A Trip Planner for the Itract System
Supporting Real-time Updates

Natalie Liden

Faculty: Department of Mathematics and Computer Science
Subject: Computer Science
Points: 30hp
Supervisor: Annika Klockar
Examiner: Thijs Holleboom
Date: June 12, 2014
Serial number: -
A Trip Planner for the Itract System supporting real-time updates

Natalie Liden
Department of Mathematics and Computer Science

Abstract

Mobile applications and real-time data are excellent tools for rapidly sharing information. Such information may concern public transportation, such as time tables and traffic delays. This project has involved the development of a trip planner, which can subscribe to real-time data in order to inform the end user about the position of transit vehicles and trip updates. A trip planner is an application which, after having been given a start and a destination by the user, generates the possible trip between these two locations. The route is displayed upon a map, along with information of how the trip is travelled. The real-time data, which is pushed to the application, will inform the user if vehicles are delayed and if the trip needs to be updated due to a missed bus or train.

The trip planner for Itract developed in this project is using the graphical interface and some necessary Java classes from the open source application Open Trip Planner. The new trip planner, developed in this project, is compatible with the API of Itract, has some additional functionality and can subscribe to real-time information. To subscribe to real-time information, a database called Redis has been set up in connection to Itract. Another database, known as MongoDB, is used for persistent storage.

Keywords: trip planner, Itract, OTP, GTFS, publish-subscribe, Redis
This thesis is submitted in partial fulfillment of the requirements for the Master’s degree in Computer Science. All material in this thesis which is not my own work has been identified and no material is included for which a degree has previously been conferred.

________________________________________
Natalie Liden

Approved,

________________________________________
Supervisor: Annika Klockar

________________________________________
Examiner: Thijs Holleboom
Acknowledgements

I would like to thank my supervisor Annika Klockar for her advice and good counsel. I would also like to give thanks to Andreas Kassler and Robayet Nasim for introducing me to the project and teaching me about Itract. The project has been very interesting to work with. Also, I wish to thank Andreas Arvidsson for being helpful whenever I needed information about the Itract system.

Karlstad, June 12, 2014

Natalie Liden
## Contents

1 Introduction ............................................. 1

2 Background .............................................. 3
   2.1 Project Introduction ................................. 4
   2.2 Itract ................................................ 6
   2.3 The Publish-Subscribe Model ......................... 10
   2.4 Open Trip Planner .................................... 15
   2.5 Summary .............................................. 17

3 Android Application ...................................... 19
   3.1 Trip Planner Module .................................. 19
   3.2 Subscriber Module .................................... 20
   3.3 Trip Planner Implementation ......................... 22
      3.3.1 AsyncTask ....................................... 26
      3.3.2 ItractGraph ..................................... 28
      3.3.3 ItractTrip ........................................ 29
      3.3.4 ItractBusStops ................................... 36
      3.3.5 ItractDepartures ................................ 39
      3.3.6 ItractBusPos ..................................... 40
   3.4 Subscriber Implementation ............................ 42
      3.4.1 GTFS and Publisher ............................... 43
      3.4.2 Positions ......................................... 45
      3.4.3 Trip Updates ..................................... 46
      3.4.4 Alerts ............................................ 49
   3.5 Summary .............................................. 50

4 Intermediary System ...................................... 51
   4.1 Distributed Databases - NoSQL ....................... 53
   4.2 Redis .................................................. 56
      4.2.1 Redis Cluster ..................................... 61
   4.3 MongoDB .............................................. 62
   4.4 Cassandra ............................................. 64
   4.5 Solution ............................................... 65
      4.5.1 Redis Cluster Setup ............................... 70
      4.5.2 MongoDB Setup ................................... 71
      4.5.3 Client ............................................. 72
      4.5.4 Publisher ......................................... 73
   4.6 Summary .............................................. 77

5 Conclusions .............................................. 78
   5.1 Future Work .......................................... 80
1 Introduction

Applications for mobile devices are very useful and popular. These applications are easily accessed if you have a smartphone or a tablet, whether you are at work, in a restaurant or waiting for the bus. Connected to the Internet, information is quickly and easily accessible at almost any place.

One category of information that is the most useful when you can access it anywhere, anytime, is the information concerning public transportation. Such information does, for instance, enable one to find the fastest route from one location to another. One does no longer have to read through multiple train and bus schedules to plan a trip; all that is taken care of by a trip planner application.

One popular trip planner is Open Trip Planner [11], which is available for mobile platforms. By choosing a start and end location, Open Trip Planner will draw a route between these two locations, displaying times of arrival and departure for each vehicle used along this route.

What would happen, however, if the times of arrival and departure changed due to delay or other problems? A trip may be static on the Open Trip Planner application, but it is not so in real life. Therefore, this project has been carried out to make an improved trip planner, which is aware of real-time data such as traffic delays. With the new application, people can be updated while they are waiting for or sitting on a train, and be able to easily modify their trip would they miss the next departure.

An additional task in this project was to find a way to store the real-time data and trip information, in order to keep a history of e.g. traffic delays. The stored data can later be used for statistics on public transportation. By storing data from planned trips, it is also possible to see what trips are most popular and how often they are travelled. Such information can be useful when trying to improve public transportation.

The goal of the project was to make a trip planner that can handle real-time information, displaying such information visually to the user in form of bus positions or trip updates. In the project, the work has been done in connection to the Itract system [6]. Itract is a cloud network containing servers that provide traffic information. The Open Trip Planner application has its own kind of servers and API it communicates with. The trip planner developed during this project is using the graphical user interface and many of the Java classes used in Open Trip Planner, but is able to communicate with the Itract API instead.

The project was handed out by researchers in the Itract project at Karlstad University, who wished to see how mobile applications handle the real-time data sent from the Itract system. They requested a trip planner for Android. An iOS version was a request they gave to another student.

In this thesis will follow some background on the project, Itract and Open Trip Planner. The project itself involved the development of a trip planner and a solution to the real-time updates. These two parts will be given an overview in Section 2.1. The trip planner receives trip data, including real-time updates, from Itract, and the application itself is based on Open Trip Planner; therefore, Itract and OTP will be introduced in Section 2. The model used to get the real-time data transmitted to the application will also be explained in Section 2.
The implementation of the new functionality for the trip planner will be discussed in Section 3. The implementation includes the new classes added to the original OTP application as well as the changes done to already existing classes. The classes handling real-time updates will also be explained.

In Section 4, the solution for storing trip history and handling real-time data will be discussed. While Section 3 focuses on the implementation of the application itself, Section 4 turns to the remote system used to handle real-time updates and how the application connects to this system. Section 4 will provide an explanation of how the intermediary system has been set up to work with the Android application and the provider of real-time data.

In Section 5 is a summary of the results of the project and a section about improvements that can be made in future work.

2 Background

To give an understanding of the project, there will follow a specification of what it involves. Important parts, such as Itract and Open Trip Planner, will be introduced as well. This section is meant to give a background in order to understand the trip planner application itself, whose implementation will be discussed in Section 3. The solution for publishing and storing real-time data will be introduced in Section 4, but the model for publishing and subscribing to these data will be covered here in Section 2.3. The thesis is divided such, because the project has consisted of two tasks: to get a trip planner for the Itract system with the wanted functionality, and to find a system to handle the real-time data and trip history.

First, in Section 2.1, the project will be specified. There are two parts in the project: the trip planner application and a remote system to handle the real-time information that is to be provided to the application. Itract, the cloud network that e.g. calculates trips, will be introduced in more detail in Section 2.2. The important components and their roles will be explained. In order to deliver real-time data to the client application, the publish-subscribe model is used; this model will be described in Section 2.3. Since the trip planner for Itract is based on another application, called Open Trip Planner [11], the latter will be introduced in Section 2.4.

2.1 Project Introduction

The aims of this project were to develop an Android application for trip planning and to implement publish-subscribe functionality for this Android application. A trip planner application is a tool for getting the necessary traffic routes to get from source A to destination B, passing through any number of intermediate stops. The publish-subscribe functionality enables the application to receive pushed data without the need of frequent requests. The publish-subscribe model will be explained in section 2.3.

The Android application requests and subscribes to data which it will use to plan trips. These data are available in the Itract system [6], a cloud network which contains graphs concerning traffic routes. At a request, Itract can return a parsable response to the application. Itract will be discussed further in section 2.2.
To complete the publish-subscribe model, an intermediary system was necessary between the user application and the Itract system. The project can be divided into two parts: Android application for Itract and intermediary system for publish-subscribe.

**Android application** Itract is used by a web-based trip planner application. This project aimed at developing a mobile Android application for the Itract system. The Android application includes both functionality for planning trips based on information from Itract, as well as subscriber software that can receive pushed real-time data. The functionality for trip planning will be based on the Open Trip Planner application [11], discussed in Section 2.4.

**Intermediary system** Itract’s old publish-subscribe system can only communicate with Java applications. In order for Itract to be able to push data to iOS and Javascript based applications, there has to be another system between Itract and these applications. This module, e.g. a push-based server, must be compatible with iOS, Javascript and Android in order to work for both mobile and web-based use cases. In the project, an intermediary system was chosen to solve this issue.

Below, Figure 1 illustrates the Android trip planner connected to Itract via the intermediary system, which is a broker between the publisher and the subscriber. The Android application consists of two modules: the trip planner, which sends requests directly to Itract’s servers, and the subscriber, which subscribes to real-time data through the intermediary system.

![Figure 1: Application, intermediary system and the Itract system.](image-url)
2.2 Itract

Itract [6] is a system that provides end-users with information concerning public transportation. The purpose of Itract is to, by using communication technology, improve public transportation. Information provided by Itract can be time schedules, traffic delays and the location of bus stops. The system aims to support services from different transport providers, to have information accessible for mobile users who use different services and applications. Itract exists in a cloud network, using virtual machines to efficiently share resources.

Itract has a distributed system of virtual machines. Each virtual machine contains software for graph building and trip generation. When a virtual machine has built a graph, it stores it in the repository. There are graphs for different geographical areas, such as Sweden and the Netherlands. These graphs are used by the virtual machines to compute traffic routes. These route data are used to provide the client with a requested trip plan.

The components of Itract are described in more detail below.

Proxy  The proxy takes a request from a client and looks up the graph for the requested trip in a routing table. The proxy then tells the appropriate virtual machine to generate a trip. The graph can be loaded from the repository by the virtual machine if necessary. When the trip generation is complete, the virtual machine sends the trip data to the proxy, which will forward it to the client. The trip information is given in JSON format to the client.

Virtual machines  The VMs, virtual machines, are doing the work of building graphs and generating trips. When building a graph, a VM requests time tables with information such as bus lines, depart times, arrivals and destinations and uses this information to construct graphs for the bus and train routes. The information fetched by the VM is using the General Transit Feed Specification (GTFS) [4] format. This is a format used to provide information about public transportation, including both geographic information and schedules. GTFS also supports real-time information. Once a graph is constructed in the graph builder, it is sent to the repository, so that it is available to all the virtual machines.

A VM can also generate a trip. It does this by loading the needed graph from the repository, then calculates the requested route. The generated trip is given as JSON text and sent to the proxy which forwards it to the client. For example, if an end-user wants to travel from location A to location B, a VM will provide a description of the route and how it is travelled, e.g. 1 kilometre with bus, 100 metres walk and 5 kilometres with bus.

Repository  The repository contains the graphs defining the routes of an area. These graphs are fetched by the virtual machines that use the graphs to calculate routes. There may be, for instance, a virtual machine about to calculate traffic routes for the Netherlands; this VM will then load the graph for Netherlands from the repository. In the repository is also a configuration file. The information in the file is used to determine how often the graphs are to be updated. A graph is updated by a virtual machine requesting GTFS real-time data, connecting to the URLs defined in the configuration file.
**Broker** The broker can be found in the Itract system. The purpose of the broker is to handle real-time data that will be sent to subscribers, according to the publish-subscribe model. The Itract broker can handle over a thousand connections. A subscriber may be a web-based application or mobile applications based on Android or iOS. The original Itract broker is compatible only with Java, so it has been necessary to find an alternate broker system that can handle the subscribers of other platforms, such as iOS.

A virtual machine can also start a publisher when it gets the GTFS real-time data. The publisher provides the broker with real-time data concerning public transportation. The real-time data can be divided into three categories: alerts, positions and trip updates. Alerts are notifications about the changes in the traffic, e.g. if a bus is out of order and thus unavailable. Positions will inform subscribers where buses are located. Trip updates will inform about delays. This information will help end-users to know e.g. how much time they have to wait for the next bus to arrive. Since such information may change at any time, the real-time data must be updated and published frequently, e.g. once every minute.

An overview of the Itract system is illustrated in Figure 2. In the figure there are three virtual machines, used as an example. The machine to the right is getting GTFS data to build a graph, which it stores in the repository. The proxy is getting requests from clients, so it tells the other two virtual machines to generate trip plans. The arrows show the trip plan going to the proxy, then to the client. The leftmost machine is also fetching GTFS real-time data to publish to the broker, which push this information to a subscriber.

There are two clients in the picture, one for the trip plan data and one for the real-time data. In reality, these clients could be two different modules in the same application, as was shown in Figure 1.
2.3 The Publish-Subscribe Model

The publish-subscribe model will be explained in this section. A publish/subscribe system differs from the request-reply system; in a request-reply system, the client transmits a request to the server, whereafter the client awaits a reply. This means that the client has to send frequent requests to keep up with updated information, resulting in an unnecessary large number of requests [12].

In order to transmit data to the client solely when it is necessary, e.g. when the data has been updated, the system can be push-based, i.e. deliver information to interested clients without their need to send any further requests [12]. The publish/subscribe system is a push-based system. This is useful when end-users are to be updated with e.g. traffic delays.

Publish/subscribe systems use content-based co-operation, where the receiver of information defines what type of content it should be sent [12], e.g. what bus line the end-user is interested in. An intermediary system, i.e. a broker or a broker network, will match pub-
lished information with the demands of the potential receivers, and if there is a match, the intermediary system will deliver the information to the appropriate receivers. A sender, the publisher of information, does not need to know the address of a receiver.

In a publish/subscribe system, there are, as mentioned, three significant entities: the publisher, the subscriber and the broker. The publisher transmits data to the broker. Such data may be news, weather reports, schedules or other information the publisher is supposed to share. The virtual machines in Itract contain publisher software, which will publish traffic information, based on the GTFS real-time data, to the broker in the Itract system.

The subscriber sends to the broker a subscription, which defines the information the subscriber is interested in. In the broker, the subscription is matched with the existing information, and if there is a correct match, the information is delivered to the subscriber as a notification.

The broker may hold the published information during a certain period of time determined by the publisher of that information. In this case, the published information has a time to live value. With the broker, the publisher and the subscriber need not be connected directly, nor do they need to be connected to the broker at the same time if data are persistent. The publisher and the subscriber do not need to be aware of the address of one another, only of the broker address.

There may be, and usually are, a multiple number of publishers and a multiple number of subscribers connected to a broker. There may also exist a multiple number of brokers, forming a broker network. A new publisher or subscriber can connect to a publish/subscribe system without changing its functionality [12].

In the case of a broker network, the subscribers must be able to subscribe and obtain notifications from brokers to which they are indirectly connected. When information is published on a broker, the broker sends an update to the neighbouring brokers about the publishers and their publications. The neighbouring brokers will update their neighbours, and finally the entire broker network will be updated if the brokers are connected properly. In this way, the brokers act as publishers publishing the update to the neighbouring brokers. This transmission of updates is called flooding.

Whenever a subscription finds a match and the published information is not on the current broker, the broker will take the role of a subscriber and send the subscription to its neighbours. Once it reaches the broker that holds the information, the subscription will match the publication, which will be transmitted to each proxy subscriber until reaching the original subscriber.

A broker can be seen as a subscriber for its local publishers, and as a publisher for its local subscribers [12]. Thus, the broker can represent its local subscribers via a proxy subscriber, by forwarding subscriptions to neighbour brokers. The broker can represent its local publishers via a proxy publisher by forwarding publications or updates.

To achieve its intended functionality, the publish/subscribe system has a number of functions used by its components [12]:

subscribe(s) is called by the subscriber when the subscriber is about to subscribe to content it has once defined in s.
unsubscribe(s) is called by the subscriber when the subscriber is about to remove a subscription s. It will thus no longer receive notifications matching this subscription.

publish(p) is called by the publisher in order to transmit a publication to the broker. Subscribers will then be able to subscribe to this publication.

notify(S, p) is called by the system when a subscription s matches a publication p. The publication p is thus delivered as a notification to the correct subscriber S.

advertise(a) is called by the publisher to inform with advertisement a that this publisher is about to publish information. Subscribers can thus subscribe to information before it has been published.

unadvertise(a) is called by the publisher to remove an advertisement a.

The following scenario is a simple illustration of how a publish/subscribe system may work. We have one publisher P and one subscriber S separated by two brokers, B1 and B2.

\[
P --- B1 --- B2 --- S
\]

The publisher publishes a publication p on broker B1.

\[
P --p-> B1 --- B2 --- S
\]

When broker B1 receives the publication, it will inform its neighbours that it has received a publication p from publisher P. The proxy publisher pP of broker B1 will send to broker B2 an update of this information.

\[
P --- pP --u-> B2 --- S
\]

The subscriber S sends a subscription s to B2. This subscription matches the publication in the publisher/notification update u.

\[
P --- B1 --- B2 <-s-- S
\]

As a response to the match, the proxy subscriber of B2 sends the subscription s to the appropriate neighbour, in this case broker B1.

\[
P --- B1 <-s-- pS --- S
\]

Subscription s matches publication p, so broker B1 will respond by sending p to the proxy subscriber pS.

\[
P --- B1 --p-> pS --- S
\]

Broker B2 will respond to the subscription from S and send p to S.

\[
P --- B1 --- B2 --p-> S
\]
2.4 Open Trip Planner

The trip planner application in the project is based on existing source code from Open Trip Planner. Open Trip Planner (OTP) is an open source application for planning trips using a client-server model [11]. The application user can select his travels using OTP, and OTP will generate bus or train lines and time tables according to the trip the end-user wishes to take.

The OTP application can be used to request trip data from servers matching the OTP API. The Itract system has its own API, however, and will not work with a regular OTP application.

The OTP application for Android was used in the project to make a trip planner for the Itract system. An overview of the structure of the OTP application for Android is explained below.

The largest class in OTP is the `MainFragment` class, which contains most of the graphical user interface [10]. `MainFragment` is also the listener of the asynchronous tasks, containing the callbacks to be executed when an asynchronous task has finished.

Open Trip Planner for Android uses the asynchronous tasks to request data from OTP servers, so that the user interface will not get blocked in the meantime [10]. Implemented listeners will ensure that the results from the tasks will be returned to the caller when the task is complete. The `TripRequest` class retrieves a trip based on the user's input and the `MetadataRequest` class fetches graph bounds from the OTP server. The graph bounds define the geographical area provided by the server, e.g. a nation like Sweden.

OTPGeoCoding handles the processing of addresses. If the application user types an input in one of the text boxes for start and end locations, the geocoder fetches a list of matching addresses, then, once an address is chosen, places the location marker on the address displayed on the map [10]. If the user puts a start or end marker on the map, the geocoder will ensure that the correct address of this point is displayed in the start or destination text box.

When the application user presses the plan-trip button, a call will be made to `requestTrip()`. To collect data such as times and locations, a `Request` object is created to hold these data. The data include start and end location, traverse modes such as bus and walk, maximum walking distance, and date and time. The code in the `TripRequest` class is then executed, using the `Request` object and its attributes to create a URL string. The URL string is used to get trip data from a chosen OTP-compatible server. The trip data is then processed by the JSON parser in the OTP application to generate a trip plan, which is used to draw routes on the map. The user will be able to see a set of possible routes from start to end location through any number of intermediate spots.

Other classes used for generating a trip plan are `Itinerary`, which defines a trip, and `Leg`, which defines one part of the entire route, e.g. a walk, and `WalkStep` which is a part of the route defined by `Leg`. These classes were used in the project and will be explained in more detail in Section 3.

The OTP application runs the graphical interface, while the trip planning itself is done at a remote server. The application makes requests to the server in the classes `MetadataRequest`
and TripRequest. The response is parsed and used by the application to e.g. draw a route. Figure 3 shows the OTP application communicating with a server using the OTP API.

![Open Trip Planner](image)

**Figure 3: Open Trip Planner**

### 2.5 Summary

The background information relevant for the trip planner should now have been covered. This includes the project specification, an overview of the Itract system, the publish-subscribe model and an introduction of OTP.

The two important parts of the project were to develop a trip planner application for Android and to find an intermediary system that can act as a broker. As mentioned in the introduction of the project, there is open source code for a trip planner known as OTP. This code was used in the trip planner for Itract. More about its implementation will be discussed in Section 3. The new trip planner application was also to have a subscriber, which subscribes to real-time updates. The subscriber will also be covered in Section 3.

Itract and its important modules were discussed. The major components are the proxy, the virtual machines, the repository and the broker. The process of delivering traffic data and routes has been explained. GTFS data is used to construct graphs, and these graphs are used to create routes for different geographical areas. This information is used to plan trips at a client’s request. Data is sent via a proxy before reaching the clients. There is also a GTFS feed for real-time data, which will be used to update the graphs.

The real-time data, e.g. trip updates and vehicle positions, should be pushed to the client to avoid unnecessary requests. The publish-subscribe model is used for this purpose. In this model there is a number of subscribers, brokers and publishers. The publisher publishes data to the broker and the subscriber sends subscriptions to the broker. When the broker can match a subscription and a publication, it will notify the subscriber with the published information. In the case of the trip planner, a publication can be the departure time of a train, published on the channel matching the train’s ID. The subscriber will subscribe to this train, i.e. to the corresponding channel. When the publisher publishes new updates, the broker will push them to the subscriber.

Open Trip Planner was introduced in Section 2.4. The application uses a client-server model to get trip plans at a request to a server matching the OTP API. More about OTP and how its code was used in the project will be discussed in Section 3. Needed functionality for the new trip planner and how it was implemented will be explained in the same section.
3 Android Application

One part of the project was to make an Android application for trip planning. This Android application can be seen as two different modules: the trip planner and the subscriber. The trip planner module is based on the trip planner OTP, which was introduced in Section 2.4. The subscriber uses information received by the trip planner to subscribe to information regarding generated trips, routes and vehicles.

The two modules and their implementation will be described; first the trip planner module in Section 3.1, then the subscriber module in Section 3.2. The trip planner module contains a set of important Java classes that will be discussed in Section 3.3. Some of these classes are from the OTP application, the others have been implemented during the project to suit the new trip planner.

The subscriber module also has a set of classes, described in Section 3.4. Since OTP did not have a subscriber, none of these classes can be found in the original OTP application. The classes are three different subscribers: one for vehicle positions, one for trip updates and one for alerts. They were implemented during the project.

3.1 Trip Planner Module

The trip planner developed in the project is based on Open Trip Planner for Android. The following changes and added functionality were implemented in the application:

Parser A new function for parsing the server response was implemented, so that the new trip planner can communicate with the Itract servers rather than the servers used by OTP. The server request was also modified to the appropriate format.

Bus stops The web-based Itract application is able to display bus stops. This functionality was added to the Android application also.

Vehicle positions The Itract trip planner for Android is also able to show the positions of vehicles, such as buses.

Subscriber The subscriber is a module that was added to the application to get real-time data, such as updated vehicle positions. The subscriber will be discussed in the next section.

Most of the graphical user interface is generated by OTP code, and the classes to store trip information are classes used by the original OTP application. The classes new to the Itract trip planner will be described in detail in Section 3.3. They are the following: ItractGraph, ItractTrip, ItractBusStops, ItractDepartures, and ItractBusPos.

3.2 Subscriber Module

The subscriber application can connect to Itract through a broker. The purpose of the subscriber application is to fetch information from the network regarding public transportation,
such as trip updates and vehicle positions. Information regarding departures and arrivals is not based on fixed time schedules, but is updated to take time changes such as delay into consideration. Therefore, the subscriber is updated with the appropriate information when connected to the network. According to the publish-subscribe model, a publisher transmits information about, e.g., the train and bus traffic to the broker, whereafter this information is available to the subscribers that subscribe to it, if these subscribers are connected. If a subscriber is not connected, the information should be available once the subscriber connects, if the information is still relevant, i.e., if the publication has not expired. For example, a publication on how far a train has travelled between two locations is not relevant once the train reaches the destination. The time to live value of a publication is set by the publisher when the information is published.

The subscriber application is able to fetch information about public transportation using the information from a planned trip. The information provided by the trip planner module is used by the subscriber to create subscriptions that are then transmitted to the broker. For example, if the user wants to travel from one location to another, the trip planner will display to him the generated trip with times of departure and arrival for each vehicle. In this example, to travel from A to C, one will take Bus 1 to an intermediary location B, then take Bus 2 to reach destination C. The fixed schedule for the buses looks like the following:

| Bus 1: A -> B | Departure: 11:00 | Arrival: 11:20 |
| Bus 2: B -> C | Departure: 11:25 | Arrival: 12:00 |

As the subscriber subscribes to Bus 1 and Bus 2, it will receive real-time data concerning the two buses. The subscriber receives a delay value for each bus. If there is no delay, the value is 0. In this case, the subscriber receives a delay value for Bus 1 with value 600 seconds; thus the arrival time of Bus 1 will be 11:30 rather than 11:20. This is later than the departure of Bus 2, which departs at 11:25, which means that one cannot make the connection. The application user is in this case notified by the subscriber that he will miss Bus 2, and is then offered to plan a new trip from location B.

It could occur that the departure of Bus 2 is delayed. Therefore, delays of both arrivals and departures are taken into consideration when the subscriber module handles the real-time information. Any notification of delay received by the subscriber will update the schedule in the application, so that the user can see the new arrival and departure times.

### 3.3 Trip Planner Implementation

Since the original Open Trip Planner application is not compatible with the Itract servers, a modified OTP application had to be written. Below will be explained the changes done to the original OTP application in order for it to work with Itract. The additional functionality for the Itract trip planner will also be explained.
Two parts that had to be modified with regard to compatibility with Itract, were the trip request and the metadata request. These tasks are carried out in the TripRequest and MetadataRequest classes of the OTP application. The calls to the server had to be modified, since the Itract server has a different API from the regular OTP servers.

The request to the OTP server to get the graph bounds may look like this:

http://rtp.trimet.org/opentripplanner-api-webapp/ws/metadata

This metadata contains the bounds as well as what transit modes are available. The request to the Itract server is different:

http://itract.cs.kau.se:8081/proxy/api/secure/proxyInfo?bounds=true

Thus to get the graph bounds from the Itract server, the request had to be reprogrammed. A request and a parser function for the response were implemented in a separate class ItractGraph during the project. The parser function is called in the MetadataRequest class of OTP, where the original request for a server’s metadata used to be.

Once the graph bounds are available to the application, it can compute whether the user is placing start and end markers within the bounds or not. As long as the user is placing the trip markers within the bounds, it is possible for him to get a trip plan based on the marked start and end locations. In order for the application to fetch a trip plan from Itract, changes had to be done to the OTP code. A number of parameters has to be passed to the Itract server, which differs from the set of parameters passed to an OTP server. Once the trip plan response is available to the application, it is parsed as in the case of the graph data. Since the trip plan format is different from that of regular OTP servers, the parsing also had to be modified, just as in the case of the metadata request.

The OTP parser contains a lot of complex code, so rather than modifying this code, it was more efficient to write a new class for parsing the trip plan as well. The class ItractTrip was added to the application. This class, used from OTP’s TripRequest, uses the Java classes JSONObject and JSONArray, found in the org.json library, to parse the trip plan and sets the attributes of necessary objects to the correct values given by the plan. When the TripPlan object is made available to the rest of the OTP application, the routes can be drawn on the map by original OTP code.

The major changes and additions concerning parsing were done in the ItractTrip class. This is where the response from the Itract server is parsed and converted into a trip plan that the application can display to the user in an intelligible format.

The two additional classes, ItractGraph and ItractTrip, will be explained in more detail in the following subsections. Both classes were added to the OTP code during the project and cannot be found in the original OTP application.

Besides modifying the OTP application to work with the Itract server, there was additional functionality that was added to make the Android application more similar to the web-based Itract application. The web-based Itract application has two services: display bus stops and display vehicle positions. The bus stops are displayed on the map as icons, using Google
Maps’ Marker objects. When the user touches the label of an icon, he will be provided a list of departures from that bus stop. The vehicle positions are real-time information of where buses and trains are located. In the project, these two services were implemented in the Android program.

To implement these two services, three buttons were added to the left-hand menu. These were labelled ‘Show stops’, ‘Show stops in area’ and ‘Show vehicle positions’, respectively. The buttons are touchable text views with checkboxes defined in an xml-file. In MainFragment, they are added to a so-called list view, which has an item listener. The listener will identify the button the user presses, and run the appropriate instructions.

If a button is checked, i.e. the bus stops or vehicle positions are displayed, the button will be unchecked upon press and the markers for stops or positions will be removed from the map. If the button is unchecked, it will be checked upon press, and any bus stops or positions on the map will be removed, in order to prevent cases with duplicate stops, whereafter the new markers will be placed.

It is also possible to show the bus stops within a given area based on the user’s location. The ‘Show stops in area’ button was added for this purpose.

Whenever any of the ‘show stops’ buttons are checked, the other one will be unchecked. The ‘Show vehicle positions’ can be checked simultaneously as any of the other two, i.e. one can display the stops and the positions at the same time. Since the vehicle positions are updated constantly by the server, new data must be requested at certain points, e.g., when the user moves the camera, in order to display markers that match the real positions as accurately as possible. Another solution is using the subscriber, discussed in more detail in Section 3.4, to retrieve real-time data.

The co-ordinates of the bus stops and positions are available on the Itract server. To make calls to the server, the classes ItractBusStops and ItractBusPos were implemented. Since requests to servers should not block user input to the application, these classes were set as subclasses to AsyncTask, just as the OTP classes MetadataRequest and TripRequest. An asynchronous task is run in a separate thread with the command execute(). This will call the doInBackground function. A short explanation of AsyncTask follows in Section 3.3.1.

ItractDepartures is a class that was added to generate a departure list for each bus stop. The ItractBusStops, ItractDepartures and ItractBusPos classes will be explained in Section 3.3.4, 3.3.5 and 3.3.6, respectively. They are similar to the ItractGraph and ItractTrip classes discussed previously, as they parse a JSON response from the server; but the response in these cases will yield bus stop information and co-ordinates for positions.

3.3.1 AsyncTask

AsyncTask is a Java class found in the Android OS library android.os. It is used to create threads, which can be used in an Android application to process e.g. server requests and responses in the background. In a simple case, the asynchronous task process will go through three different procedures:

**doInBackground** The task itself. Any results of the computation is returned from doInBackground().
**onPostExecute** Is called when doInBackground() has finished and the task has not been cancelled.

**callback** The callback function is defined in the class listening to the task and can be called from onPostExecute().

To implement an asynchronous task, one needs a listener class which contains the callback function, mentioned above. The task itself is run as a separate thread, defined in a separate class. When creating a subclass of AsyncTask one has to specify the type of input and output of the task. In most cases of the OTP application, the input is a String, containing the URL request to the server. The output can be, for example, in the case of MetadataRequest, an ItractGraph. The output is what will finally be sent to the callback.

A task can be started from the listener class with

```java
new ItractBusStops(this).execute(url);
```

The input to the task is a string named url. In order to call the callback function later, the listener object needs to be sent to the task’s constructor, i.e. to the method which shares the name of the class and is called upon creation.

When the task has finished successfully, it will call its onPostExecute() method. The value returned from doInBackground() will be sent as a parameter to onPostExecute(). It is from here that the call to the callback can be made, forwarding the return value, e.g.

```java
listener.onItractBusStopsComplete(stops);
```

In the case of the OTP application, the listener class implements an interface for each AsyncTask. The interfaces are written in separate files and each interface contains a declaration of the callback function associated with the appropriate AsyncTask. The interfaces enable the listener class to type cast. So, if the AsyncTask expects an object of type MetadataRequestCompleteListener, any class can be a listener by implementing the MetadataRequestCompleteListener interface.

### 3.3.2 ItractGraph

In ItractGraph there is an ArrayList of ItractGraph objects, which represent the graph information. The class ArrayList can be found in the java.util library. After making a request to the Itract server, a JSON message is returned as response. The response contains the graph identifier and the bounds of a graph. The bounds are specified by latitude-longitude points. As mentioned earlier, parser() is called from OTP’s MetadataRequest, which then returns the main ItractGraph object used to retrieve the graphs. The parser() function processes the JSON document, using the JSONObject and JSONArray classes.

Since MetadataRequest is an asynchronous task, there is a callback function in the MainFragment class which awaits the completion of the task. Once the ItractGraph object is returned and the callback executed, the graph bounds are converted from String to double values. The minimum co-ordinates and maximum co-ordinates will be found for each graph, so
that the bounds will form a rectangle, using the upper right and lower left latitude-longitude points as corners. The bound values did originally define a polygon, when retrieved from the Itract server, but is converted to a rectangle to suit the OTP application better. This rectangle can be displayed with lines on the map, by using Google Maps’ addPolyline() function.

3.3.3 ItractTrip

The purpose of the ItractTrip class is to return a TripPlan object. TripPlan is an OTP class, which uses a set of related classes to define a trip. These classes exist in the original OTP application and are the following

Itinerary A possible trip, from start to destination.

Leg A sub-trip of the itinerary, which can be taken by a travel mode, e.g. bus or walk.

WalkStep A sub-trip of a leg. Steps are points used in drawing the line on the map.

Place A location, with a name or address and co-ordinates.

In Figure 4, the four classes and their relations can be seen. Some important attributes for each class are listed. The TripPlan class contains both a couple of Place objects and a list of Itinerary objects. Itinerary contains a list of Leg objects. The Leg class contains two Place objects and a list of WalkStep objects.

There are two significant Place objects used in ItractTrip; these are the start and the end of the trip, from and to. The two Place objects can be seen in TripPlan in Figure 4. They are used in the creation of the TripPlan object.

In order to draw a route connecting intermediate locations, the values of walk steps are fetched from the Itract server. The steps are intermediate locations that are used to draw the lines that form the route. Without them, the route would be a straight line from start to destination of the legs. The step values are provided for walk routes. For buses, the location of the intermediate bus stops are used to draw a route. Thus the bus route is based on the bus stops rather than on the road the bus travels.

The OTP application used an additional object called legGeometry to keep track of steps when they were added as points on the map. The modified version does not use this object; instead, the step values are retrieved directly from the WalkStep objects through a loop and added as points to be used in drawing the route.
To understand the data parsed by the ItractTrip class, one can take a look at the response from the Itract server. The server returns the data fetched from the graphs in an array where each element represents a graph. Using the Itract URL for trip plans, one can retrieve this example of a graph, seen below. The graph contains an identification name and an array of data.

```json
{
    "graphId": "Test_otp",
    "data": [3]
}
```

Each element in the data array contains information about one itinerary. Each itinerary contains a number of legs. The legs each have a transit mode. Below is a JSON example of a leg with mode ‘walk’.

```json
{
    "mode": "walk",
    "walkSteps": [
        {
            "(from): "place1",
            "(to): "place2",
            "distance": 1000,
            "duration": 120
        },
        {
            "(from): "place2",
            "(to): "place3",
            "distance": 2000,
            "duration": 240
        }
    ]
}
```
The leg for bus trips looks similar, but contains some different data that will be discussed later.

The parse function of ItractTrip contains four loops, which are (a) for the graphs, (b) for the itineraries, (c) for the legs and (d) for the walk steps, respectively. Since the server returns data fetched from separate graphs, the array of graphs is parsed in the outermost loop. The function creates a JSONArray object to extract the JSON arrays and a JSONObject to extract the objects. These two classes can be found in the org.json library. In the itinerary loop, an Itinerary object is created. A Leg object is created in the loop for legs. When the parser function iterates through the legs, it fetches the information for each leg, such as start and end time, distance travelled during the leg, as well as what transit mode the leg represents.

To store information about the start and end locations of a leg, Leg contains two Place objects. The information is retrieved from the to and from objects of the JSON-formatted response. A place has a name, latitude-longitude co-ordinates and a time of departure or arrival, as shown below.

"from":
{
    "name":"Vikengatan",
    "lat":59.37729090764417,
    "lon":13.490118271002833,
    "departureTime":1382599091000
}

The parser function extracts the co-ordinates, using JSONObject.getDouble(), and uses the co-ordinates to create a Place object. The name, ‘Vikengatan’ in this example, is also used in the initialisation of the Place object. The departure time is then added to the leg itself, stored in the string startTime. A similar procedure is carried out for the destination to in the response. A field arrivalTime is extracted in this case, then added to the leg’s endTime. The two Place objects, start and destination, are assigned to leg.from and leg.to, the Leg object’s own internal Place objects.
To summarise the result of the process, we have the following assignments from the JSON response to the Leg attributes:

```plaintext
leg.from.name = from.name
leg.from.lat = from.lat
leg.from.lon = from.lon
leg.startTime = from.departureTime
leg.to.name = to.name
leg.to.lat = to.lat
leg.to.lon = to.lon
leg.endTime = to.arrivalTime
```

Both mode and distance will be fetched from the JSON string and assigned to the leg’s data objects, `leg.mode` and `leg.distance` respectively. As seen in the earlier JSON example of a leg, there is an array called `steps`. This array contains the intermediate points the leg will pass. The points are given as latitude-longitude co-ordinates. Using these points to draw a line on the map, the application will provide the user with a path he can follow to reach the destination. Steps are only available for mode WALK, where it is important for the user to get an as accurate path to follow as possible. The more step points that are provided, the more accurate a walk route will be delivered. The step points are stored as `WalkStep` objects in the list `walkStep` found in the `Leg` class.

The JSON object for bus trips looks a bit different from the walk leg. For a leg with mode BUS, there are some additional values to be read. Amongst these values are `agencyName`, `tripId`, `tripHeadsign` and `routeShortName`. They are stored in `leg.agencyName`, `leg.tripId`, `leg.headsign` and `leg.routeShortName` respectively.

Since steps are not available for bus trips, the array `intermediates` can serve the same purpose. This array contains the location of intermediary bus stops as latitude-longitude co-ordinates. The route on the map will thus be drawn as lines between these intermediary bus stops.

The parser processes data as explained above, creating new `Leg` objects for each leg in the JSON response. For each element in the JSON array `data`, the parser function creates a new itinerary. Each itinerary is added to the trip plan’s itinerary list, so that all the trip data can be accessed through the `TripPlan` object.

The callback function in `MainFragment` is given the list of `Itinerary` objects from the trip plan in `TripRequest`. The itineraries are then used by the trip planner application to display a trip.

In Figure 5 is an example of a trip displayed on Google Maps. The graphical user interface is that of Open Trip Planner. Trip information, such as place names and the points between which the line was drawn, was requested from Itract and parsed in `ItractTrip`. Using the original OTP application, a similar trip can be displayed, when using trip data from an OTP server.

Since the bus routes are drawn between the bus stops, there is a long straight line in the picture between two stops. This line does not correspond to the actual route the bus is driving, only the shortest way between the two stops.
3.3.4 ItractBusStops

A bus stop has the following attributes: latitude-longitude co-ordinates, a name, a bus identifier, a stop identifier and an identifier for the associated map marker. The bus stop objects are stored in an ArrayList.

In doInBackground() of the class ItractBusStops, the application makes a request to the Itract server for data on bus stops. Once the response is available, it is parsed as in the case of ItractGraph and ItractTrip. Each graph in the JSON array will be iterated, as well as each object in the JSON array ‘data’. The objects in the data array are the bus stops. These contain the aforementioned attributes, except for the marker ID.

The ItractBusStops task will be triggered when the application user presses a button in the menu. When the button is pressed, showBusStops(), defined in MainFragment, will be called to start the task. Any old markers for bus stops will be removed before new ones are created. The user has two options when he wants to see the bus stops: there is a button for showing bus stops within the area displayed by the camera and another button for displaying bus stops around the user’s own location on the map. The callback onItractBusStopsComplete() will be creating the markers by using information from the ItractBusStops object once the task is finished.

In order to update the bus stop information when the user moves the camera outside the previous camera bounds, a new call is made to showBusStops() in onCameraChange(),
which is triggered at camera movement. If the ‘Show stops’ button is ‘on’, whenever the camera is moved, the program will check how far the camera has been moved, and update the bus stop information if necessary, by starting another asynchronous task for that purpose.

Figure 6 shows the menu with three check boxes that have been added during the project. When ‘Show stops’ is checked, the bus stops will be displayed on the map, as is seen in the figure.

Figure 6: The ‘Show stops’ button is checked and the bus stops displayed.

In Figure 7 the menu is closed. The bus stops are displayed in a given area based on the camera view. Moving the camera from this view will update the bus stops.
3.3.5 ItractDepartures

When the application user touches the title of a bus stop marker, the list of departures from that bus stop will be displayed. The ItractDeparture class was made for that purpose. This is another AsyncTask, which is executed when the user touches the title of a bus stop marker.

A departure has the attributes routeShortName, tripHeadsign, time and agencyName, to match the information from the JSON response. Departures are stored in an ArrayList.

As in the previous parser functions, the graphs and the objects, in this case the departures, are processed in nested loops. The values from the JSON document are assigned to the departure attributes. Because the routeShortName attribute is sometimes defined as tripShortName in the JSON response, if routeShortName is not found, the value of tripShortName will be read instead. Both contain the bus line number.

The server is requested for departures within the next 24 hours, but a maximum of 10 departures are parsed. These are the 10 departures closest in time.

In MainFragment, there has been written a callback onItractDeparturesComplete(), which fetches the departure information from the returned ItractDepartures object. Information will be displayed in a dialogue. Bus route, headsign, date and time will be displayed for each departure. For example:

Bus 12 Karlstad Stora Torget
01 Apr 2014 15:47
3.3.6 ItractBusPos

The ItractBusPos class contains information about latitude-longitude co-ordinates, agency ID, route ID and vehicle ID. The former will determine the position of a bus, while the other attributes will be displayed in the title of the bus icon. In doInBackground() of ItractBusPos, a request is made to the Itract server to get the position of vehicles. The request URL contains the boundaries of a given area within which the server should return vehicle positions. The response is parsed as in the previous cases, in order to create BusPosition objects.

ItractBusPos will be executed as a response to that the user presses the ‘Show vehicle positions’ button. As in the case of bus stops, the camera position will be used as boundary parameters for the area where icons are to be shown. The number of bus positions available from the server is too large to be handled efficiently, so the application will only request the bus positions within the area viewed on the screen.

The callback for ItractBusPos is onItractBusPosComplete() which will display the markers for bus positions. As in the case of showing bus stops, the bus positions will be updated when the camera moves a certain distance, by using the onCameraChanged() to detect camera movement. The updating of bus positions will also be accomplished with the subscriber module, which will be discussed later.

As explained, the application user will be provided a dialogue with departures whenever he touches the title of a bus stop icon. Already implemented in the original OTP application, there is an information dialogue displayed whenever the user touches the title of a trip mode icon; if he plans a trip and presses the title of a ‘walk’ icon, a dialogue with e.g. duration time and distance will be displayed for that walk. Google Maps markers can only have one event handler, so the functionality of the different type of markers were implemented in the same handler. The functionality for departure information was added to the existing event handler for trip information. The touched marker will be identified by the text in its title; if it contains the strings “to” or “from”, the marker is a mode marker, else, if it does not contain “Agency ID”, which is the title of the position markers, the marker is a bus stop. The position markers have no functionality in the event handler.

The markers for bus stops and positions use icons downloaded from http://mapicons.nicolasmollet.com. They are loaded into the application from the assets folder. OTP also has a set of icons in the resource folder, where the application icon is loaded from. The icons for bus stops and positions were chosen so that they differ in colour, to easier separate them on the map.

3.4 Subscriber Implementation

The subscriber module sends subscriptions to the intermediary system which acts as a broker. The subscriber uses data from the trip planner module to create the subscriptions. Such data may be vehicle identifiers or trip identifiers.

When the subscriber gets a notification, it will send the data in the notification to the trip planner module, which uses the data to provide feedback to the end user. The subscriber
is run in a separate thread and provides a callback mechanism. The Java classes Thread, Handler and Message were used for this purpose, where Handler and Message are classes used for the Android system. Since no threads but the main thread can interact with the graphical user interface, the handler will be used as a callback to be run in the main thread.

The real-time data the subscriber module is to handle are positions, trip updates and alerts. Three different subscribers were thus implemented, to subscribe to each category of real-time data. A publisher was implemented in the project to work with the intermediary system, thus providing these real-time data. Below is a short description of GTFS and the publisher, followed by a description of how each subscriber were implemented in the Itract planner application.

3.4.1 GTFS and Publisher

The General Transit Feed Specification [4], GTFS, provides updates concerning public transportation. The publisher fetches GTFS feeds to provide real-time data concerning trip updates and alerts. The GTFS feeds are parsed in the publisher, which uses the data to create publications. The publications are then interpreted by the subscriber subscribing to them, whereafter the subscriber will send the information to the main thread of the application. The data will thus be transmitted and transformed through these steps:

GTFS -> publisher -> broker -> subscriber -> trip planner

A GTFS document contains a header, followed by entities. An entity may be

```plaintext
entity
{
    id: "trip_update0"
    trip_update
    {
        trip
        {
            trip_id: "54452738"
        }
        stop_time_update
        {
            stop_sequence: 1
            arrival
            {
                delay: 60
            }
        }
        stop_time_update
        {
            stop_sequence: 2
            arrival
            {
                ...
Once the values in the above example have been parsed and extracted by the publisher, a Java object can be created using a `TripUpdate` class:

```java
class TripUpdate {
    int arrivalDelay;
    long arrivalTime;
    int departureDelay;
    long departureTime;
    int stopSeq;
}
```

Using the trip ID, also available in the feed, the publisher can publish the data on a channel identified by the trip ID.

A similar procedure is carried out for the alerts. These are fetched as GTFS, parsed in the publisher and then published to the broker or intermediary system.

### 3.4.2 Positions

In `onItractBusPosComplete()`, rather than just creating the position markers directly, a `PosSubscriber` object is created and a new thread started using the `PosSubscriber` object:

```java
t = new Thread(mainSub); t.start();
```

Sent to the `PosSubscriber` object’s constructor is an array of vehicle identifiers and the `MainFragment` object as a listener. The vehicle identifiers have been fetched during the `ItractBusPos` task, with a request to the Itract server. These identifiers will be used to specify what channels the subscriber will subscribe to. `PosSubscriber` contains an `ArrayList` of `BusPosition` objects where bus position information will be stored when the subscriber has been notified with this information.

In `PosSubscriber` there is a function `notify()`, called every time the subscriber is notified with a message. The message contains the name of the channel as well as the payload of the message. The channel may be such as a broadcast channel or the channel for a certain vehicle and its attributes. The broadcast channel is what all positions subscribers listen to, and is called “broadcastPos”. When the position subscriber receives a message on the channel corresponding to a certain vehicle, the subscriber creates a `BusPosition` object to store the incoming data. What should be received on the channel is a JSON-formatted document with the attributes of a vehicle. The document will be parsed with `JSONObject`, a JSON parser for Java mentioned earlier in Section 3. Below is an example of the attributes of `BusPosition` being assigned the values from channel `<vehicleID>`:
position.lat = <vehicleID>.lat
position.lon = <vehicleID>.lon
position.agencyID = <vehicleID>.agencyId
position.routeID = <vehicleID>.routeId
position.vehicleID = <vehicleID>.vehicleId

The BusPosition object is then added to an ArrayList, which will be sent to the handler in the main thread when all vehicle positions have been published. The publisher will send on “broadcastPos” the message “Complete“, to tell the subscriber that all position data currently available have been published.

3.4.3 Trip Updates

When a trip is generated in OTP, the application user can press the icon representing the mode of one of the routes to get a table of arrivals and departures for the entire trip. This table can be seen in Figure 8. A subscriber button was implemented in Itract’s trip planner to enable the user to subscribe to the updates of the trip. When the button is pressed, subscribe2Trip() will be called. The function is defined in MainFragment which is the listener to the subscriber threads.

![Figure 8: Trip information. The two subscribe buttons were added during the project.](image_url)

The subscriber thread for trip updates is given the itineraries generated by the trip along with the trip identifiers of each leg. The trip identifiers are used to define the channels the
subscriber will subscribe to, just as the vehicle identifiers define the channels for the position subscriber.

When notified, the trip subscriber will iterate through the legs of the trip to access the departure and arrival times of each leg. As mentioned earlier, a leg is a subtrip of an itinerary and represents a mode, such as WALK and BUS. E.g. two following legs with mode BUS indicates that the traveller will switch bus on his trip.

The message received from the publisher is information parsed from a GTFS feed. It may contain a delay value of a new time of a departure or arrival. In the case of a delay value, the value is stored in the appropriate Leg object, i.e. the leg matching the trip ID in the channel name. OTP’s Leg class did not originally contain a delay variable, so this was added during the project. Two delay variables were necessary: one for arrival and one for departure.

In the case the message received contains a new time for the arrival or departure, the current arrival or departure time is simply replaced by the new value, for the appropriate leg.

The message payload is a JSON document as in the case of the position subscriber. When the subscriber is notified with a “Complete” message on the channel “broadcastTrip”, it will check for each transit vehicle in the trip if the arrival of the vehicle is later then the departure of the next vehicle. If such a case occurs, it means that the connection between the buses or trains cannot be made, as the second vehicle will depart before the first one arrives. In the case the check returns true, the subscriber will notify the main thread by sending to the handler a message containing the Leg object whose arrival time exceeded the next leg’s departure time. The main thread will forward the notification to the end user in the form of a dialogue message on the screen, also asking the user if he wishes to plan a new trip from the end location defined by the Leg object, i.e. from where he missed the connection. The new trip is prepared in updateTrip(), which is similar to the original requestTrip() that is called when a trip is to be generated; updateTrip() also creates a request and executes the TripRequest task.

Even in those cases where the trip updates received by the subscriber do not affect the transit connections, the application user should be able to see the new times of departure and arrival. The table of arrivals and departures will be updated whenever the subscriber receives a message containing updates. The delay value stored in the Leg object is added to the leg’s current arrival or departure time to display the real-time information in the table.

In order to update the table, the main thread makes a call to update() in DirectionListFragment, the class which creates the table. During the project, update() was implemented for the purpose of updating the table while it is displayed on the screen. The object which handles the content of the table is the OTP object DirectionExpandableListAdapter. After the array of data stored in this object has been updated with the new itinerary information, the call

adapter.notifyDataSetChanged();

will notify the adapter that the view must update to display the new data.
3.4.4 Alerts

Next to the trip-subscriber button in the table, a button for the alert subscriber was implemented. The alert subscriber will be started in a new thread, given the list of itineraries.

Alerts are received by the publisher as a GTFS feed. The alerts may contain the identifiers of routes, trips, vehicles and stops. This results in different sets of channels for the publisher. The alert subscriber subscribes based on the routes of the trip and the ID of the first stop.

Similar to the position subscriber, the alert subscriber will create a new object upon receiving a message on one of the subscribed channels, storing this object in an ArrayList. The object is an Alert, containing the attributes cause, effect, text and ID, similar to the alert entities in the GTFS feed. The values of these attributes are extracted from the JSON document published by the publisher. Once a “Complete” message arrives on the channel “broadcastAlert”, the alert subscriber is ready to notify the main thread, sending to the handler the ArrayList of alerts.

In onAlertNotify(), called by the handler, the alerts are checked for duplicates. Duplicates may occur, e.g. when the subscriber has received two identical alerts on one of the route channels and the stop channel. The ID of the alerts are used to detect duplicates. Once duplicates are removed, the alerts are displayed in a dialogue window for the user to see.

3.5 Summary

The implementation of the trip planner and its subscriber module has been discussed in this section. A parser for the Itract responses, markers for stops and vehicle positions, as well as a subscriber has been added and integrated with the OTP source code. The classes added to the trip planner module were

ItractGraph Used to get the bounds of the graph, in order to, e.g., visually display the graph bounds on the map.

ItractTrip Used to get a trip plan from Itract, based on the user’s input to the application.

ItractBusStops Gets the position of bus stops from Itract, so that the stops can be displayed on the map.

ItractDepartures Gets the departures for a certain stop. The user is able to see departures when pressing the title of a stop icon.

ItractBusPos Gets the position of vehicles, e.g. buses, so that they can be displayed on the map. The IDs of vehicles are used to create subscriptions for the subscriber.

The first two were implemented in order to use Itract’s servers for trip planning, while the other three were used to add new functionality to the trip planner. Some useful OTP classes were also discussed, in relation to the new parsing. These were TripPlan, Itinerary, Leg, WalkStep and Place.

To the subscriber module were added three classes: one for positions, one for trip updates and one for alerts. Each class in the subscriber module is a subscriber running in a separate
thread. They communicate with the main thread via a handler, sending received data to a callback function. The publisher of real-time data is parsing GTFS messages. The parsed results are published on the channels, thus made available to the subscriber. A subscriber will be started when the application user presses a button, e.g. ‘Show vehicle positions’.

4 Intermediary System

Since the publish-subscribe system is to work for three different types of applications, i.e. web-based, Android and iOS, there has to be an intermediary system that can communicate with the Itract publisher and these three types of applications. The old Itract broker is only compatible with Java, so it works only for a Java subscriber. The new intermediary system must then also be able to understand Javascript, to work with the existing web-based application, and any language used on the iOS system. This intermediary system must be able to handle at least a thousand subscribers that are simultaneously connected.

The intermediary system could be a distributed database such as Cassandra or a push-based server using Web Sockets. In this project, distributed databases were investigated and compared concerning performance and functionality. The so-called NoSQL databases provide good performance in a distributed environment. A background on NoSQL databases will be given in Section 4.1.

There were different solutions of how to integrate the intermediary system with Itract. One solution was to have a subscriber subscribing to all information in the broker and storing it in the database, whereafter the database would push data to the clients. Another solution was to let the database forward subscriptions to the broker and then forward the notifications to the subscriber. Thus, the intermediary system would work as a translator between broker and subscriber.

As will be mentioned in Section 4.2, there exists a database system which supports publish-subscribe functionality and which can actually take the role of a broker. This database is known as Redis. Another useful database in the project was MongoDB, which will be described in Section 4.3. A database that was examined, but not used, in the project was Cassandra, which will be given a short description in Section 4.4. In Section 4.5, the implementation of these databases will be discussed as well as why they were chosen.

4.1 Distributed Databases - NoSQL

A database can have up to two of the following characteristics: consistency, availability and partition tolerance. These characteristics are known as CAP [9] and are summarised below

**Consistency** When reading the database, the clients will see the latest updates.

**Availability** Clients can always read or write to the database and the operations return the intended results.

**Partition tolerance** Data can be partitioned in separate nodes, so that the database can be accessed even though parts of it may be down.
For a distributed database, partition tolerance is the main characteristic. Relational databases often have high consistency due to normalisation. However, when dealing with real-time data, it is important that the database is always available and have short response times. Since relational databases, using normalised data and join operations, may be slow in distributed database systems using large datasets, there exists an alternative known as NoSQL databases [13]. NoSQL databases store redundant data, thus providing better availability than relational databases, but less consistency.

Since relational databases enable complex structures and queries of data, they may not be optimised for simpler data that is to be stored in a distributed system [13]. The tables of relational databases contain static attributes, resulting in that objects with different kinds of attributes have to be stored in different tables, to avoid null values. Relational databases are often centred round consistency, thus giving up availability in a distributed system, according to CAP.

Redundant data require more space in which to store the data. However, if latency is a greater issue than the amount of storage space, then redundancy can be a solution to increase availability. Data replicated across multiple servers distribute the load on these servers; rather than having a single server handle all the reads to the database, a cluster of servers can handle the reads together, decreasing delay for the clients. Redundancy also increases availability in the case of failures; if a server is down, the data can still be read from the other replicas.

The problems that may arise with replication, however, are those concerning consistency [9]. If a replica is written to, then the written data must be written to all other nodes sharing data with this replica. For a consistent database, the write operation is not complete before all nodes have been written to. For an eventually consistent database, the writes are done asynchronously to increase availability.

Problems may occur when multiple writes are done to the same value on different nodes. Relational databases usually lock a data set when it is being modified. Some NoSQL databases uses optimistic locking; when a transaction is conflicting, the transaction is rolled back [13]. This is done, for example, by the Redis system. In the case of Itract, where the publisher is the only writer to the database, conflicts should not be an issue.

There are different kinds of NoSQL databases. Three of them are listed below.

**Key value stores** Data is retrieved by a key [13]. They are not optimised for complex queries; a value is retrieved as from an array. Key-value stores usually store data in RAM. They are simple to use when looking up objects based on a single attribute [3].

**Document stores** Key value pairs are kept in JSON-like documents, where each document has a unique identifier [13]. Keys within a document must be unique. A document can be seen as an object, making this structure more object-oriented than that of the key-value store. Document stores are useful for looking up objects based on multiple attributes [3]. Compared to key-value stores, a document is like a key containing a number of key-value pairs.

**Column family stores** While somewhat similar to relational databases, column family stores avoid null values by defining a key-value pair only when it is needed for a row. Parti-
tioning is often done by partitioning rows according to key range and columns accord-

The three types of NoSQL databases above can partition data across servers by divid-
ing the key set according to a certain key range or using a hash function [13]. In the case
of key ranges, one or more routing servers direct the client to the correct database server
by providing a partition table, thus the entire cluster becomes dependent on these routing
servers.

The choice of NoSQL database is dependent on the use case. For simple queries, a key-
value store will work, while for more complex queries, such as range queries, a document
store is a better choice [13]. For SQL like queries, column family stores will work well.

A few NoSQL database systems were examined in this project. These were Redis, a key-value
store, MongoDB, a document store, and Cassandra, a column family store.

Redis, MongoDB and Cassandra all have a Java API, but Cassandra also has a query lan-
guage similar to SQL [13]. MongoDB uses range-based partitioning, based on the ID of its
JSON documents, but can also support hash-based partitioning [8]; the others use hashing
to distribute the data across multiple servers. To increase availability, these database systems
replicate data across nodes in the cluster. Clients can read, but not write, from these replicas
[13]. In order to achieve high availability, Redis is not fully consistent, meaning that data
may be inconsistent across the replicas for short time periods.

Below, the three databases, Redis, MongoDB and Cassandra will be introduced in the
following sections. Redis was used in the project for the publish-subscribe model, and Mon-
goDB was used for storing persistent data that can be queried later. Cassandra was not nec-
essary for the trip planner application; this database is used for social networks where many
writes occur, which is not the case with the trip planner. MongoDB store data in BSON, a
binary serialisation of JSON [1], and JSON has been used in the project before, e.g. in the
server responses.

4.2 Redis

Redis is a key-value store, storing data in memory. Data can be stored on disk to avoid data
loss lest the system is shut down [3]. As a key-value store, Redis supports simple operations,
such as delete, insert and look-up. Redis is known to be able to handle 100,000 gets and sets
per second if running on an 8-core server [3].

Redis can store data on disk in two different ways. One way is for Redis to take a snapshot
of the database each time after a given time interval. This time interval depends on the
number of keys that has been modified, e.g. every 60 seconds if 1000 keys has changed [16].
The data is saved to a file called dump.rdb. The other way to store persistent data is for Redis
to write to an appendable file whenever a key is modified [16].

Redis supports both range and hash partitioning [15]. An example of mapping a key to
a Redis instance using a hash function is crc32(<key>) modulo <number of instances>. The function crc32 takes the key name and returns a number.
Replication is accomplished with master and slave nodes [15]. For read intensive cases, the slaves can be used to handle read operations while the master handles write operations, where data from the master is replicated to the slaves. A slave node can also be used to write back-up data to disk, taking this load away from the master node.

To make sure that Redis masters and slaves are functioning properly, Redis uses a system called Sentinel [15]. Sentinel can monitor the Redis instances and notify errors, as well as managing failovers where the role of a failed master node has to be passed on to a working replica.

Figure 9 illustrates a Redis master node and four slave nodes. The master is written to, whereafter the master replicates the data to the slaves. Three slaves are used to receive read operations, while the other is used for writing data to disk as back-up. Using this architecture, the load of requests from the clients is distributed across several slave nodes. If the master node fails, a slave should be able to take its place.

Figure 9: Example of Redis architecture.
In Redis, data can be stored in five different structures [16]:

**Strings** To store a string in Redis, one will use the `set <key> <value>` command, e.g. `set message "hello"`. If the value is an integer, such as "10", it can be manipulated as an integer type, by commands such as `incrby <key> <increment>`, which increments the value.

**Hashes** Hashes are similar to strings, but that one can store fields and values, e.g. with the command `hmset <key> <field> <value> [<field> <value>]*`. Fields are almost like key-value pairs within the key.

**Sets** Sets are unordered values, stored by `sadd <key> <value [<value>]*`. Each value in a set is unique. Using the available Redis commands, one can find out if a value is a member of a set or if two or more sets share members by intersecting them.

**Sorted sets** When using sorted sets, each member of the set must be given an integer score. The command `zadd <key> <score> <value> [<score> <value>]*` adds values and their score to a set. Sorted sets can be used to rank the members of the set.

**Lists** Lists let one store multiple values for a key with the command `lpush <key> <value [<value>]*`. Values can be inserted and retrieved at given indices.

Keys added to the Redis database will be stored in memory until deleted by a client. To set a time-to-live value on a key, the Redis command

`EXPIRE <key> <seconds>`

can be used for this purpose [15]. Redis is also a publish-subscribe messaging system [15]. A subscriber can subscribe to one or more channels with the command

`SUBSCRIBE <channel>`

A publisher can publish a message with

`PUBLISH <channel> <payload>`

A message is an array of three elements [15], where the first element defines the message type. The message types are ‘subscribe’, ‘unsubscribe’ and ‘message’, where the latter is a notification to the subscriber.

Subscribers can also subscribe to all channels matching a given pattern. For example, given the channels `channel101`, `channel102` and `channel11`, the pattern `channel10*` would indicate a subscription to `channel101` and `channel102`. Keys in the database can also be retrieved by pattern matching. To get all keys starting with `karlstad.bus1.*`, one can use the following command

`KEYS <pattern>`
where the pattern in this case karlstad.bus1.*. We could get back keys such as karlstad.
bus1.position and karlstad.bus1.state, i.e. any attribute of Bus 1 that is stored in Re-
dis. Using the keys command is not a good option when dealing with large number of keys,
however; Redis will have to look through every single key in the database to find the keys
matching the pattern, making this a slow operation.

The pattern matching problem can be solved by using hashes however [16]. By creating
a hash karlstad.bus1 with, for instance, the fields position and state, one can retrieve all
the fields and their values with the command hgetall <key>, i.e. hgetall karlstad.bus1.
Redis will in this case only look through the data within the hash, rather than the entire
keyspace.

4.2.1 Redis Cluster

Redis Cluster helps maintain a cluster of Redis servers by automatically partition data be-
tween the server nodes [15]. It is easy to add and remove nodes from the cluster, and still
be able to map to the correct partition when requesting data. Keys are partitioned in hash
slots, where each node holds a range of slots. Partitioning is thus not based on the number
of instances, but on the number of hash slots, i.e. 16384.

The nodes in the cluster are all connected to communicate the state of the cluster, e.g.
if a node has failed [15]. The nodes can also forward publish-subscribe messages across the
entire cluster.

A key created in Redis, will be stored in one instance, then replicated to a number of
read-only slaves [14]. The cluster will work as long as each key has at least one working
node. If a master node fails, a slave node can take over its role as master.

To minimise the risk of cluster failure, redis-trib, the cluster manager program, distributes
the replicas over different physical servers if possible [14].

When a client sends requests to a node, the cluster will redirect the client to the node
holding the requested key [15]. The client will get a MOVED message, containing the hash
slot of the requested key as well as the IP and port of the node holding this slot. The client
should then be able to keep a map of slots and nodes, in order to send requests to the correct
node directly.

The nodes in the cluster do not proxy queries; therefore they use MOVE and ASK mes-
sages [14]. ASK messages are similar to MOVE messages, but they will only tell the client to
query the next node once, not for future queries.

To set up a Redis cluster, one starts the Redis servers are usual, then connects the nodes
with redis-trib.rb, which is included in the Redis Cluster distribution. The command to set
up the cluster, using redis-trib.rb, is ./redis-trib.rb create -replicas <#replicas>
<IP>:<port> [<IP>:<port>]+. <#replicas> is the number of slaves for each master that
should be in the cluster, <IP> is the IP address of the node and <port> is the port of the
node. An example from redis.io [15] shows a cluster set up on a local machine:

./redis-trib.rb create --replicas 1 127.0.0.1:7000 127.0.0.1:7001 \
127.0.0.1:7002 127.0.0.1:7003 127.0.0.1:7004 127.0.0.1:7005
The IP addresses and ports are not only important for the Redis nodes to communicate with each other, but also for redirecting the clients. A MOVE message will contain the IP and port specified for the node using redis-trib.rb as shown above. Thus, the IP address 127.0.0.1, i.e. localhost, should not be used if the client is on a remote machine.

4.3 MongoDB

MongoDB is a document store which supports operations more complex than key-value stores, e.g. request of data based on relational operators [3].

In a Mongo cluster there are groups of processes called replica sets [8]. A replica set contains a primary, which handles write operations, and a set of secondaries, which replicate the operations from the primary, using the primary’s operation log. Data can be read from any member of the replica set, but data is read from the primary by default. The members of a replica set communicate with so-called ‘heartbeats’ to detect if each member is responsive. If the primary member fails, a secondary can take its place through an election process.

MongoDB uses query routers as intermediate nodes between clients and the so-called shards which may contain the replica sets [8]. A query router uses the metadata stored on a configuration server to direct operations to appropriate shards. Figure 10 shows a Mongo cluster. There are two clients which communicate with the database via routing servers. Each replica set contains one primary and two secondaries in this example.

MongoDB uses both range partitioning and hash partitioning [8].
4.4 Cassandra

Cassandra is a column family store, supporting partitioning, replication, failure detection and recovery of failed nodes [3].

Data is stored in columns, where a column has a value and a timestamp [17]. There is also a structure known as super columns, which can contain other columns as its values. Columns are stored in column families, which are similar to tables [17]. Each row has a key, which points to the columns in the row. A row may have any number of columns.

Cassandra is a distributed database, using a cluster of nodes shaped as a ring. The clients can send requests to any node in the cluster, but will be redirected to the node that can return a response to the request [2].

When data is partitioned, Cassandra uses a hash function and the row key to allocate rows to the nodes [17]. The nodes each have a token, where the value of the token is determined such that, all the tokens in the ring divide the range of available hash values as evenly as possible. A node will have the hash slot within the range of its own token and the next node’s token.

Figure 10: A Mongo cluster with two shards each containing a replica set.
Replication can be done to any number of nodes. The replicas are chosen based on the replication strategy, e.g. the simple strategy, where replicas are chosen clockwise from the source node [2].

4.5 Solution

In order to set up a publish-subscribe system that works with both Java and Javascript, it was decided that the Redis database was to be used in combination with MongoDB. Cassandra is optimal for heavy reads and complex data storage, qualities which did not appear necessary to the project; therefore Cassandra was not chosen for the solution. The advantages of Redis is that it is fast and has built-in publish-subscribe functionality. It keeps data in memory, rather than stores them on disk, though it is possible to write data to disk by dumping the entire database to a file or append database changes to a file. If one wishes to use the persistent data for other cases than back-up, e.g. to query it in other applications, one can store it in a persistent database, such as MongoDB. This is not done by Redis itself, so one will have to implement this in the client. For example

```
jc.setex("karlstad.bus1.pos", 60, "59.5,21.0");
```

```
doc.append("_id", "karlstad.bus1").
    append("pos", "59.5,21.0").
    append("state", "running");
coll.save(doc);
```

This code belongs to a Java client. It sets a key which will expire after 60 seconds in Redis. Then, it sets a persistent document in MongoDB.

MongoDB will be used in the project to keep a history of the traffic. One can use the stored data for analysis and statistics, e.g. to examine how often a certain route has gotten delays. When the end users will be updated with real-time information, they will not need access to MongoDB, only to Redis. When the keys expire in Redis, these keys contain no relevant data for the users any more. The data will still exist in MongoDB. The trip plans made by users will also be stored in MongoDB. This would make it possible to, when the end users wish to load an old trip they have planned in the past, access MongoDB for their trip plan history.

Figure 11 is showing the publisher to the right and the client application to the left. The publisher program publishes and inserts data into Redis. Why it both publishes and inserts data will be explained later with an example. The publisher also inserts persistent data into MongoDB. The application can both request and subscribe to data in Redis. It can also retrieve data, such as its own trip history, from MongoDB.

Since a document in Mongo will be updated if a document sharing its ID is inserted into the database, the choice of ID is important. In the above example, if karlstad.bus1 is again inserted into MongoDB, it will replace the last insert. If we want to save the bus information for certain time intervals, a time can be added to the _id attribute, e.g. karlstad.bus1.2014-03-30.10:00 for an insert done March the 30, at 10:00.
The publisher program can not only publish data on a channel, but also store data in both Redis and MongoDB. Since a publication exists only while it is being published, and the publisher does not constantly publish data, the publisher must also store the data for a certain period of time in the Redis database, in order for subscriber to request the information when they connect. MongoDB is used for persistent data storage, such as history.

An example scenario will follow, with one publisher and one subscriber, of how the publish-subscribe model can be used by Redis. P is a publisher and S is a subscriber. The two modules shown at the centre of the following figures are part of Redis: the broker module at the top and the in-memory database at the bottom.

1. In Figure 12, publisher P inserts short-lived data in Redis, then publishes on the Redis channels.

2. In Figure 13, subscriber S connects and requests data from Redis that it has missed while disconnected. Only up-to-date data will be in Redis.

3. In Figure 14, subscriber S starts subscribing to channels in the Redis system.
Figure 12: 1. Publisher P inserts data into the Redis database and publish the same data via the Redis pub/sub system.

Figure 13: 2. Client/subscriber S connects to Redis and asks for all necessary data in the database.

Figure 14: 3. Subscriber S starts subscribing as long as it stays connected.
4.5.1 Redis Cluster Setup

To set up the Redis cluster, the necessary software was downloaded from a link on the Redis web page, redis.io. The file redis-trib.rb is included in the Redis Cluster distribution. This is a Ruby file. The Ruby library for Redis is downloaded with RubyGems:

gem install redis

This was necessary for Redis Cluster to work.

The program redis-server was used to start each Redis instance. Each redis-server was given its own configuration file, put in a separate folder. The configuration file was used when starting the Redis node with the following command

`../redis-server ./redis.conf`

where redis-server was located in the folder above the folders containing the configuration files.

The configuration file, redis.conf, contained the following information in the project.

```
bind 0.0.0.0
port 7000
cluster-enabled yes
cluster-config-file nodes.conf
cluster-node-timeout 5000
appendonly yes
```

The ports used in testing were ports 7000 to 7005. A minimum of six nodes are needed for a cluster, thus redis-server was run six times.

As mentioned in Section 4.2.1, the redis-trib program was used to set up the cluster. The IP and port for each node was specified. A replication factor of 1 resulted in three master nodes and one slave node for each master, a total of six nodes.

4.5.2 MongoDB Setup

MongoDB was very simple to install. After running `apt-get install mongodb`, a client program is ready to connect to the Mongo database. For a remote device to connect to MongoDB, the `bind_ip` variable in the Mongo configuration file mongodb.conf must be set to 0.0.0.0, as shown below.

```
bind_ip = 0.0.0.0
```

By default, it is usually set to localhost. The mongodb.conf file may be found in the etc folder on an Ubuntu system.

It may also be necessary to run the following commands on the server hosting the database.
iptables -A INPUT -s <ip-address> -p tcp --destination-port 27017 -m state \n--state NEW,ESTABLISHED -j ACCEPT

iptables -A OUTPUT -d <ip-address> -p tcp --source-port 27017 -m state \n--state ESTABLISHED -j ACCEPT

where <ip-address> is the IP of the client or application server. The commands let the application server connect to port 27017, which is the default port for MongoDB, and also lets MongoDB send traffic back to the application server through this port [8].

4.5.3 Client

To implement a Java client for Redis, one can use the Jedis library. For Redis Cluster, the JedisCluster class was used. With Jedis, one can issue Redis commands with Java code, by calling the functions contained in the Jedis library. Jedis is the Java driver for the Redis database system. Since Redis Cluster is in development, JedisCluster does not contain publish-subscribe functionality. To use publish-subscribe, one will have to use a Jedis object. An example is using Jedis pools and the Java Map class [7].

Pools are used to handle multiple Jedis instances, for example when working with multiple threads. A pool can be created for each node in the cluster. A Jedis instance can then be created by calling getResource() on a pool. This instance belongs to the Jedis class in the Jedis library, and can thus call publish() and subscribe(). To create a list of pools for the nodes

Map<String, JedisPool> nodeMap = jc.getClusterNodes();
List<JedisPool> nodePoolList = new ArrayList<JedisPool>(nodeMap.values());

where jc is the cluster object. A pool can then be picked at random to be used in the creation of a publisher instance and the same can be done for a subscriber instance. Since JedisPool is thread-safe, one can run a publisher and a subscriber in separate threads.

Other problems that may arise when using Jedis Cluster, is that an Android application may fail to connect to the database. Jedis uses a library called commons-pool and a class called BaseGenericObjectPool<T>. Code in this class is not compatible with Android, but not necessary for the application to work. Commenting out the code in jmxRegister() solved this problem during the project.

4.5.4 Publisher

The following discussion will explain how the real-time data is inserted into the database by the publisher. During the project, a new publisher was written to be compatible with the intermediary system.

Data is stored in MongoDB in BSON format, which can be converted to JSON documents. The most JSON-similar data structure in Redis is hash tables; these are used when storing information about vehicle positions, trip updates and alerts.
Using `Map<String, String>`, a hash table of string fields and string values are created in Java before being inserted into Redis. Each object, e.g. vehicle position, is fetched from an `ArrayList` and its attributes are put in the hash map. This hash map is then inserted into the database using the `JedisCluster` object, as the following:

```java
jc.hmset(positions.get(i).getVehicleId(), hash);
```

The first parameter is the name, or the identifier, of the key; the second parameter is the hash itself, i.e. the value of the key. This key is then given an expire value with the `expire()` command, e.g. 60 seconds.

An example of this hash table, containing position information, is shown below. Fields are placed to the left and values to the right.

```
vehicleId: "9031017002506254"
stopId: "625452"
tripId: "6254"
agencyId: "601"
currentLocationLat: "59.3791"
currentLocationLon: "13.56075"
```

The key containing this hash would be "9031017002506254", since the publisher stores the hashes based on the vehicle ID.

When the subscriber connects, in order to get the position data and display the vehicle positions to the user as soon as possible, the subscriber can request the data by using its list of vehicle IDs, identifying the key to the hash, and the wanted field. Each requested response is sent to the `notify()` function, just like a response to a subscription. The subscriber's last request is its own "Complete" message to `notify()`. Then, the subscriber is ready to start subscribing.

Since the client makes a request to the Itract server for the vehicle IDs used in the subscription, the application can also get the vehicle positions from the server, without querying the database before subscribing, to avoid delay until a notification arrives. The possibility of requesting data from Redis has still been implemented to solve the issue of keeping persistent publications, in the cases where this is needed.

In MongoDB, the position information is stored as JSON documents. Below is an example, using the same data as in the hash table.

```json
{
    _id: "9031017002506254",
    stopId: "625452",
    tripId: "6254",
    agencyId: "601",
    currentLocationLat: 59.3791,
    currentLocationLon: 13.56075
}
```
The data structure in Redis and MongoDB is very similar in this case. When the data is published by the publisher, a Jedis object is used, rather than a JedisCluster object. In the below example, a JSON document of a position is being published on channel <vehicleID>.

```
jedis.publish(positions.get(i).getVehicleId(),
  "{ " +
  ""lat"": " + positions.get(i).getCurrentLocationLat() + ", " +
  ""lon"": " + positions.get(i).getCurrentLocationLon() + ", " +
  ""agencyId"": " + positions.get(i).getAgencyId() + ", " +
  ""routeId"": " + positions.get(i).getRouteId() + ", " +
  ""vehicleId"": " + positions.get(i).getVehicleId() + ", " +
  ""timestamp"": " + new Date().getTime() + " +
  ”");
```

As mentioned earlier, MongoDB can be used to keep a history over trips. A user can request from MongoDB the trips he has taken with the trip planner. The trips will be displayed in a dialogue. While it has yet not been implemented in the trip planner, a future version could allow the user to pick one of his old trips and quickly have it displayed on the map.

The information the trip planner application is able to display of past trips is the start and end locations along with the departure and arrival times of these trips, no delays taken into consideration. Two classes were implemented to store and retrieve trips: TripSaver and TripLoader. Both classes are part of the Android application and connect to MongoDB which is on a remote server.

The data stored in MongoDB is at the moment very simple and contain the following:

```
_id: <ID>
from: <start location>
startTime: <departure time>
to: <end location>
endTime: <arrival time>
```

The ID is determined by the installation ID of the application. This ID can be created and written to a file when the application runs for the first time [5]. The file is removed once the application is uninstalled. In order to give each trip a unique ID, another value is added to the installation ID of each trip document. This value is incremented each time to avoid one document overwriting another.

When the user wants to see his trips, the trip documents starting with the installation ID are fetched from MongoDB. Example of a document ID is <installation ID>:<incremented value>. If <installation ID> matches the application’s ID, then the data of the document is retrieved and displayed to the user.

Both TripSaver and TripLoader are run in separate threads. A Handler is used to relay the data to the main thread. Each document retrieved from the database is put in a JSON array, for the handler to parse.
A trip will only be stored in MongoDB if the user has subscribed to it. When requested by the user, the trip will be displayed in the following format:

\(<\text{start location}> \rightarrow \text{<end location>}\)
\(<\text{departure time}> - \text{<arrival time>}\)

The intermediary system is now consisting of three servers. Two servers are used for the Redis cluster, since replicated data should not be stored on the same server. The third server is used for MongoDB. There is also a fourth server, where the publisher is running. The entire publish-subscribe system is thus consisting of the Android application, which is the subscriber, the three servers acting as broker and database, and the publisher running on its own server. To integrate the publish-subscribe system with Itract, the publisher should run on virtual machines in the Itract system. More machines can be added if necessary, since the Redis cluster and MongoDB are expandable.

4.6 Summary

This section has covered the solution to the intermediary system. Distributed databases were chosen to achieve high performance. According to the CAP theorem, a database can have up to two of the following attributes: consistency, availability and partition tolerance. A distributed database has partition tolerance, which leaves a choice of one of the other two. In this project, availability was chosen. While relational databases provide consistency, NoSQL databases provide availability. The NoSQL databases investigated were key-value stores, document stores and column families. One database system from each category was chosen and have been discussed in the thesis: Redis, MongoDB and Cassandra.

Redis solved the publish-subscribe problem, thus this system was included in the project. MongoDB was also chosen, to store persistent data. Cassandra was not necessary for this case, since the JSON format of MongoDB fulfills the demands of storing the trip data. With Redis, one can store data as strings, hashes, sets, sorted sets and lists. MongoDB stores data in BSON format, which can be converted to JSON documents.

How to set up the databases has been explained. The publisher used in the project was also explained, to give an example of how data can be stored in the databases. An overview of the publisher-subscriber relationship was also given.

5 Conclusions

A trip planner for the Itract system, supporting real-time updates, was developed in this project. The trip planner is based on another application, called Open Trip Planner, which uses a request-response server model to fetch trip information, such as trip plans. During the project, the source code of the Open Trip Planner application was modified so that it can communicate via the Itract API, receiving trip information from the Itract servers. The goal of the project was also to enable real-time data to be pushed to the application. The real-time data can be vehicle positions, trip updates and alerts. Vehicle positions are marked visually on the map of the trip planner, and updated whenever new data are received. Trip
updates will inform the user if there are, for example, delays in traffic. Alerts will provide other information, such as traffic problems.

A background of the project and the parts involved were discussed; the Itract system and Open Trip Planner were introduced, along with a specification of the project.

The implementation of the trip planner has been described. The trip planner application was divided into two modules: the trip planner itself and the subscriber. The trip planner is similar to OTP, but is compatible with Itract and contains some additional functionality, such as marking bus stops on the map. The subscriber was not available in the OTP application, but was integrated with the new trip planner in the project. The user can start the subscriber through the graphical user interface. Three different subscribers can be started: one to mark vehicle positions, one to get trip updates and one to get alerts. Such data will always be pushed to the subscriber, which is only listening to a number of channels. Thus, no frequent requests are needed.

To set up a publish-subscribe system, where data is pushed to a subscriber, one needs a broker between the publisher and the subscriber. A solution for a broker has been discussed in the thesis. Since the database system Redis is able to match publications and subscriptions based on topics, such as channel names, Redis was chosen as the broker system. To better store persistent data, the database MongoDB was chosen. MongoDB stores data in the BSON, binary JSON, format and provides richer queries than Redis.

The publisher connects to both databases, Redis and MongoDB, to store the GTFS data it receives. The GTFS is converted to a format better suited for the databases as the data is inserted and published. The user can use the trip planner application to get data from the databases, such as a list of his previous trips, and to subscribe via the Redis system to get, e.g., trip updates and traffic alerts.

The new trip planner application for Itract is able to get trip plans from Itract and subscribe to real-time information, as were the goals of this project. Improvements to the trip planner and the way data are stored in the database are possible. Such improvements will be discussed in Section 5.1. The application has not gone through much testing, so there is a possibility of problems to occur. Problems that are already expected will also be mentioned in the following section. These topics are brought up to make further development of the application easier.

5.1 Future Work

There are a few points to be made about the trip planner application developed in the project. These points include additional functionality that has yet not been implemented, but that will prove useful, and some problems or unexpected behaviour that should be mentioned. In the future, the additional functionality can be added and the problems solved to improve the trip planner.

One of the missing options in the trip planner is for the user to be able to select one of his old trips and generate it on the map. For example, if he has once travelled from A to B, he should quickly be able to generate a new trip between A and B without the need to look up these locations on the map. Classes to save and load trips, using the MongoDB database, were implemented with this new functionality in mind. It is at the moment possible for
the user to load a list of previous trips. When a trip is subscribed to, its data will be stored in MongoDB, in a document based on an ID unique for the installation of the application. The list of previous trips contains the absolute start and end locations as well as the times of departure and arrival. In the future, the trip planner could be changed to make the items in the list selectable, enabling the user to generate a new trip between the two locations.

Trip data stored in the database can also be structured so that it is easy to look up statistics about planned trips. This is another area in the project that can be developed further. At the moment, trip information is stored with regard to the user’s wish to see his past trips.

The trip planner fetches graph bounds from the Itract servers. The original OTP application used these bounds to determine whether the user was placing a marker within an available area or not. For the Itract application, the bounds have been used to draw rectangles on the map, mostly for test purposes. The graph bounds are not entirely necessary, since placing the markers in an area of which there is no graph, then planning a trip, will simply yield a message, telling the user that no trip is available. This is a matter of choice that can be made, whether the call to MetadataRequest is really needed.

The publish-subscribe system for the Itract trip planner works successfully, updating the trip planner with real-time data. Some problems may still occur that has yet not been solved. One such problem is that when the application user wishes to see the vehicles in the area displayed on the map, the target channels will be based on the vehicle IDs fetched from the Itract server based on this area. If new vehicles arrive within this area, they may not be subscribed to, unless a new request to the server is made, by, for instance, moving the camera to trigger a new call. Only the vehicle positions within the camera bounds are displayed to avoid unnecessary load. Depending on the zoom of the camera, this area is not always accurate.

The bounds for showing bus stops and vehicle positions might have to be improved, as well as the distance the camera needs to move to trigger another call to showBusStops(). There is yet no formula to determine this distance depending on the zoom level of the camera.

There is likely to be some security issues when accessing MongoDB directly from the Android client. Access to MondoDB should be done through an application server, to improve security. This is a problem that can be solved in future work with the application.
References


