solved by searching the internet from information and some old fashioned trial and error before the whole system was up and running again.

**Lacking experience**
In some areas of this project we were lacking the prior experience needed to make to make educated estimates and decisions which lead to instances where our only option was to learn by reading and testing to understand how the specifics of different software work. Due to the time restraints of this project these learning curves were quite steep sometimes. Systems, software and formats which we had lacking experience with and that needed research before we could start to use was:
1. Git-Hub (Revision control)
2. Tomcat (Web server software)
3. Maven (Build automation tool)
4. GTFS (Transit data format)
5. JSON (Data transfer format)
6. XML (Markup language)
7. JavaScript (Programming language)
8. CSS (Style sheet language)
9. Open trip planner (Public transit platform)
10. One bus away (Public transit platform)
11. General distributed computing and computer cluster design

All of the aforementioned software and formats were areas where we had no or lacking knowledge about. So before or during our design and implementation of this project we had to do research and testing to gain the knowledge needed to use these software and formats in a correct and efficient way.

**Lacking computer resources**

During the entire time span of this project we had had a problem with lacking computer resources to test our software on. The core of our project is the distributed computing network in which we have implemented the entire structure and all algorithms from our own design. Due to our inexperience and the fact that we are using algorithms we ourselves came up with to not actually have a system to test our software on during the implementation phase was an inconvenience.

There was people working on deploying a virtualized server environment able to support our project in Karlstad University, but due to circumstances that we are not completely knowledgeable about the deployment of the system was delayed and we didn’t get access to the system until the very end of the project when all the design and implementation parts were already completed and even then there was a lot of performance and stability issues with the system which made it unsuitable for any reliable testing of our product.

During the project we made our own makeshift computer cluster with all the physical computers we had at our disposal. We were running our software on a variety of stationary and portable computers at home so we at least had something to test the software on.
This setup was sufficient to determine if the system actually worked, but was somewhat inept to determine how our routing and load balancing algorithms worked and with what performance.

**Technical difficulties**
The technical difficulties we encountered have almost all of them been related to our IDE Netbeans and its synergy with other software like Tomcat, Maven and Git-Hub. A part of the problem is probably due to the fact that we lack experience about these software, but we believe that there are some instability in the Netbeans software especially together with Tomcat. We had problems that when we change something in the code and want to deploy the changes on the Tomcat web server that the serve just froze. Other problems was that Netbeans was not able to shut down the Tomcat server so when we exited Netbeans we had to manually terminate Tomcat from the task manager. Instability issues like this was a big problem when they sometimes occurred quite frequently.

We have both have preexisting experience writing Java software with Netbeans and neither of us have had these kind of problems before. The whole “web API” aspect of the project made it harder and less efficient to implement the code in our experience.

### 6.3 Future work

As we said before there was no possible way that we were going to be able to finish the product during the length of this course so there are several things left on our “to do” list. We are here going to list the parts of the project that we didn’t have time to complete and shortly describe what needs to be done and possible ways to do them.

**Build client applications**
Since all our time went into the computer cluster our client web page was left in a prototype state. Either this web page needs to be continued or a new one created.

In excess of a web page there also needs to be created applications for mobile platforms.

**Implement security/authentication**
Security is needed for this project. We can’t have a scenario where unauthorized people can monitor or especially configure the system. The whole management API need to be secured. An option for doing this can be HTTPS.
**Build management application**
The current management web page fulfill its purpose but is lacking somewhat in functionality. We propose to replace the management with a x86 application. If a .NET applications is a valid solution then a suitable application could be made with C# in a short matter of time.
The pros of an application instead of a homepage is that it would be easier to implement extended functionality. We would like data tables where you can sort the tuples based on any column data and to deploy, undeploy and move graph between server just with drag and drop.
In addition to the current functionality we would like the option to change the graph builder and Open trip planner settings from the management applications in a user friendly way.
Instead of manually editing text we would like the functionality to be more object based. You add/remove the options/beans you like with the given parameters and the program then constructs an XML document out of these options that the programs read. The whole idea is that the user should never have to see it has XML just options in a list.
This application should also be able to display graphical curves showing the statistics of each graph and server over a given time span.

**Export statistics to a database**
Periodically export statistics and current statuses of resources of each node in the computer cluster to an external database. This database is then used to display the statistics. This database would be used by the management homepage/application.

**Create test suite**
Once the product is finalized and we have a real server environment to run it on the systems performance needs to be tested. We propose that we create a test suite application or if the management application exists then that we implement a test suite into the management program.

This test suite should be able to run a series of planning test with varying difficulties to the proxy.
Test like: 1 trip planning per second for each graph, 10 trip plannings per second for each graph, 100 trip plannings per second for each graph etc.
When the test is running each node in the network is then logging(preferably to a database, see previous title) their performance and resource usage.
Once all the different tests are done we can evaluate the performance on the system, make changes/optimizations, run the test again, evaluate and so on.
**Build graph revision control system**
Create a system where you can build a new instance of a graph already existing in the repository. This should not overwrite the existing file but create a new with a higher versioning number.
Once the new graph have been tested and verified all servers having the older version should be instructed to undeploy the old graph file and deploy the new one.

**Integrate ridesharing**
Once the other student working with the ridesharing have completed their work it needs to be implemented into our computer cluster and use our proxy as a front for their API.

**Integrate demand responsive transport**
Once the other student working with the demand responsive transport have completed their work it needs to be implemented into our computer cluster and use our proxy as a front for their API.

**Integrate transit platforms into the server helper**
Right now the proxy calls the server helper that then calls a transit platforms. Since both the server helper and the software is running on the same local Tomcat instance this is a waste of resources.
With the current design a new thread is created on the server helper for each API call, the server helper then calls the API of the server platform with an HTTP request and a new thread is created on the transit platform.
Our idea is to extract the classes from the transit platforms and add them into the server helper directly. This way we don’t do any unnecessary HTTP request or create unnecessary threads. It would also give us the option to extend the functionality between these software. e.g. as it is right now Open trip planner reads an XML file at start-up with the path to the graphs and this path then can’t change. If we integrate the software we could have a system where the proxy gives the full path to the graph file on each deploy request.

**Create a graph deployment pool**
With the current design each graph object has a list of all the servers where this graph should be deployed. If the server goes offline and there are other available server we don’t deploy the graphs automatically on a new machine. The reason why we did it this way is because
different servers could have different resources and we didn’t want to indiscriminately deploy graphs on new servers.

The server environment that is being built in Karlstad University run all servers virtualized. This means that each machine can have the same resources or be given resources dynamically depending on load. With this kind of design it would be a good idea to not have each deployed graph and a server tied together. Instead we propose that we build a graph pool. This pool will be a list of all the graph that should be deployed. If a server goes offline those graphs can immediately be deployed on the first available server and also if a new server connects to the proxy that server will get the graphs the graph pool says needs to be deployed and not what the proxy remembers it had before i.e a specific server doesn’t need to have the same graphs after a restart.

**Update the proxy backup system**
Right now the proxy saves all its info into one text file and we do full backups to this file. This probably needs to be enhanced. Either with multiple files for different areas so we can do incremental updates or better yet a database which contain all the necessary data.

**Gather GTFS data**
We think that the hardest part of this project isn’t actually to write the code but to gather the needed GTFS data for the program to use.

The undertaking of retrieve updated transit information in GTFS format for all public transportation agencies in Europe is close to impossible with today’s communication infrastructure.

What is needed is that all the public transportation agencies in the same country come together and form a shared company that is responsible for aggregating all the transit data from all these companies. It is from these companies that ITRACT then can get their information.

We think that is someone have the notion of gathering all this data directly from each agency they are going to have a very hard time.

**6.4 Andreas personal reflection**
This course have been and interesting and learning experience in which I’ve greatly expanded my knowledge regarding programming and software development in general. I feel that this
The project required a lot of hard work and effort, but I think it was worth it. The hardest part was probably right after we decided to rewrite our design. The following weeks I worked between 60 and 90 hours per week to get the new design ready for a scheduled demo and when the date came to our great relief the program was up and running and worked without a glitch.

My main focus when starting this course was just to get as good grade as possible, but during the time span of the course I discovered a real passion for this kind of work and a pride in the work I’ve done. I personally feel that the work I’ve done is really good and I’m proud of it and apparently so did my project supervisor as well because I got a job offer to continue my work with ITRACT at the university after I’m finished my education which I accepted.

6.5 Anders personal reflection

This project has been fascinating to work on and see how all different parts of projects should fit together to create one complete product. This time spent on the project has been most well spent time of the my education.

I have spent more time on this project than I planned from the beginning of the course. In the start of the project I did the hour required but when we decided to redo the design of the proxy the hour just run away on the project. This was for of two reason one the project was interesting and the second was we wanted to get it ready before a scheduled demo.
7 References