MASTER'S THESIS 2010:XX

Augmented Reality on iPhone

Full implementation of a Client-Server Application.

JEAN-LOUIS GIORDANO

Department of Signals and Systems
Division of Communication Systems
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2010

Augmented Reality on iPhone
Full implementation of a Client-Server Application.
JEAN-LOUIS GIORDANO

Master's Thesis 2010:XX

Signals and Systems
Division of Communication Systems
Chalmers University of Technology
SE-41296 Göteborg
Sweden

Tel. +46-(0)31 772 1000

Cover:
Artwork of the application used on the AppStore, drawn by the author.

Reproservice / Department of Applied Information Technology
Göteborg, Sweden 2010

Augmented Reality on iPhone
Full implementation of a Client-Server Application.
Master's Thesis in Communication Engineering
JEAN-LOUIS GIORDANO
Department of Signals and Systems
Division of Communication Systems
Chalmers University of Technology

Abstract

This thesis work was aimed at implementing an entire Augmented Reality system for the iPhone platform. The application developed was designed to display nearby bus stops in Augmented Reality. This paper shows how to estimate the position of an object within six degrees of freedom using the instruments available on modern smartphones, how to structure a view hierarchy for an Augmented Reality application and how to render objects in 3D using projection matrices.

During this study, a Yaws server has also been implemented for the application to retrieve data. The implemented server aggregate and parse data from various online providers such as Västtrafik, the Göteborg's public transport company. On this server runs Erlang code that manages queries to Mnesia and SQLite databases, parallel threads, http requests and caching. A Rails server is also implemented enabling Ruby to be used for database maintenance.

Keywords: iPhone, Augmented Reality, Realtime, AppStore, Ruby, Yaws, Erlang, Objective-C, Mnesia, SQLite

Contents

Acknowledgements

I would like to thank every person that helped me during this thesis:

- Firstly Nils Svangård who gave me the opportunity to work with ICE House, and an additional thanks for the trip to the Scottish Ruby Conference in Edinburgh,
- Claes Strannegård for accepting to be my examiner,
- A big thank you to David Vrensk and Magnus Enarsson, my two colleagues that helped me all along this thesis. They provided many good advices and tips to perform my work, gave me many books and papers related to my subject and rescued me when I was stuck,
- And eventually a special thanks to my girlfriend who gave me advices and motivation while I was writing this report.

1. Introduction

Augmented Reality is a dream coming true. The concept itself is not brand new and has been developed and studied for many years, but modern technologies and devices are now allowing Augmented Reality to be implemented almost anywhere and be used on a daily basis in a vast range of applications from advertising to medical assistance.

In the mean time, the smart-phones of the last generation have been equipped with sensors that locate the user and enable cleaver and innovative interactive experiences. The iPhone is one of those phones, and taking advantage of its hardware to build an Augmented Reality experience is an interesting challenge.

This master thesis focuses on the design of an application of Augmented Reality on iPhone 3G-S and its implementation from both a Client and Server point of view. This paper presents the work that has been done, starting with the methodology that was used (Chapter 2).

At first an overview over the basic principles of Augmented Reality (Chapter 3) will help understanding what it is, how it works and how it can be used. Then we will have a look at the ″client side″ of the application (Chapter 5) to see the definition of the application from a user′s point of view, and we will go through the implementation on the iPhone to understand the underlying code structure. We will also have a look at the server implementation (Chapter 6) to get an idea of the architecture that is required to provide data in real-time.

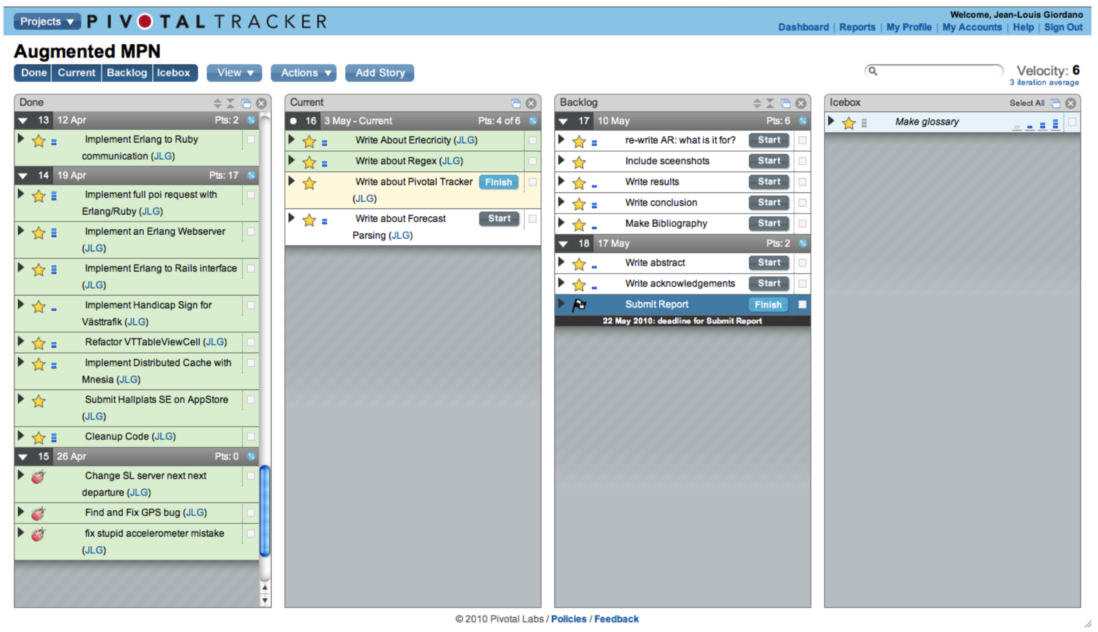
Finally, we will describe the results achieved with the current implementation (Chapter 7), and a view on further improvements will finalize the thesis.

This thesis has been carried at ICE House AB, a development company for web enterprises based in Göteborg.

2. Methodology

2.1. Planning

All along this project, planning and task scheduling have been carried using Pivotal Tracker, an online agile project management tool. With this tool, one can estimate his working velocity to make planning decisions based on past performance. It also enable real-time collaboration so that everyone in the team is able to interact with the planning.



Augmented MPN

Done
13 12 Apr Pts: 2
Implement Erlang to Ruby communication (JLG)
14 19 Apr Pts: 17
Implement full poi request with Erlang/Ruby (JLG)
Implement an Erlang Webserver (JLG)
Implement Erlang to Rails interface (JLG)
Implement Handicap Sign for Västtrafik (JLG)
Refactor VTTableViewCell (JLG)
Implement Distributed Cache with Mnesia (JLG)
Submit Hallplats SE on AppStore (JLG)
Cleanup Code (JLG)
15 26 Apr Pts: 0
Change SL server next next departure (JLG)
Find and Fix GPS bug (JLG)
fix stupid accelerometer mistake (JLG)

Current
16 3 May - Current Pts: 4 of 6
Write About Erlecricity (JLG)
Write about Regex (JLG)
Write about Pivotal Tracker (JLG) Finish
Write about Forecast Parsing (JLG) Start

Backlog
17 10 May Pts: 6
re-write AR: what is it for? Start
Include sceenshots Start
Write results Start
Write conclusion Start
Make Bibliography Start
18 17 May Pts: 2
Write abstract Start
Write acknowledgements Start
Submit Report Finish
22 May 2010: deadline for Submit Report

Icebox
Make glossary

© 2010 Pivotal Labs / Policies / Feedback

Figure 2.1.: Screenshot of the Pivotal Tracker tool

With this tool, a task is represented by a story. A story can be a new feature, a chore, a bugfix or a release date. The last one will appear as a deadline, whereas the others will be represented as a task to be carried out.

When creating a new story, one should evaluate the time to be spent on it. This helps the user to estimate his working velocity. A task should be defined such that it would not last longer than half a day.

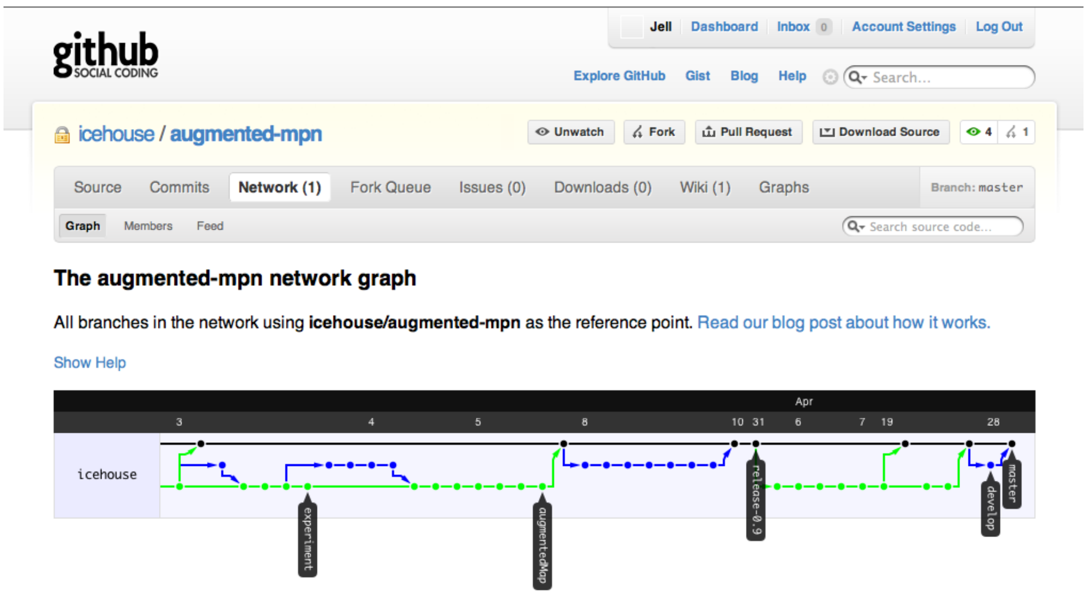
Once the story is added, it is put into an Icebox, which means that it is saved but not added to the planning. This can be useful if the story describes an optional feature or idea. Once the user decide that the task should be carried out someday, the story is put in the backlog. Then when the user starts working on the story, it goes to the Current pipe, and when the task is achieved it is kept in the Done stage.

This is a really convenient tool, and it has been used from the very beginning of this project both as a reporting and scheduling system.

2.2. Version Control

In order to keep a safe and coherent development process, Git has been used as a version control system. Git is a fast and powerful tool that allows complex revision management, including branching and merging with distributed development.

The project is hosted on GitHub, an online Git repository that anyone in the team can access and fork.



The augmented-mpn network graph

All branches in the network using **icehouse/augmented-mpn** as the reference point. Read our blog post about how it works.

Figure 2.2.: Screenshot of the GitHub repository

The commits in the Git repository correspond roughly to the stories defined in the Pivotal Tracker (see previous section).

3. Basic Principles of Augmented Reality

Before diving into the description of this project, one should know what is Augmented Reality. In this chapter, we will try to define Augmented Reality, how it works and how it could be used.

3.1. What is Augmented Reality?

Augmented Reality is a term that is used to describe digital systems that display an artificial visual layer of three- or two-dimensional models upon our natural perception of reality in real-time. It aims at improving or completing our limited way of seeing the real world by presenting artificial elements that would otherwise be hidden to our senses.

The most commonly known Augmented Reality application is the weather forecast program on television. In this system, a reporter is acting in a real environment in front of a blue/green screen, but the spectator watching his TV will see the reporter standing in front of an animated map: the content of the background consists of virtual data mapped on a real world object in real-time. This is Augmented Reality.

It is during the year 1992 that a Boeing researcher named Tom Caudell first defined ″Augmented Reality″ in a paper called ″Augmented reality: an application of heads-up display technology to manual manufacturing processes″ [Cau 92]. The term was used to describe a digital system that displays virtual graphics onto a physical reality.

Yet the concept of Augmented Reality itself is older than its name, since the first Augmented Reality device was build and presented in a 1968 paper by Ivan Sutherland [Sut 68]. The system was designed to track the head position of the user to project before his eyes two-dimensional models in order to create an illusion of three dimensions. It relied on the idea that moving perspective images would appear in three dimensions even without stereo presentation, a principle which is called the ″kinetic depth effect″. In such a system, the user will perceive the object as three-dimensional only when he moves, otherwise the object will only be seen as planar.

It has to be noted that Augmented Reality differs from Virtual Reality. In Virtual Reality, there are no elements of Reality, as much as in Reality there are no elements of Virtuality by definition. In his paper called "Augmented Reality: A Class of Displays on the Reality-Virtuality Continuum" [Mil 94], Milgram defines the Reality-Virtuality Continuum shown in Fig. 3.1.

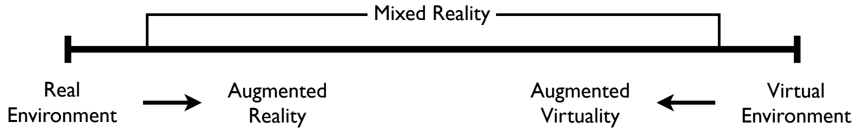


Figure 3.1.: Milgram's Reality-Virtuality Continuum [Mil 94]

As can be seen from this diagram, Augmented Reality is based upon reality and tends to Virtuality in an area called "Mixed Reality". It lies near the real world end of the line, which means the predominate perception is the real world. On the symmetric position, Milgram defines Augmented Virtuality. Behind this concept lie systems where real world items are added to a Virtual Environment, such as texture mapping on virtual objects in video games.

So the limits of Augmented Reality are defined by the furthest one can go at adding virtual data on a real world representation without depriving the user of its predominate perception of Reality.

3.2. How does it work?

In order to get an Augmented Reality application, one must integrate artificial objects into a real scene. Of course, such objects must be rendered to scale and at the correct position and orientation. To do so, the system requires to know the position of the camera or the user from the object. This is the core principle of Augmented Reality.

To solve this problem, there are two main approaches:

- Use sensors to know the position of the object from the camera

By the mean of tags on a real object, one can get a coordinate system on which to project a virtual model. Those tags can be visual or based on any technology that can be use to locate that object, for example a set of Infra-Red diodes.

- Use sensors to know the position of the camera in a known space

If the environment is known, i.e. the application is internally aware of the position of the object, or if the object is fully virtual and does not rely on any physical counterpart, one can evaluate the position of the camera in this environment and render the object accordingly. The position of the camera can be known by the mean of a large enough set of sensors that will be able to position it in the environment space.

Once the virtual environment is known, each virtual object must be rendered on top of an existing view by the mean of geometric projections that fit the estimated position of this object in the virtual space. The virtual object can also be projected onto a real world object such as a screen or an holographic system.

3.3. What is it for?

Augmented Reality has been used in domains where the purely visible environment is not sufficient or satisfactory and has to be completed by additional information in a non-obstructive way. One of the first application was in head-up displays for military aircraft as shown in Figure 3.2.

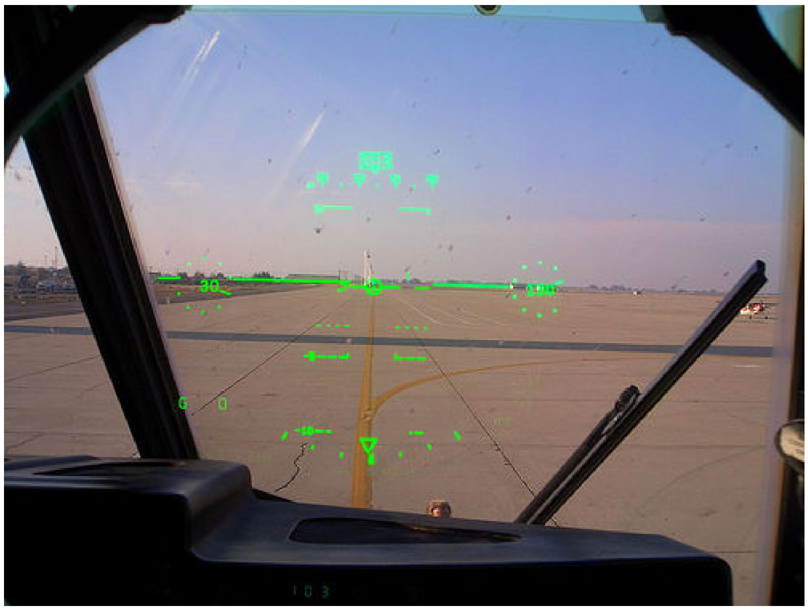
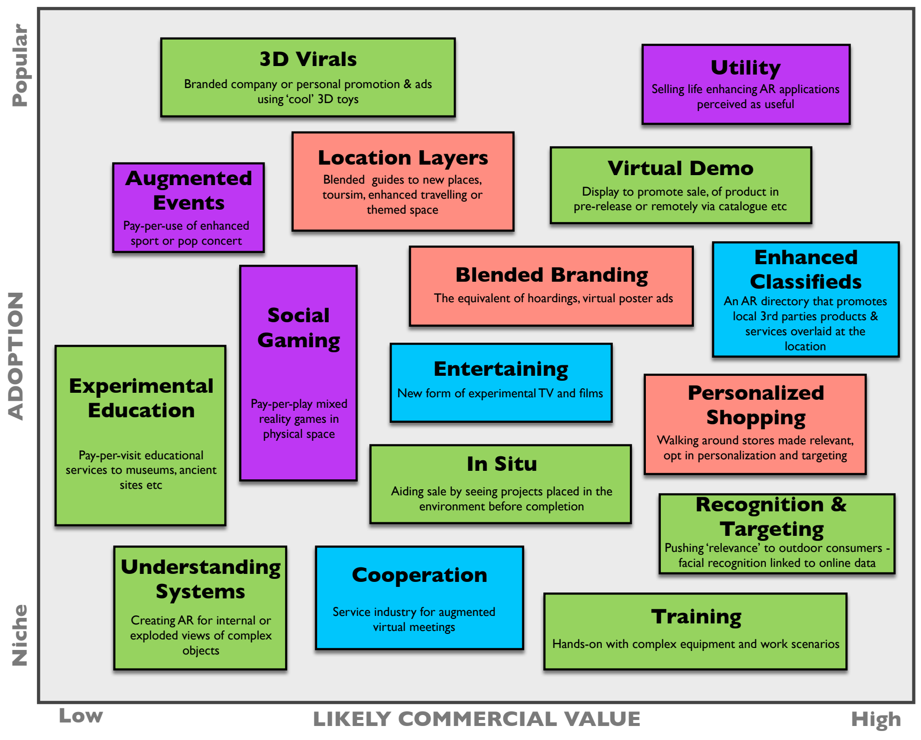


Figure 3.2.: An Head-up display in a military plane [Tel 05]

Many applications have also been developed for surgery and medical imagery, presenting serious advantages over non-assisted methods.

But nowadays, we also see appearing more casual applications, especially for marketing purposes.

In a recent web publication [Hay 09], an interesting study on an Augmented Reality business model has been proposed by Gary Hayes. In his study, he tries to categorize all the exploitable applications of Augmented Reality. Figure 3.3 shows an attempt at classifying the different approaches at Augmented Reality applications according to their potential as marketable products.



Popular
ADOPTION
Niche
Low
LIKELY COMMERCIAL VALUE
High
3D Virals
Branded company or personal promotion & ads using 'cool' 3D toys
Utility
Selling life enhancing AR applications perceived as useful
Augmented Events
Pay-per-use of enhanced sport or pop concert
Location Layers
Blended guides to new places, toursim, enhanced travelling or themed space
Virtual Demo
Display to promote sale, of product in pre-release or remotely via catalogue etc
Blended Branding
The equivalent of hoardings, virtual poster ads
Enhanced Classifieds
An AR directory that promotes local 3rd parties products & services overlaid at the location
Social Gaming
Pay-per-play mixed reality games in physical space
Experimental Education
Pay-per-visit educational services to museums, ancient sites etc
Entertaining
New form of experimental TV and films
Personalized Shopping
Walking around stores made relevant, opt in personalization and targeting
In Situ
Aiding sale by seeing projects placed in the environment before completion
Recognition & Targeting
Pushing 'relevance' to outdoor consumers - facial recognition linked to online data
Understanding Systems
Creating AR for internal or exploded views of complex objects
Cooperation
Service industry for augmented virtual meetings
Training
Hands-on with complex equipment and work scenarios

Figure 3.3.: 16 Augmented Reality Business Models by Gary Hayes [Hay 09]

In this thesis, we tried to fit into the "Utility" business model, being the most promising one according to this analyse (popular and high likely commercial value).

4. The Project

Now that we have the sufficient knowledge on Augmented Reality, let's have a look at the project in itself. In this Chapter, we shall see how the iPhone fits the requirements for Augmented Reality and how our project takes advantage of this.

4.1. The iPhone

The iPhone is a last generation smartphone developed by Apple and was revealed for the first time in 2007. It features advanced functionalities such as multimedia support, internet access, video games and many more. Up to this day, forty-two millions iPhones have been sold across the globe.

Figure 4.1.: Picture of an iPhone 3G-S [App 10]

Its main characteristic is that its User Interface (UI) is almost only based on two inputs:

- a large multi-touch screen
- a 3-axis accelerometers

As such, the iPhone does not possess any physical keyboard. Instead, a virtual keyboard can be summoned any time it is necessary, leaving a larger screen for applications.

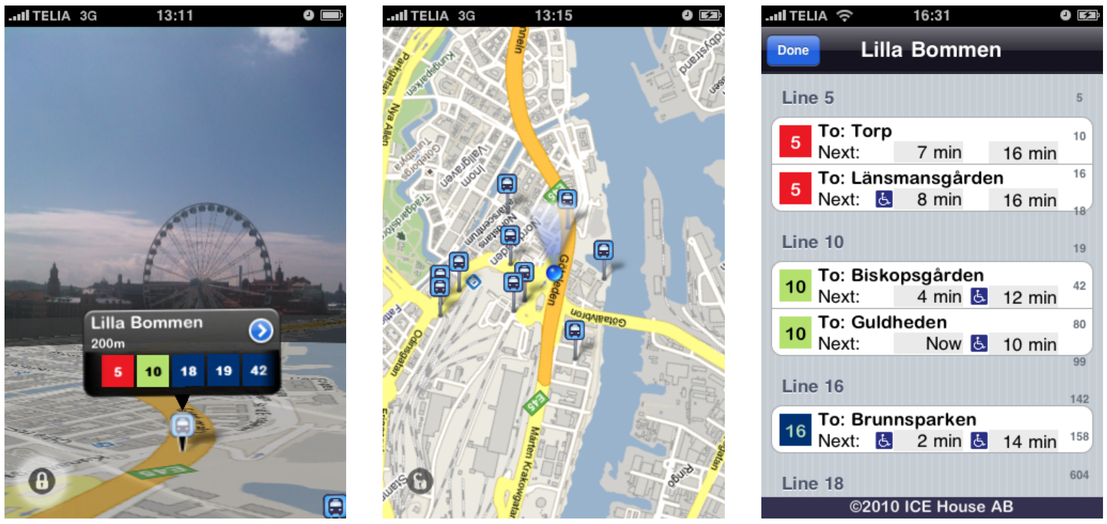
The last generation of iPhone, the iPhone 3G-S, also includes a GPS, a compass, and a 3-megapixels camera, which seems to make it a perfect candidate for any Augmented Reality application. Unfortunately, the pragmatic access to the camera is not yet fully available, so image recognition is not yet an option, thought an upcoming version of its Operating System (OS) will allow the access to the video data of the camera.

In order to install new applications on an iPhone, one must download it from the AppStore, the internet-based retail store for iPhones, iPods and the iPad. Apple having a highly proprietary policy on its product, this is actually the only way to distribute iPhone applications.

4.2. The Application

4.2.1. What it does

Our application has been designed for travellers in Sweden that are looking for a nearby public transport station. Augmented Reality makes it very intuitive to find a nearby stop: the user just has to "look around" with his iPhone, and the closest stops will appear at their positions.



Lilla Bommen

Line 5

5	To: Torp		
	Next:	7 min	16 min
5	To: Länsmansgården		
	Next:	♿ 8 min	16 min

Line 10

10	To: Biskopsgården		
	Next:	4 min	♿ 12 min
10	To: Guldheden		
	Next:	Now	♿ 10 min

Line 16

16	To: Brunnsparken		
	Next:	♿ 2 min	♿ 14 min

Line 18

5 10 16 18 19 42 80 99 142 158 604

©2010 ICE House AB

(a) Augmented Reality View (b) Map View (c) Information View

Figure 4.2.: Screenshots of the Application

When the user holds his iPhone toward one direction, a virtual Map is projected on the estimated ground, and on it a set of three-dimensional pins indicating nearby stops. All this is done in realtime and fits the position of the iPhone within six degrees of movement.

To make it even easier for the user, a flat Google Map is available when the device is held horizontally (see Figure 4.2(b)). The flat map also displays pins representing the nearby stops. The Map is always oriented in the direction of the user for more convenience. This can be useful when directions are required.

If the user press a pin in either Augmented Reality or Flat Map, a bubble will appear with the stop name, its distance from the user and the Bus/Tram/Boat/Subway line numbers it serves (see Figure 4.2(a)). If this stop is interesting to him, the user can check the next departures by pressing the arrow in the bubble.

When pressing this arrow, the application flips horizontally and displays the next departures in a scrollable view, with indexes on the side for quick navigation (see Figure 4.2(c)). This is a really efficient way to check for the next departures: only two "clicks" are required to access the desired data.

To refresh data, the user can shake his iPhone. A new request over the Internet will be sent and the stops will be updated with their latest values. It might be useful to refresh data if the user has changed location, or if he has been waiting for a while and wants to see the updated forecasts.

Also, if the image is not stable enough, the user can press the "lock" icon to freeze animation in order to interact more comfortably with the application. This is particularly useful when walking.

Augmented Reality makes the application very interesting for people that are uncomfortable at reading maps when discovering a new city.

But this application has also been designed for people that are already familiar with the city they visit. Simply by pointing at a stop they are interested in, they can get the next departures without wasting their time by going to a stop in order to consult its timetable.

For now, the application is available for Göteborg and Stockholm, with their respective public transport companies Västtrafik and Storstockholms Lokaltrafik. But additional providers could be added later on.

The application is available in English, Swedish, French and Chinese and can be downloaded from the AppStore under the name ”Hållplats SE”.

4.2.2. How it works

The application takes advantage of the GPS capabilities of the iPhone to locate the user on a map. Then thanks to the compass, we are able to estimate the direction he is looking at, and the 3-axis accelerometer allows us to evaluate the angle at which he holds his phone. Note that this application requires a GPS and a compass, and therefore is only compatible with the iPhone 3G-S.

Built-in APIs makes it easy to access the accelerometer, compass and GPS data. When an update to either of these sensors is generated, a message is sent to their respective delegates in a way that could be compared to Events in Java.

Once the position of the user is known, a request is made over the internet to determine the bus stops that are close to him. The answer to this request will be a list of stops with their names and locations together with a forecast list for each of them.

An item in the forecast list is composed of a line number, a destination and a set of attributes indicating the time of departure plus some information on the quality of the travel.

Once the positions of the bus stops are known, we can project them in the virtual space according to the user’s coordinates, heading direction and phone holding angle. Our implementation of Augmented Reality hence follows the second method defined in Section 3.2: we use sensors to know the position of the camera in a known space.

To cope with the precision errors of acceleration and heading measurement, a buffer is used to get the average values on an arbitrary period of time before updating the knowledge on the user’s position. Animations are also used to make transitions between two views appear smoother.

The accelerometer let us know when the iPhone is in flat position, allowing us to switch between Map or Augmented View. It also detects when a shake gesture which will trigger the data refresh.

5. Implementation on the Client Side

This Chapter describes in detail the implementation on the client side, which corresponds to the application that is installed on the iPhone.

5.1. The Development Tools

To develop an application for iPhone, a specific Framework is required. We will have a look at its nature in this section.

5.1.1. Objective-C

First of all, any software developed for iPhone must be programmed in Objective-C, although it is possible to call C and C++ functions from code.

Objective-C is an Object-Oriented (OO) reflexive programming language build upon the C language. It is comparable to C++ from this perspective, but differs greatly in many ways, especially in its dynamic message passing system, in being weakly typed and in being able to perform dynamic loading. In its latest version, Objective-C also features a Garbage Collector which subtracts the programmer from advanced memory management considerations. Unfortunately, this feature is not available for iPhone development.

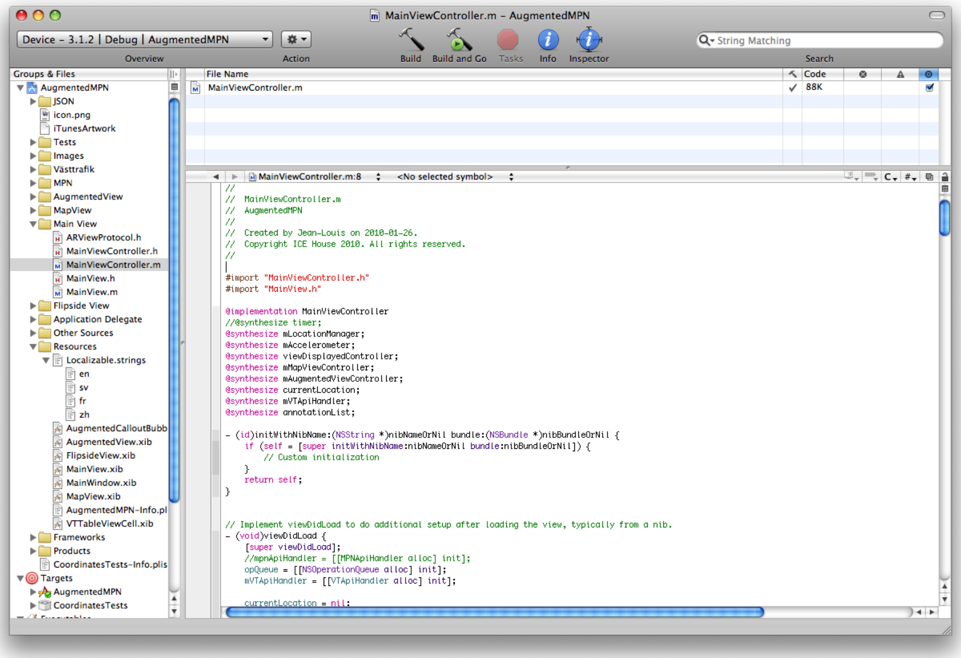
As mentioned above, one of the interesting features of Objective-C is its message passing in lieu of method call for an object. In Objective-C terminology, calling a Method from an Object consists instead of sending a Message to a Receiver. The following example send a message ”say” with parameter @”hello world!” to the receiver named ”receiver”, and the answer ”result” is of type ”BOOL”.

```
BOOL result = [receiver say:@"hello world!"];
```

This is very close to the elegant Smalltalk messaging system. In practice, the Receiver is an object ”id”, which is similar to an anonymous pointer to an object. When messaging a receiver, the object sending a message does not need to know if the object it is sending to will be able to handle its request or not. This is really convenient since the sender does not need to be aware of the receiver. This system allows complex interactions between object.

5.1.2. iPhone SDK

In order to compile code for the iPhone, the use of Xcode as Integrated Development Environment is almost unavoidable. Apple has a highly proprietary approach for its products, and programming for an Apple environment is much restrictive to this regard. Fortunately, Xcode and the set of tools provided by Apple offer a great comfort of use in many cases, especially for debugging, or for creating interfaces with the use of the Interface Builder (IB).



```
//
//  MainViewController.m
//  AugmentedMPN
//
//  Created by Jean-Louis on 2010-01-26.
//  Copyright ICE House 2010. All rights reserved.
//

#import "MainViewController.h"
#import "MainView.h"

@implementation MainViewController
//@synthesize timer;
@synthesize mLocationManager;
@synthesize mAccelerometer;
@synthesize viewDisplayedController;
@synthesize mMapViewController;
@synthesize mAugmentedViewController;
@synthesize currentLocation;
@synthesize mVTApiHandler;
@synthesize annotationList;

- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil {
    if (self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil]) {
        // Custom initialization
    }
    return self;
}


// Implement viewDidLoad to do additional setup after loading the view, typically from a nib.
- (void)viewDidLoad {
    [super viewDidLoad];
    //mpnApiHandler = [[MPNApiHandler alloc] init];
    opQueue = [[NSOperationQueue alloc] init];
    mVTApiHandler = [[VTApiHandler alloc] init];

    currentLocation = nil;
```

Figure 5.1.: Screenshot of XCode, the Apple IDE

Once Xcode is set up, the iPhone Software Development Kit must be installed. The iPhone SDK takes advantage of all the features of Xcode, including its set of external tool to the largest extent. After being registered to the iPhone developer's program, Xcode can also be linked directly to an actual iPhone device in order to test an application in real-time with any monitoring tool available.

The SDK contains Application Programming Interfaces (API) to all the accessible libraries of the iPhone, including Cocoa. Unfortunately, many functionalities such as the access to raw data from the video camera are not part of a public API, so their libraries are considered as not accessible and therefore their use is forbidden by Apple.

5.2. View Hierarchy

When developing for iPhone, the Model-View-Controller (MVC) is the best architecture to go with. There are well defined classes for views and controllers in the Cocoa environment, so we took advantage of that when designing our application.

When using the Cocoa environment, each view must be a subclass of the UIView class, and each controller of the UIViewController class. A view is automatically linked to a view controller.

A view can contain other views that will be considered as its subviews, and each view has a frame in which it renders itself. Views can have transparent or semi-transparent backgrounds, so it is possible to create a complex view hierarchy without hassle.

Figure 7.1 shows the view hierarchy for our application.

In order to render a view on screen, one must define an application window that is a subclass of UIWindow and that will take care of rendering every visible view on the iPhone's screen. By definition, a UIWindow must be the higher node in any view hierarchy.

Before further considerations, one should know that there is a serious restriction on the iPhone SDK up to the version 3.2 which prohibits to insert a camera view (UIImagePickerView) as a subview to a UIView. Instead, it must be directly put into a UIWindow. Since we want to use the camera in our application, we have no choice but to put it on top of our view stack. This is semantically wrong, but there are no alternatives left here by Apple.

Furthermore, the camera view itself is locked, so we can't help but use a view called "Camera Overlay" to display any view on top of it.

As a result, the camera is always rendering in the background of our application, and everything else is rendered on top of it.

Now we have a window and a background with a camera view, and we want to be

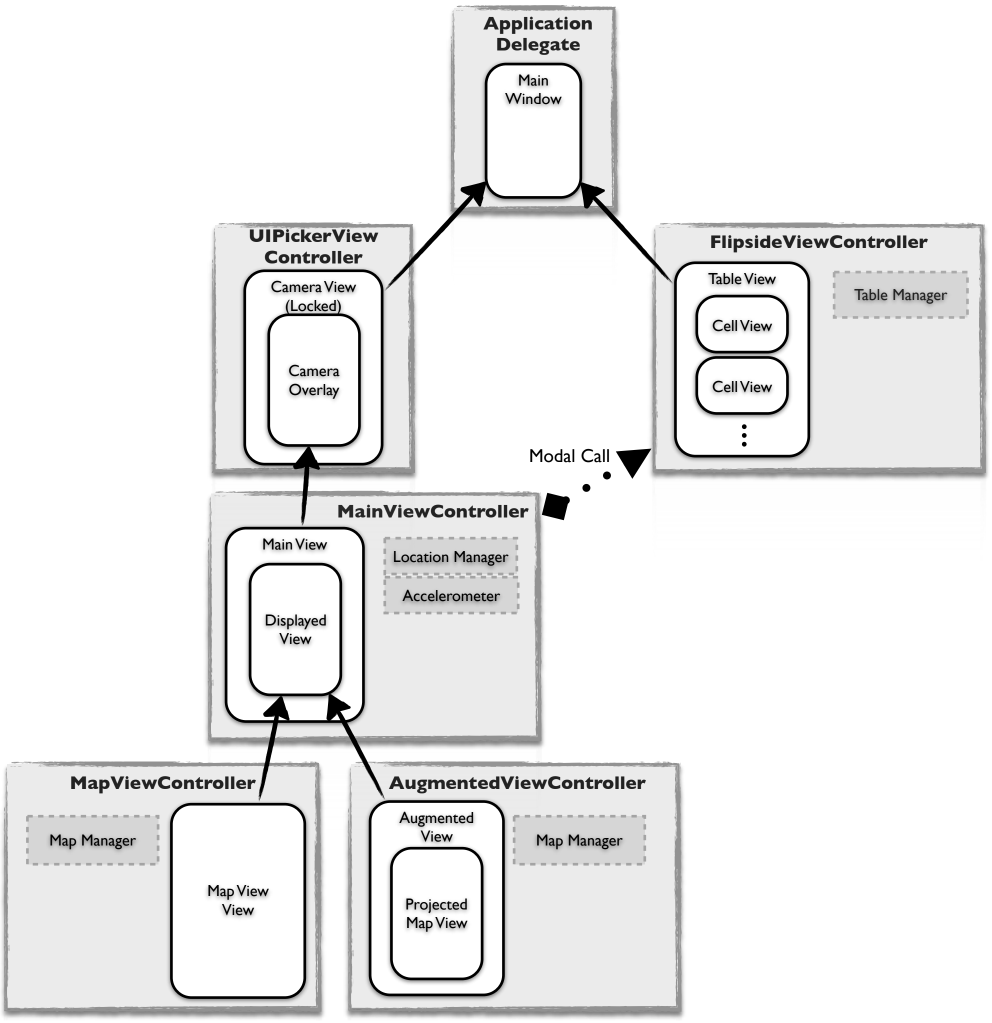


Figure 5.2.: Scheme of the View Hierarchy of the application

able to switch between a Map View and an Augmented Reality View. To do so, we implement a Main View that will display either of them according to the iPhone position. In order to avoid redundancy, all the data of the application is handled by the Main View Controller and dispatched in its subview, be it a Map View or an Augmented View. The same goes for the acceleration and location data.

To present the forecast of a stop, a ”modal view” is opened by the Main View Controller. The display of a modal view will replace the content of the actual window by a temporary view. When dismissing this view, the previous content of the window is loaded again and the application continues.

This modal view is a table view itself composed of view cells that are respectively subclasses of UITableView and UITableViewCell. Those view cells are used to display the forecasts list for a bus stop.

5.3. Positioning within 6 degrees of freedom

In our application, we need to locate the camera of the iPhone within 6 degrees of freedom:

- Latitude (φ)
- Longitude (λ)
- Altitude
- Azimuth (θ)
- Roll angle (α)
- Pitch angle (β)

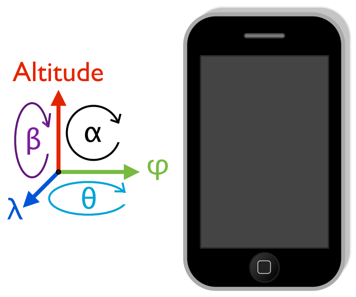


Figure 5.3.: the 6 degrees of freedom of the iPhone

Those degrees of freedom are illustrated in Figure 5.3. The GPS directly gives us the Altitude, Latitude and Longitude of the user whereas the compass gives us his Azimuth. This directly takes care of 4 out of the 6 degrees.

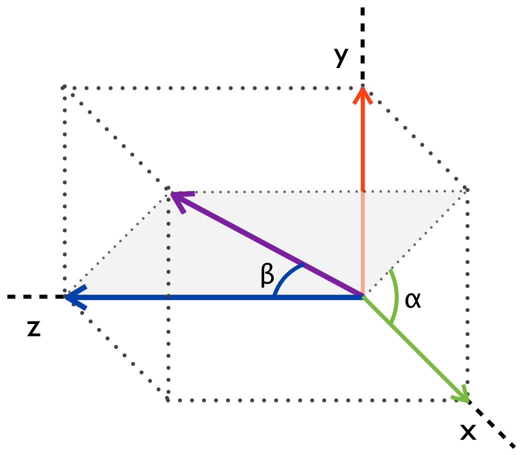


Figure 5.4.: Projection giving the Roll and Pitch angles, respectively α and β

But thanks to the 3-axis accelerometer, the roll angle and pitch angle can easily be computed by the mean of simple trigonometry as shown in Figure 5.4. To compute α and β, we use the following equations:

$$\alpha = \text{atan2}(x, y) \tag{5.1}$$

$$\beta = \text{atan2}\left(\sqrt{y^2 + x^2}\,, z\right) \tag{5.2}$$

Where x, y and z are respectively the accelerations values on the X-, Y- and Z-axis an atan2(a,b) is a function that gives the angle in radians between a vector (a , b) and the X-axis.

5.4. 3D Projection

Now that we know the position of the camera relatively to the object we want to display, we must use our knowledge to render the Augmented View.

5.4.1. Rendering the bus stops

We project the Bus Stops on the plane defining the ground to begin with. The iPhone has an API to perform 3D projections, by the mean of layer transformations. To apply a transform to a layer, one must provide the transform matrix corresponding to the desired projection.

So to render a Bus Stop at the proper position, one must provide a 4×4 projection matrix. The first step is to compute the distance and the azimuth of the stop relatively to the camera.

To compute the distance d, we use a simplified version of the great-circle formula as shown in Equation 5.3 where φ_s and λ_s are the Latitude and Longitude of the standpoint (camera position), φ_f and λ_f the ones of the forepoint (bus stop location) and R is the average radius of Earth ($\approx$ 6371km).

$$d = R \times \arccos\left(\sin\varphi_s \sin\varphi_f + \cos\varphi_s \cos\varphi_f \cos(\lambda_f - \lambda_s)\right) \tag{5.3}$$

Then to get the relative azimuth θ_r, we use the formula shown in Equation 5.4. In this formula, atan2(x,y) is a function that gives the angle in radians between the vector (x, y) and the X-axis.

$$\theta_r = \text{atan2}\left(\sin(\lambda_f - \lambda_s)\cos\varphi_f\,, \cos\varphi_s \sin\varphi_f - \sin\varphi_s \cos\varphi_f \cos(\lambda_f - \lambda_s)\right) - \theta \tag{5.4}$$

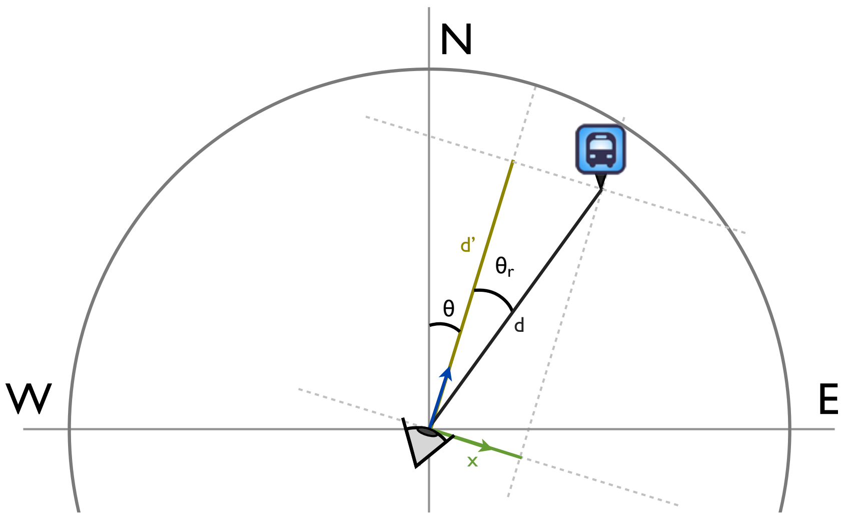


Figure 5.5.: Viewpoint orthogonal to the ground

Now that we have the distance and the azimuth of the stop relatively to the camera position, we can use them as polar coordinates on the ground plane centred on the user's location as shown in Figure 5.5.

From this, we get:

$$x = d \cos\theta_r \tag{5.5}$$
$$d' = d \sin\theta_r \tag{5.6}$$

where x is the x-coordinate of the projected stop in the camera coordinate system and d' is the distance of the projected Bus Stop position on the YZ-plane. Now that might sound confusing, so in Figure 5.6 we represent a viewpoint orthogonal to the YZ-plane.

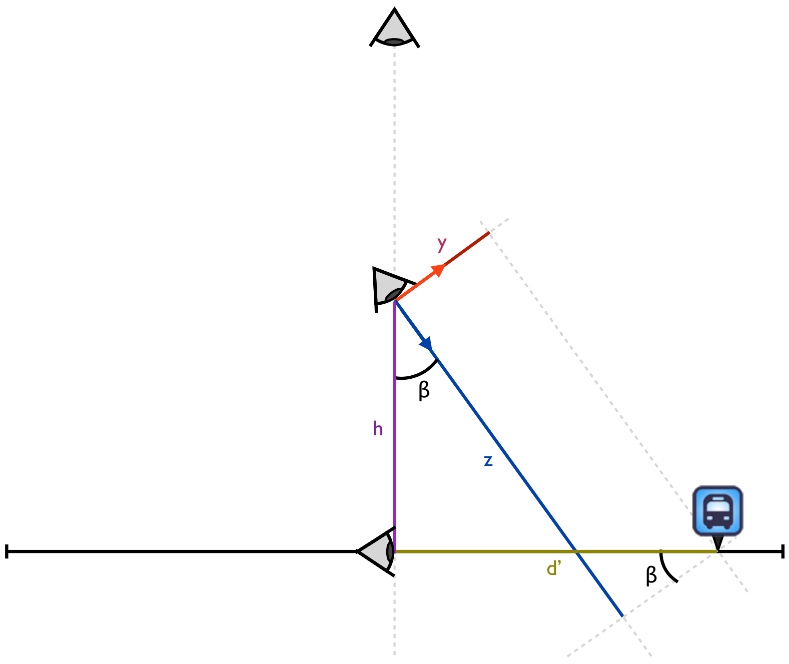


Figure 5.6.: Viewpoint orthogonal to YZ-plane

In this figure, h represents the height of the camera to the ground. We make it dependent on β so that when the device is vertical ($\beta = \pi/2$) the ground appears to be at the user's feet, and when the device is held horizontal it will seem that he is seeing the ground from the sky. A simple enough approach is to take:

$$h = h_{min} + (h_{max} - h_{min})\sin\beta \tag{5.7}$$

So what we want now is to find the y and z coordinates. Figure 5.7 helps us to see more clearly the trigonometric problem we get to solve.

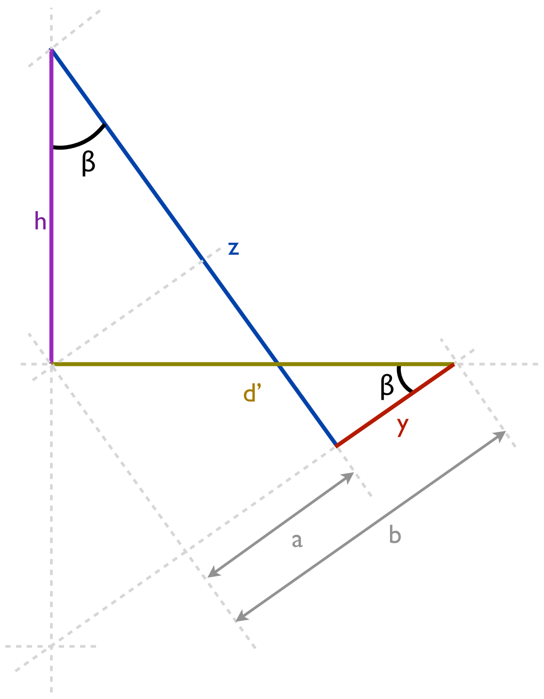


Figure 5.7.: Schema of the Trigonometric Problem

From this schema, we find:

$$y = b - a \tag{5.8}$$
$$b = d' \cos\beta \tag{5.9}$$
$$a = h \sin\beta \tag{5.10}$$
$$y = d' \cos\beta - h \sin\beta \tag{5.11}$$
$$y = d \sin\theta_r \cos\beta - h \sin\beta \tag{5.12}$$

and:

$$z = (a + h)\cos\beta \tag{5.13}$$
$$a = d' \tan\beta \tag{5.14}$$
$$z = (\frac{\sin\beta}{\cos\beta} d' + h) \cos\beta \tag{5.15}$$
$$z = d' \sin\beta + h \cos\beta \tag{5.16}$$
$$z = d \sin\theta_r \sin\beta + h \cos\beta \tag{5.17}$$

So we finally got the full coordinates of the Bus Stop in the camera coordinate system, which are the following:

$$x = d\ \cos\theta_r \tag{5.18}$$

$$y = d\ \sin\theta_r\ \cos\beta - h\ \sin\beta \tag{5.19}$$

$$z = d\ \sin\theta_r\ \sin\beta + h\ \cos\beta \tag{5.20}$$

Now we have to find the projection matrix for this position. This is the easy step: we just have to translate the view with coordinates $(0, 0, 0)$ by a vector (x, y, z). We have to use a homogeneous projection matrix for the "Tranform3D" API. This corresponds to the following matrix:

$$T = \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & -1/e_z & 1 - z/e_z \end{pmatrix} \tag{5.21}$$

where e_z is the distance of the user from the displayed surface. We used an arbitrary value of 900 fitting the viewing angle of the iPhone camera.

There is one last little change to make to this projection matrix. Indeed, the projected stop size will be proportional to its distance to the camera, thus it can become infinitely large or infinitely small. To counter this effect, we scale it back to a normal size, hence we use the following matrix:

$$M_{bus} = \begin{pmatrix} 1 - z/e_z & 0 & 0 & x \\ 0 & 1 - z/e_z & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & -1/e_z & 1 - z/e_z \end{pmatrix} \tag{5.22}$$

5.4.2. Rendering the ground Map

Now we want to render the Map on the estimated ground plane. The first step is to find the projection matrix to centre the map at the proper location, which is equivalent to projecting a Bus Stop with a distance $d = 0$. This gives the following matrix:

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & h\ \sin\beta \\ 0 & 0 & 1 & -h\ \cos\beta \\ 0 & 0 & -1/e_z & 1 \end{pmatrix} \tag{5.23}$$

Then we want to rotate the map on the Z-axis with θ to get the correct orientation, which corresponds to this matrix:

$$R_1 = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.24}$$

Then we need a rotation of β on the X-axis, which corresponds to the following matrix:

$$R_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\beta & -\sin\beta & 0 \\ 0 & \sin\beta & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.25}$$

Eventually, to get our final transformation matrix M we have to compute:

$$M_{map} = T \times R_1 \times R_2 \tag{5.26}$$

This gives the following matrix:

$$M_{map} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta\cos\beta & \cos\theta\cos\beta & -\sin\beta & h\sin\beta \\ \sin\theta\sin\beta & \cos\theta\sin\beta & \cos\beta & -h\cos\beta \\ -\sin\theta\sin\beta/e_z & -\cos\theta\sin\beta/e_z & -\cos\beta/e_z & 1 \end{pmatrix} \tag{5.27}$$

5.4.3. Taking into account the Roll angle

The final step of the projection is to take into account the roll angle. Since this transform is the same for all of the projected views (Bus Stop and Map), we use a trick to avoid making the previous transform matrices even more complicated: we render them in a container view that we then rotate with the roll angle α (See Figure 5.8).

So the transform to apply to the container is a simple rotation of α on the Z-axis, which gives the following projection matrix:

$$M_{container} = \begin{pmatrix} cos\alpha & -sin\alpha & 0 & 0 \\ sin\alpha & cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.28}$$

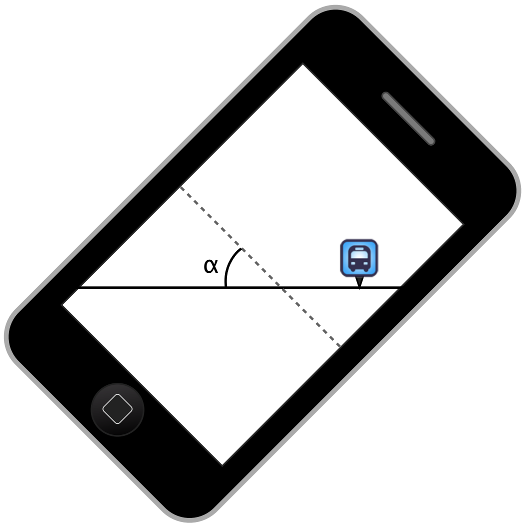


Figure 5.8.: Rendering the roll angle

5.4.4. Summary

To sum up, we first project each bus stop using this matrix in a container view:

$$M_{bus} = \begin{pmatrix} 1 - z/e_z & 0 & 0 & x \\ 0 & 1 - z/e_z & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & -1/e_z & 1 - z/e_z \end{pmatrix} \tag{5.29}$$

Where:

$$x = d\,\cos\theta_r \tag{5.30}$$

$$y = d\,\sin\theta_r\,\cos\beta - h\,\sin\beta \tag{5.31}$$

$$z = d\,\sin\theta_r\,\sin\beta + h\,\cos\beta \tag{5.32}$$

Then we project the map in the container using this matrix:

$$M_{map} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta\,\cos\beta & \cos\theta\,\cos\beta & -\sin\beta & h\,\sin\beta \\ \sin\theta\,\sin\beta & \cos\theta\,\sin\beta & \cos\beta & -h\,\cos\beta \\ -\sin\theta\,\sin\beta/e_z & -\cos\theta\,\sin\beta/e_z & -\cos\beta/e_z & 1 \end{pmatrix} \tag{5.33}$$

And finally we render our container view with this matrix:

$$M_{container} = \begin{pmatrix} cos\alpha & -sin\alpha & 0 & 0 \\ sin\alpha & cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.34}$$

6. Implementation on the Server Side

In order to retrieve information about nearby stops, the client must request a remote source of information by calling a server. The server that has been implemented in our case acts as a middle-ware between the public transport providers (Västtrafik and SL) and the client. In that way, requests to the different providers are transparent to the client.

6.1. Architecture

The server processes a request for nearby bus stops as follows:

1. receive a request with the user's coordinates (Latitude and Longitude),
2. find the 10 nearest bus locations from the given coordinates in its SQLite database,
3. for each of these stops, launch a light-weighted process that:
   a) if valid cached data is available in its Mnesia database, returns this data as forecast
   b) otherwise, fetch forecast data from the stop's provider
4. once each forecast is obtained, parse the result into JSON format,
5. finally, send a reply to the client containing the JSON data.

Figure 6.1 shows a simplified request processing on an UML sequential diagram, the grey area representing our server. The sequence is simplified because it describes a process that only returns three stops instead of ten, plus it does not explicitly shows the caching mechanism, but it is still a good representation of what is happening.

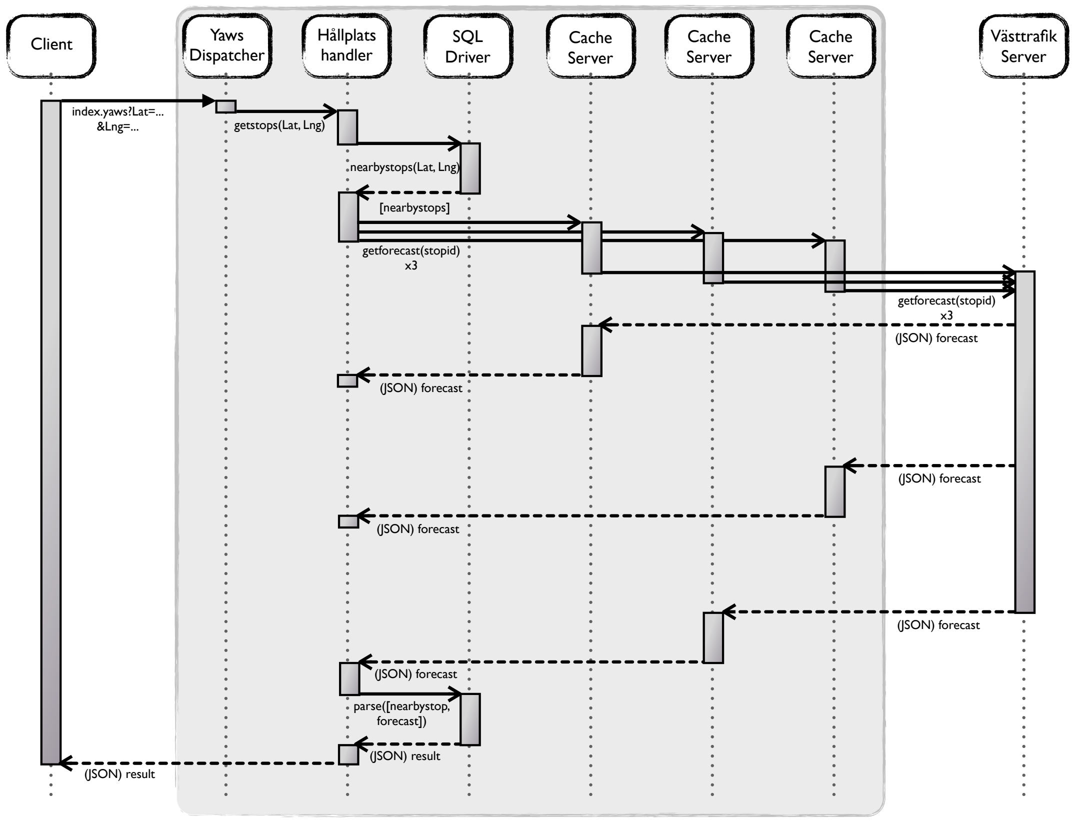


Figure 6.1.: UML sequential diagram of the request processing

This diagram follows the classic UML notation, so plain arrows represent synchronous calls, stick arrows represent asynchronous calls, and dashed arrows represent return values.

The application is hosted on a Yaws web server installed on a Debian (Ubuntu) virtual machine from the Amazon cloud web services. Yaws is the Erlang alternative to web servers, and as such offers high availability and high scalability.

In order to handle high quantities of lightweight processes, the best solution was to use Erlang. Erlang is a functional programming language especially designed to handle high concurrency and distributed systems. A large part of the application is therefore implemented in Erlang in order to take advantage of this .

But to keep the application easy to maintain and update, the more complex tasks are implemented in Ruby. Ruby is a modern dynamic scripted language, and has been designed to make the developers happy. As such, it is a powerful tool to implement complex algorithms painlessly. The ruby part is hosted on a rails web server, and communicates with the Yaws server through Erlang Ports via the Erlang Binary Protocol.

Figure 6.2 shows the details of the server's architecture.

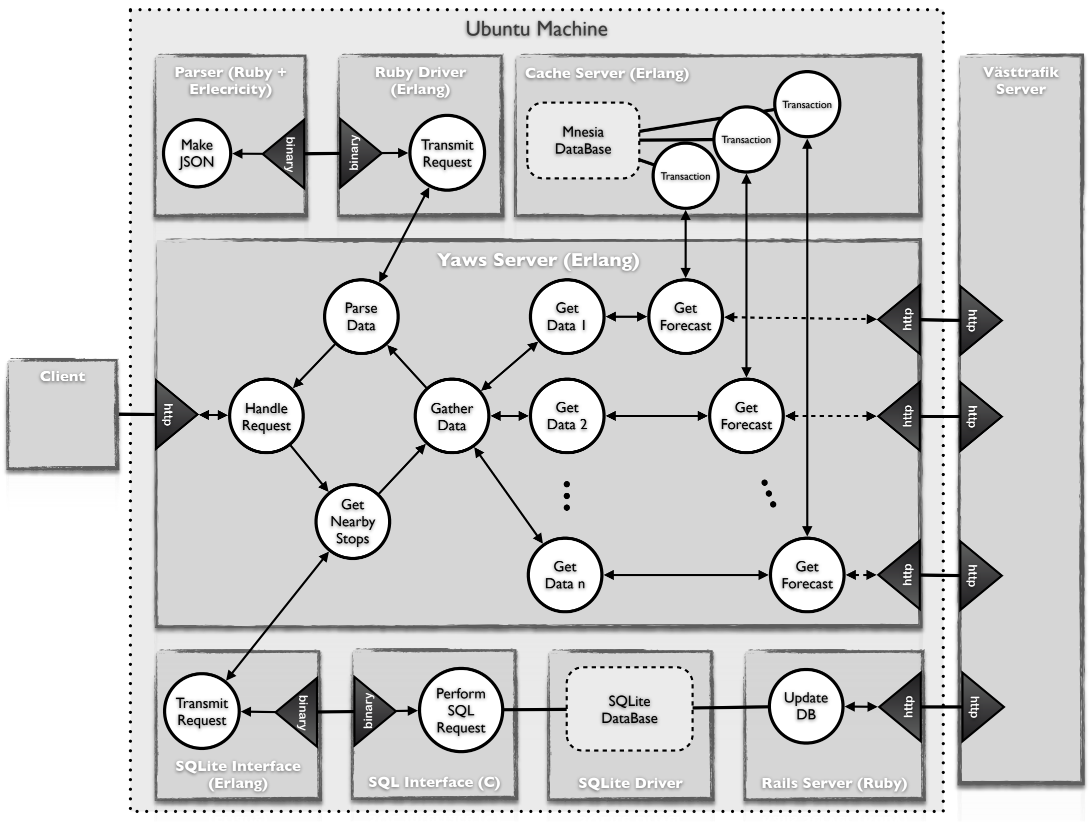


Figure 6.2.: Scheme of the Server Architecture

In this figure, each circle represents an independent process and the arrows indicate message passing between processes. The dashed rounded rectangles represent databases servers. Black triangles represent communication ports.

6.2. Erlang

Erlang is used to dispatch the lightweight processes, to query the SQLite database, to perform the distant requests to Västtrafik and Storstockholms Locaktrafik, but also to take care of data caching via a Mnesia database. This obviously represents the larger part of the server's application.

The complete implementation of the Erlang parts is given in Appendix A.

6.2.1. Concurrent Programming

As mentioned in Section 6.1, Erlang is a functional language, and as such differs greatly from classic programming paradigms. In Erlang, functions are treated in a pure mathematical way: variables are immutable and functions have no side effects. This may seems like a very handicapping restriction, but that is what makes Erlang so powerful at smoothly handling thousands of concurrent processes without having problems of dead locks, starvation nor memory corruption.

Being a functional language, Erlang allows function recursion with very little overhead. It also feature strong pattern matching capabilities. The piece of code shown in Figure 6.3 is an implementation of the factorial function programmed in Erlang.

```
-module(example).
-export([factorial/1]).

factorial(0) ->
    1;
factorial(N) ->
    N * factorial(N-1).
```

Figure 6.3.: Implementation of the factorial function in Erlang

From this snippet we can see that the Erlang implementation is very close to the mathematical definition of factorial:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

This is made possible thanks to pattern matching, and it is efficient thanks to tail recursion optimization.

But the real power of Erlang reside in its messaging system that allows complex asynchronous calls. When a process sends a message to another process, the message is put into the receiver's message box to be treated whenever it is available. By doing so, message passing is a totally non blocking procedure.

```
-module(example).
-export([start/0, ping/0]).

ping() ->
  receive
    {ping, Pong_PID} ->
      io:format("Ping!~n", []),
      Pong_PID!pong,
      ping()
  end.

start() ->
  Ping_PID = spawn(example, ping, []),
  Ping_PID!{ping, self()},
  receive
    {pong} ->
      io:format("Pong!~n", [])
  end.
```

Figure 6.4.: Simple example of message passing in Erlang

Figure 6.4 shows a simple message passing procedure between two processes. The "spawn" command spawns a new process with the function passed as its argument, then returns its Process ID (PID). To send a message to a process, one must use the "!" command preceded by the PID of the receiver and followed by the message to be sent. The "self()" function returns the PID of the current process, so if an answer is required from the contacted process, one can pass its own PID along with the message it sends. The "receive" block is defined to handle received messages matching the supported messages.

Erlang has many more features that could not be treated in this paper such as list comprehension or hot code swapping, but more information can be easily found on the Internet [Erl 10].

6.2.2. Port Communication

In our implementation, we need Erlang to communicate with Ruby. To do so, one can open a communication ”Port” in Erlang that will maintain a link to an external driver, in our case a ruby script (See Figure 6.5). The Port will be kept open as long as a process is linked to it, so in our case we spawn a process that will act as a server to the Ruby port.

```
start_driver() ->
  Cmd = "rails runner ./lib/echo.rb",
  Port = open_port({spawn, Cmd}, [{packet, 4}, nouse_stdio,
            exit_status, binary]).
```

Figure 6.5.: Example of Port opening with Erlang

A Port is transparent in Erlang, so communicating through it is equivalent to talking to a local process. This is actually very powerful, because Ports are typically bridges between servers, so distant and local process communications are perfectly identical.

An Erlang port is also opened in the same fashion to perform requests to the SQLite database.

6.2.3. Mnesia

The standard database to be used with Erlang is Mnesia. With Mnesia, Erlang records represent key-value tuples where the value can be any data structure, from simple integer to Erlang lambda functions.

The query language to the database is Erlang itself and not a third party language such as SQL, which enable all the features of Erlang within the request procedure: pattern matching and list comprehension among others. To perform a query, one must perform a ”transaction”. A transaction will ensure that any read/write procedure to the database is performed properly. Transactions can easily be nested and can also be distributed on different process/servers transparently. A simple query example is shown in Figure 6.6.

Our implementation of the cache server uses Mnesia to store cached data. The database only has one table containing the pairs of key-values corresponding to ”url”, ”data” and ”timestamp”. The URL is used as an index for queries to the database, and the timestamp is here to check when the cached data should expire.

```
all_females() ->
  F = fun() ->
      Female = #employee{sex = female, name = '$1', _ = '_'},
      mnesia:select(employee, [{Female, [], ['$1']}])
      end,
  mnesia:transaction(F).
```

Figure 6.6.: Simple example of query to Mnesia Database returning the names of all female employees

More information about Mnesia can be found on the Internet [Mne 10].

6.3. SQLite

Mnesia is very powerful and fast, but it can only be accessible via Erlang. This makes it an inappropriate choice for a database that requires maintenance or arbitrary updates. On the contrary, SQLite is a very convenient database accessible in virtually any language, particularly Erlang in Ruby in our case.

By querying the SQLite database with Erlang while maintaining it with Ruby makes the application efficient and yet easy to maintain.

The SQLite database contains information on all the bus stops for each provider. Each entry in the database has the following attributes:

• stop_name	:	the name of the stop
• lat	:	the Latitude of the stop
• lng	:	the Longitude of the stop
• provider_name	:	the name of the stop's provider (Västtrafik or SL)
• stop_id	:	the id of the stop as defined by its provider
• timestamp	:	the timestamp of the stop creation
• id	:	a unique ID for indexing

The ”lat” and ”lng” attributes are the key entries allowing geographic search in the database, but finding nearby points from a database can be a tricky problem. In our case, we perform queries iteratively as shown in Figure 6.7 until a satisfying number of stops are returned.

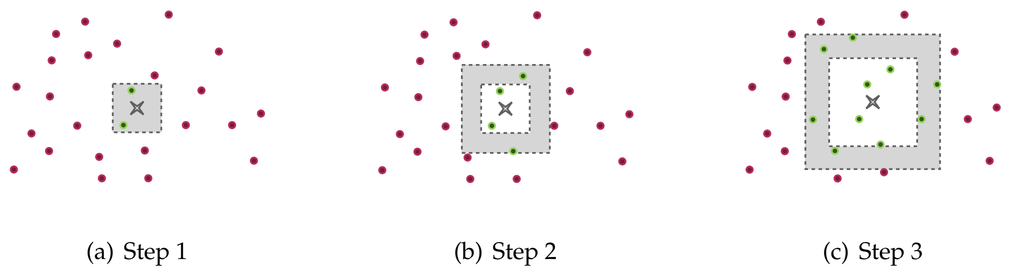

(a) Step 1 (b) Step 2 (c) Step 3

Figure 6.7.: Scheme of the SQL queries on the Ruby side

The first step consist of a SQL query is of the following form:

```
SELECT * FROM bus_stops WHERE lat >= lat1 AND
                              lat <= lat2 AND
                              lng >= lng1 AND
                              lng <= lng2
```

This will return all the bus stops within the area defined by the square defined by:

$$(lat, lng) \in [lat_1, lat_2] \times [lng_1, lng_2] \tag{6.1}$$

The retrieved stops are stored in a bus list. The following steps produce queries defined as follow:

```
SELECT * FROM bus_stops WHERE lat >= lat1 AND
                              lat <= lat2 AND
                              lng >= lng1 AND
                              lng <= lng2 AND NOT
                                   (lat > lat3 AND
                                    lat < lat4 AND
                                    lng > lng3 AND
                                    lng < lng4)
```

This will return all the bus stops within the area defined by the area defined by:

$$(lat, lng) \in ([lat_1, lat_2] \times [lng_1, lng_2]) \setminus (]lat_3, lat_4[\times]lng_3, lng_4[) \tag{6.2}$$

The retrieved stops are added to the stop list until the list is large enough to proceed. In our case, we stop the algorithm when the stop list has more than 10 entries.

Once we get a list of stops within an arbitrary large area, we need to sort those stops by distance in order to keep only the 10 closer ones. The distance d between the user's location and a stop is computed according to a simplified version of the great-circle distance formula given in Equation 6.3, where φ_s and λ_s are the Latitude and Longitude of the standpoint, φ_f and λ_f the ones of the forepoint, and R being the average radius of Earth ($\approx$ 6371km).

$$d = R \times \arccos\big(\sin\varphi_s \sin\varphi_f + \cos\varphi_s \cos\varphi_f \cos(\lambda_f - \lambda_s)\big) \tag{6.3}$$

6.4. Ruby

Ruby is a modern dynamic object-oriented programming language in which everything is an object, exactly as the programming language Smalltalk from which it is partly inspired. It has been designed to follow the least surprise principle during development: code should never behave in an unexpected way. It is a powerful language for writing scripts and developing applications in a very productive manner.

The complete implementation of the Ruby parts is given in Appendix B.

6.4.1. Setup

Ruby comes with a package manager called RubyGems that provides a standard distribution system for libraries. A Ruby library is called a Gem.

Among those Gems we find Rails, a framework that allows to deploy web applications using Ruby. When creating a Rails application, a web server is packed in, and that is what we have used in our project. We worked with Rails 3.0.0beta3, the current edge version of the framework, to deploy our Ruby application.

An other Gem was require to implement an interface with Erlang, so we used Erlectricity [Fle 09]. This library enable Ruby to send messages via the Erlang Binary Protocol, making communication between the two languages transparent. Figure 6.8 shows a simple implementation of communication.

One problem that we faced when communicating between Ruby and Erlang was when dealing with strings: Erlang can only handle Latin-1 encoded strings, whereas

```ruby
require 'rubygems'
require 'erlectricity'

receive do |f|
  f.when([:echo, String]) do |text|
    f.send!([:result, "You said: #{text}"])
    f.receive_loop
  end
end
```

(a) Ruby Side

```erlang
-module(echo).
-export([test/0]).

test() ->
  Cmd = "ruby echo.rb",
  Port = open_port({spawn, Cmd}, [{packet, 4}, nouse_stdio,
                                  exit_status, binary]),
  Payload = term_to_binary({echo, <<"hello world!">>}),
  port_command(Port, Payload),
  receive
    {Port, {data, Data}} ->
      {result, Text} = binary_to_term(Data),
      io:format("~p~n", [Text])
  end.
```

(b) Erlang Side

Figure 6.8.: Simple example of communication between Erlang and Ruby using Erlectricity

Ruby manipulates UTF-8 strings by default. So whenever a string is sent to be treated in Erlang, it has to be encoded in Latin-1 first, and when a string is received by Ruby, it has to be converted to UTF-8.

It has to be mentioned that Erlang and Yaws can handle strings in an arbitrary encoding under binary form, so when answering to the client the reply is actually UTF-8 encoded.

6.4.2. Database Management

Now that we know the required setup for our Ruby Implementation, let us have a look at what we want Ruby to do for us.

In our server application, Ruby is used to maintain the SQLite database. This database is populated by a background process that fetch data from the providers on monthly basis (it is not that often that new stops are build or that old one are destroyed). Updates are performed via requests to the webservices of the providers.

In order to perform those required maintenance processes, we make use of Rake, a Ruby-based software build tool that is packaged as a middle-ware within Ruby on Rails.

Rake combined with Rails offers powerful tools for database management, especially thanks to the ActiveRecord class. An Active Record object does not specify its attributes directly but infer them from the table definition with which they're linked. Adding, removing, and changing attributes and/or their type is done directly in the database.

The are different parts of the code required to set up our SQLite database are shown in Figures 6.9, 6.10 and 6.11.

```
# SQLite version 3.x
development:
  adapter: sqlite3
  database: db/development.sqlite3
  pool: 5
  timeout: 5000

test:
  adapter: sqlite3
  database: db/test.sqlite3
  pool: 5
  timeout: 5000

production:
  adapter: sqlite3
  database: db/production.sqlite3
  pool: 5
  timeout: 5000
```

Figure 6.9.: Configuration file that tells how to access the database (database.yml)

```
class CreateBusStops < ActiveRecord::Migration
  def self.up
    create_table :bus_stops do |t|
      t.string :name
      t.float :lat
      t.float :lng
      t.string :stop_id
      t.string :provider_name
      t.string :url

      t.timestamps
    end
  end

  def self.down
    drop_table :bus_stops
  end
end
```

Figure 6.10.: Migration file for the Bus Stops table (create_bus_stops.rb)

```
class BusStop < ActiveRecord::Base
  validates_presence_of :name, :lat, :lng, :stop_id,
                              :provider_name, :url
end
```

Figure 6.11.: Active Record of a Bus Stop (bus_stop.rb)

To perform the migration, one simply has to type the command "rake db:migrate", and the database will be instantiated if it doesn't exist, a Bus Stops table will be created with the proper attributes and an Active Record will be linked to it.

6.4.3. Parsing

In order to get forecast from Västtrafik, we must parse the XML that is returned by the provider. XML Parsing is made simple in Ruby thanks to its Gems, so there is no much need to expand this point. In our application, we used Nokogiri [Nok 10] to parse the data.

On the other hand, retrieving data from Storstockholms Lokaltrafik (SL) is less straightforward, and a large part of the parsing algorithm had to be implemented manually.

In order to parse the replies from SL's web services, we made intensive use of Regular Expressions. A Regular Expression (or Regexp) is used to define a pattern in a string. In Ruby, Regexps are Objects and can be declared as follows:

```
r = /[a-zA-ZöäåÖÄÅ]+/
```

This Regexp describes a succession of one or more arbitrary letters from the Swedish alphabet, be they capital or minuscule. It has to be noted that UTF-8 encoding is enabled within Regexp in Ruby.

Once we have a regular expression, we can scan a string to find its sub-strings matching the Regexp. Figure 6.12 gives an example of string parsing with the following pattern: ”Number␣Destination Name␣Number␣min”.

```
tram_regexp = /\d+\s[a-zA-ZöäåÖÄÅ\s\.]+\d+\smin/

text1 = "Next departure: Line 5 Lilla Torg. 25 min"
matches = text1.scan(tram_regexp) # => ["5 Lilla Torg. 25 min"]

text2 = "There is no next departure"
matches = text2.scan(tram_regexp) # => []
```

Figure 6.12.: Simple example of string parsing with Regular Expressions

Unfortunately SL does not have a public API, so we will not be able to describe in further details our implementation.

6.5. Performances

To test the implementation of our server, we used the Apache Flood tool. This tools allows dynamic and complex load tests via an XML configuration. In a first time, we tested our implementation by requesting nearby stops at different places in such a way as to avoid using the server's cached data. The test has been run during one day and one night, and the result is given in the Figure 6.13.

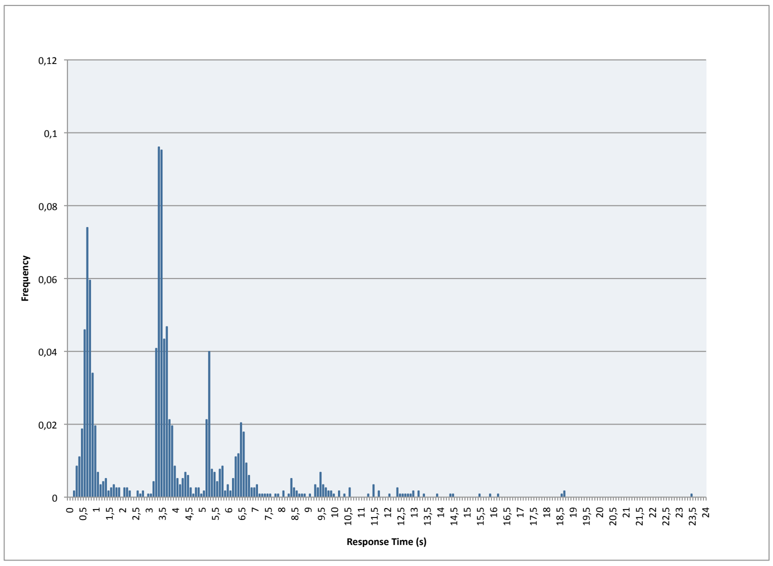


Figure 6.13.: Histogram showing the response time of the server without caching

We get a mean response time of 3.7 seconds with a standard deviation of 2.8 seconds, but when we look at the histogram we can clearly see a succession of Gaussian distributions, so those values are not really representative of the response model. If we look at the response time over time (Figure 6.14) we can see that there are plateaux in the curve.

After investigation, it appears that each plateau corresponds to the number of retries that are necessary on the server side to access Västtrafik's data.

With further considerations, we deduced that when a request is made from Västtrafik and the data is cached on Västtrafik side, the response time gets around 0.6 seconds. When a request is made and the data is not cached on Västtrafik side, the response time gets around 3.5 seconds. This correspond to the two lower plateaux and the first two Gaussian pikes of the histogram in Figure 6.13. Each higher plateau correspond to a number of retries that are necessary on the server side to access Västtrafik's data.

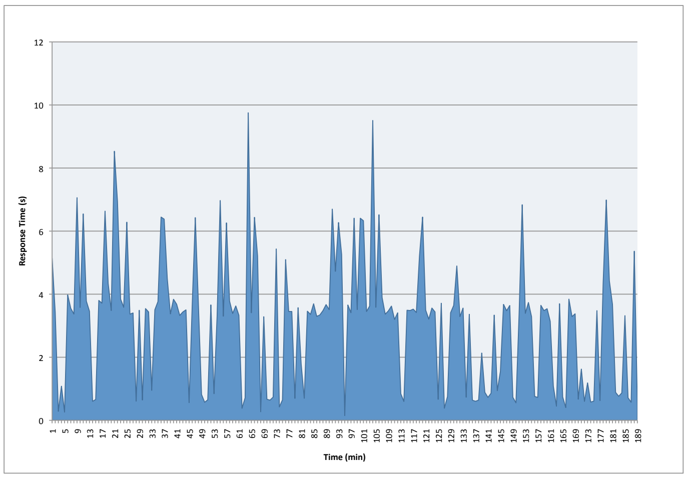


Figure 6.14.: Response time over time

When Västtrafik's servers are overloaded, time-outs and errors can occur thus requiring multiple retries.

After getting those results, we also analysed the response time when the data retrieved was beforehand cached by our server. To do so, we sent several requests successively at the same location within 30 seconds and discarded the first query.

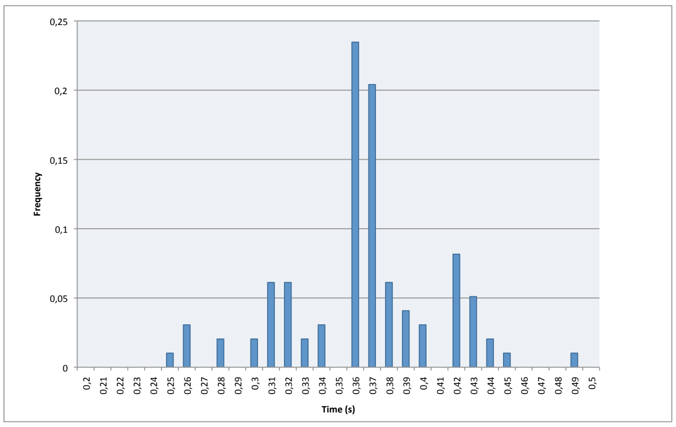


Figure 6.15.: Histogram showing the response time of our server with only cached data

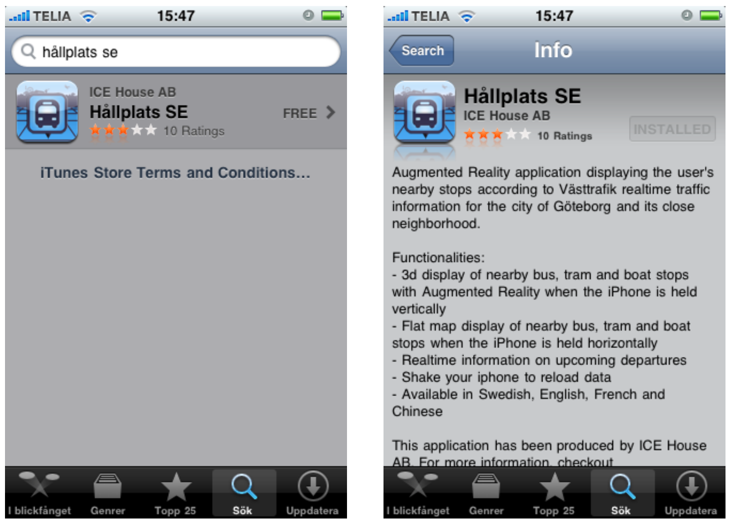
The result is shown in Figure 6.15. We get an average response time of 360ms and a standard deviation of 43ms. When looking at the histogram, we see that the distribution is close to a Gaussian distribution, so those values are representative.

7. Conclusion

7.1. Results

7.1.1. On the AppStore

The first release of the application in its revision number 0.9 was uploaded under the name "Hållplats Väst"on the AppStore on March the 8th and was ready for sale on March the 13th. The last version has been renamed "Hållplats SE" as it now covers Göteborg and Stockholm regions.



Augmented Reality application displaying the user's nearby stops according to Västtrafik realtime traffic information for the city of Göteborg and its close neighborhood.

Functionalities:
- 3d display of nearby bus, tram and boat stops with Augmented Reality when the iPhone is held vertically
- Flat map display of nearby bus, tram and boat stops when the iPhone is held horizontally
- Realtime information on upcoming departures
- Shake your iphone to reload data
- Available in Swedish, English, French and Chinese

This application has been produced by ICE House AB. For more information, checkout

Figure 7.1.: Screenshots of the Application on the Appstore

7.1.2. Statistics

After two months on the store, more than 800 units were downloaded, as shown in Figure 7.2.

But the number of downloads does not necessary reflects the number of real users of the application, so instead Figure 7.3 shows the numbers of unique users per day. The

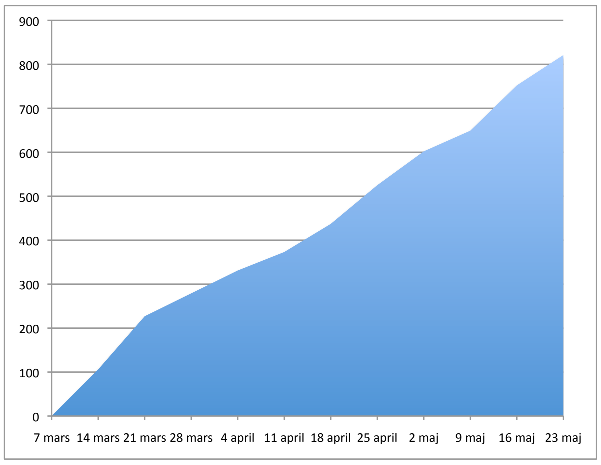


Figure 7.2.: Cumulative downloads of the Application on the AppStore

average number is around 40 unique users per day, which is rather satisfying.

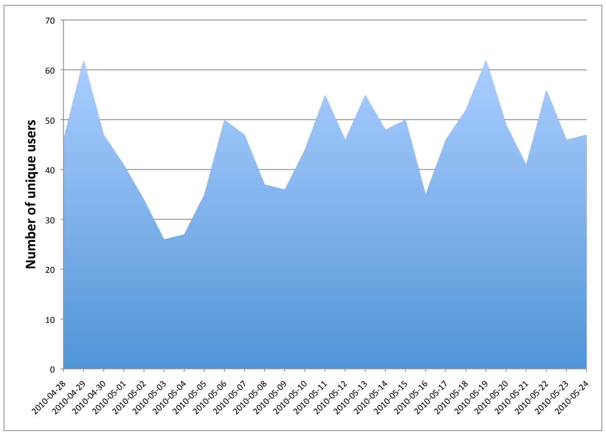


Figure 7.3.: Unique users per day

We can also know the user's location when he uses the application since the URL that is used to request his nearby POI contains his latitude and longitude coordinates. As we can see from Figure 7.4, users are well spread in the region of Göteborg and Stockholm, even though there is no advertisement made for the last one.

Funny enough to be mentioned, Figure 7.5 shows that some users also tried the application from Germany, South Korea, China, Singapore, and Qatar.

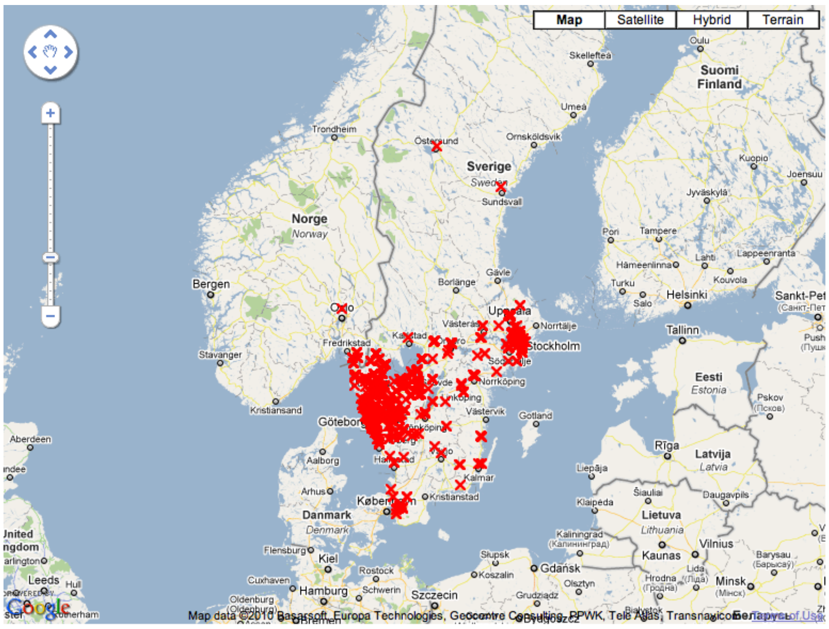


Figure 7.4.: Users' locations in Sweden

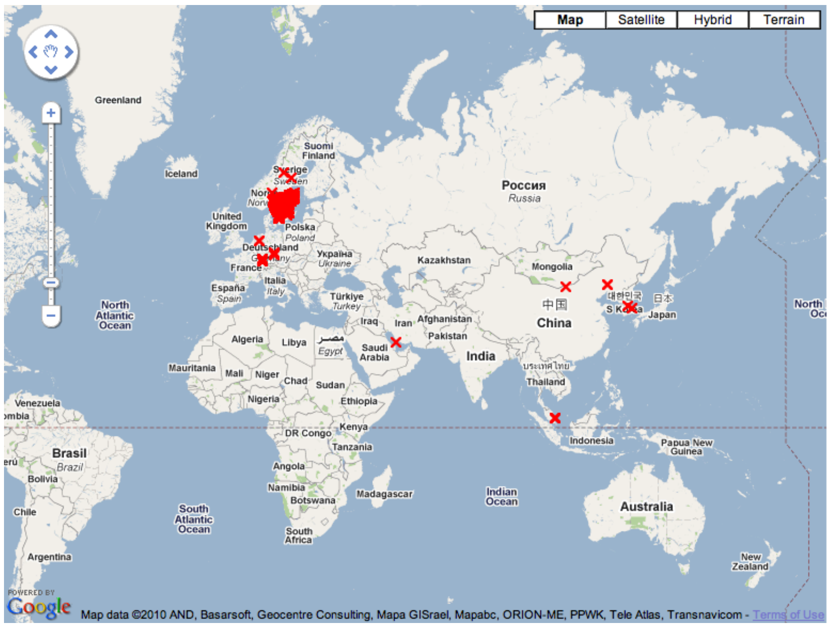


Figure 7.5.: Users' locations in... the world?

7.2. Further Considerations

7.2.1. on the Client

Our implementation of Augmented Reality only relies on GPS, compass and accelerometers inputs to estimate the position of the camera. But an additional input could be the camera itself and the images it captures.

Unfortunately, the iPhone SDK does not allow us to access the camera data so for now, but it the camera API will be made public with the release of the iPhone OS4.0 SDK.

7.2.2. on the Server

The server side has been designed in way as to be extensible easily. As such, new cities can be added to the database by creating a Ruby module that talks to the service provider. It would be interesting to extend the availability of the application to all the largest city of Sweden or even Europe, given that the city has a public transport provider with a web service API.

7.3. Summary

We have seen along this paper the implementation of an Augmented Reality for iPhone. The Client side was implemented using Objective-C and the framework necessary for iPhone development. Most of the problem that we faced were concerning trigonometry and projection, plus some restrictions imposed by the SDK that had to be handled.

The Server side was implemented using Erlang and Ruby. This part was really interesting and was also the most instructive part of the thesis. It dealt with concurrency, parsing, caching data and much more.

More work can be done and the application can be improved in many ways, especially the geographical coverage, but a stable release is already available from the Appstore and more that 800 units have been downloaded since its first release and 40 persons are using each day.

References

[App 10] *Apple IPhone Gallery*, Available at:
http://www.apple.com/iphone/gallery/, 7 may 2010.

[Cau 92] Caudell, T.P. and Mizell, D.W: *Augmented reality: an application of heads-up display technology to manual manufacturing processes*, Res. & Technol., Boeing Comput. Services, Seattle, January 1992.

[Erl 10] *Erlang Reference Manual*, Available at:
http://www.erlang.org/doc/, 7 may 2010.

[Fle 09] Fleckenstein, S. and Preston-Werner, T.: *Erlectricity*, Available at:
http://github.com/mojombo/erlectricity/, 7 may 2010.

[Hay 09] Hayes, G.: *16 top Augmented Reality business models*, Available at:
http://www.personalizemedia.com/16-top-augmented-reality-business-models/, 7 may 2010.

[Mil 94] Milgram, P., H. Takemura, et al.: *Augmented Reality: A Class of Displays on the Reality-Virtuality Continuum.* SPIE Proceedings: Telemanipulator and Telepresence Technologies . H. Das, SPIE. 2351 : 282-292, 1994.

[Mne 10] *Mnesia Reference Manual*, Available at:
http://www.erlang.org/doc/apps/mnesia/, 7 may 2010.

[Nok 10] *Nokogiri*, Available at:
http://nokogiri.org/, 7 may 2010.

[Sut 68] Sutherland, I. E.: *A head-mounted three dimensional display*, Proceedings of the December 9-11, 1968, fall joint computer conference, part I, San Francisco, California, December 1968.

[Tel 05] Telstar Logistics: *C-130J: Co-pilot's head-up display panel*, Available at:
http://www.flickr.com/photos/telstar/4136242/, 7 may 2010.

A. Erlang Code

File: hallplats.erl

```
-module(hallplats).
-export([start/0, restart_driver/0, get/1]).

start() ->
  inets:start(),
  cache_server:start(),
  spawn(fun() -> start_driver() end).

start_driver() ->
  io:format("New Port opened from ~p~n", [self()]),
  Cmd = "ruby ./lib/echo.rb",
  Port = open_port({spawn, Cmd},
                   [{packet, 4}, nouse_stdio, exit_status, binary]),
  register(ruby_port, Port),
  DriverPid = spawn(fun() -> ruby_driver(Port) end),
  port_connect(Port, DriverPid),
  register(ruby_driver, DriverPid),
  ok.

restart_driver() ->
  exit(whereis(ruby_driver), kill),
  port_close(whereis(ruby_port)),
  start().

ruby_driver(Port) ->
  sqlquery:start(),
  port_connect(Port, self()),
  link(Port),
  receive
    {transmit, Payload, FromPid} ->
      FromPid!{ok, driver_call(Port, Payload)},
      ruby_driver(Port);
    {Port,{exit_status,_}} ->
      start();
    {'EXIT', Port, Reason} ->
      io:format("~p ~n", [Reason]),
      start()
  end.

driver_call(Port, Payload) ->
  port_command(Port, Payload),
  receive
      {Port, {data, Data}} ->
        {result, _Text} = binary_to_term(Data)
  end.

get(Params) ->
  MyPid = self(),
  Lat = proplists:get_value(lat, Params),
  Lng = proplists:get_value(lng, Params),
  StopList = sqlquery:fetch_nearby_stops(Lat,Lng),
  spawn(fun() -> fetch_forecasts(StopList, MyPid) end),
  receive
    {ok, {forecast, ForecastList}} ->
      %io_lib:format("~p~n", [ForecastList])
      ToParse = term_to_binary({parse, ForecastList}),
      ruby_driver!{transmit, ToParse, MyPid},
      receive
        {ok, {result, FinalResult}} ->
          FinalResult
      end
  end.

fetch_forecasts(StopList, ToPid) ->
  MyPid = self(),
  [spawn(fun() -> fetch_forecast(Stop, MyPid) end) || Stop <- StopList],
  ForecastList = wait_for_forecasts([]),
  ToPid!{ok,{forecast, ForecastList}}.

wait_for_forecasts(ForecastList) ->
  receive
    {ok, Forecast} ->
```

```
      if
        length(ForecastList) < 9 ->
          wait_for_forecasts([Forecast | ForecastList]);
        true ->
          _Result = [Forecast | ForecastList]
      end
  after
    5000 ->
      _Result = ForecastList
  end.

fetch_forecast(Stop, ToPid) ->
  {_,_,_,_,_,_,_,_,URL} = Stop,
  Body = cache_server:get(URL),
  ToPid!{ok, {Stop, Body}}.
```

File: sqlquery.erl

```erlang
-module(sqlquery).
-export([start/0, fetch_nearby_stops/2, distance_geodesic/4]).

start() ->
  % sqlite:start_link(development),
  sqlite:start_link(development,
                    [{db, "../Rails/db/development.sqlite3"}]),
  sqlite:list_tables().

fetch_nearby_stops(Lat, Lng) ->
  Area = 0.002,
  Ratio = distance_geodesic(Lat, Lng, Lat, Lng + 0.1) /
                      distance_geodesic(Lat, Lng, Lat + 0.1, Lng),

  Query = io_lib:format( "SELECT * FROM bus_stops WHERE
                          lat > ~p AND lat < ~p AND
                          lng > ~p AND lng < ~p LIMIT 50",
                       [Lat - Ratio*Area,
                        Lat + Ratio*Area,
                        Lng - Area,
                        Lng + Area]),

  case sqlite:sql_exec(Query) of
    ok ->
      Result = find_10_closer([], Area * 2, Ratio, Lat, Lng);
    List ->
      if
        length(List) > 10 ->
          Result = List;
        true ->
          Result = find_10_closer(List, Area * 2, Ratio, Lat, Lng)
      end
  end,
  lists:sublist(sort_result(Result, Lat, Lng), 10).

sort_result(Result, Lat, Lng) ->
  lists:sort(
        fun(A, B) ->
```

```
          {_,_,Lat1,Lng1,_,_,_,_,_} = A,
          {_,_,Lat2,Lng2,_,_,_,_,_} = B,
          {A1, []} = string:to_float(Lat1),
          {B1, []} = string:to_float(Lng1),
          {A2, []} = string:to_float(Lat2),
          {B2, []} = string:to_float(Lng2),
          distance_geodesic(Lat, Lng, A1, B1) =<
                                         distance_geodesic(Lat, Lng, A2, B2)
        end, Result).

distance_geodesic(Lat1, Long1, Lat2, Long2) ->
    A1 = Lat1 * (math:pi() / 180),
    B1 = Long1 * (math:pi() / 180),
    A2 = Lat2 * (math:pi() / 180),
    B2 = Long2 * (math:pi() / 180),
    R = 6356.75,
    R * math:acos(math:cos(A1)*math:cos(B1)*math:cos(A2)*math:cos(B2) +
                  math:cos(A1)*math:sin(B1)*math:cos(A2)*math:sin(B2) +
                  math:sin(A1)*math:sin(A2)).

find_10_closer(OldList, Area, Ratio, Lat, Lng) ->
  Query = io_lib:format("SELECT * FROM bus_stops WHERE lat > ~p AND
                           lat < ~p AND lng > ~p AND lng < ~p AND NOT
                               (lat > ~p AND lat < ~p AND
                                lng > ~p AND lng < ~p) LIMIT 50",
                          [Lat - 2 * Ratio * Area,
                           Lat + 2 * Ratio * Area,
                           Lng - 2 * Area, Lng + 2 * Area,
                           Lat - Ratio * Area,
                           Lat + Ratio * Area,
                           Lng - Area,
                           Lng + Area]),

  Result = sqlite:sql_exec(Query),
  case Result of
    ok ->
      find_10_closer(OldList, Area * 2, Ratio, Lat, Lng);
    List ->
      NewList = erlang:append(List, OldList),
      if
      length(NewList) > 10 ->
        NewList;
      true ->
        find_10_closer(NewList, Area * 2, Ratio, Lat, Lng)
    end
end.
```

File: cache_server.erl

```
-module(cache_server).
-export([start/0, get/1, cleaner_daemon/0]).
-record(cached_url, {url, timestamp, data}).

start() ->
  mnesia:start(),
  mnesia:create_table(cached_url,
                      [{attributes, record_info(fields, cached_url)}]),
  spawn(fun() -> cleaner_daemon() end).

cleaner_daemon() ->
  F = fun() ->
      {Mega, Sec, _Micro} = erlang:now(),
      T = Mega * 1000000 + Sec - 60,
      TimeStamp = {T div 1000000, T rem 1000000, 0},
      MatchHead = #cached_url{url = '$1', timestamp = '$2', data = '_'},
      io:format("~p~n", [TimeStamp]),
      Guard = {'<', '$2', {TimeStamp}},
      Result = '$1',
      ToDelete = mnesia:select(cached_url,
                               [{MatchHead, [Guard], [Result]}]),
      [ mnesia:delete({cached_url, Url}) || Url<-ToDelete]
    end,
  receive
  after
    600000 ->
      mnesia:transaction(F),
      cleaner_daemon()
  end.

get(Url) ->
  case request_mnesia(Url) of
    {atomic,[[TimeStamp,DataTmp]]} ->
      case is_timestamp_valid(TimeStamp) of
        true ->
          Data = DataTmp;
        false ->
          Data = get_over_http(Url, 3),
          spawn(fun() -> write_mnesia(Url, Data) end)
      end;
    _ ->
      Data = get_over_http(Url, 3),
      spawn(fun() -> write_mnesia(Url, Data) end)
  end,
  Data.

get_over_http(Url, Retries) ->
  {ok, RequestId} = http:request(get, {Url, []}, [], [{sync, false}]),
  receive
    {http, {RequestId, Result}} ->
      case Result of
        {{"HTTP/1.1",200,"OK"}, Header, Body} ->
          UglyTest = [ 'TEST'|| {"content-length", "2442"} <- Header],
          case UglyTest of
            [] ->
              Body;
            _ ->
              if
                Retries > 1 ->
                  get_over_http(Url, Retries - 1)
              end
          end;
        _ ->
          if
            Retries > 1 ->
              get_over_http(Url, Retries - 1)
          end
      end
  after
    3000 ->
      if
        Retries > 1 ->
          get_over_http(Url, Retries - 1)
      end
  end.

is_timestamp_valid({MegaSec, Sec, _MicroSec}) ->
  TimeStamp = MegaSec*1000000+Sec,
  {NowMega, NowSec, _NowMicro} = erlang:now(),
  NowStamp = NowMega*1000000+NowSec,
  NowStamp - TimeStamp < 45.

request_mnesia(Url) ->
  F = fun() ->
      Cache = #cached_url{url = Url, timestamp = '$1', data = '$2'},
      mnesia:select(cached_url, [{Cache, [], [['$1', '$2']]}])
    end,
  mnesia:transaction(F).

write_mnesia(Url, Data) ->
  Cache   = #cached_url{  url= Url,
              timestamp = erlang:now(),
                         data = Data},
  F = fun() ->
      mnesia:write(Cache)
    end,
  mnesia:transaction(F).
```

B. Ruby Code

File: echo.rb

```ruby
# coding: utf-8
require 'rubygems'
require 'active_support/inflector'
require 'active_support/json'
require 'erlectricity'
require 'nokogiri'
require '../Rails/app/models/bus_stop_loader.rb'
require '../Rails/app/models/provider.rb'
require '../Rails/app/models/provider/vasttrafik.rb'
require '../Rails/app/models/provider/storstockholms_lokaltrafik.rb'

puts "loaded"

receive do |f|
  f.when([:parse, Array]) do |array|
    result = []
    i = 0
    while array[i] do
      element = {}
      stop_info = array[i][0]
      #element[:id] = stop_info[0].map(&:chr).join
      element[:name] = stop_info[1].map(&:chr).join
      element[:lat] = stop_info[2].map(&:chr).join
      element[:lng] = stop_info[3].map(&:chr).join
      #element[:stop_id] = stop_info[4].map(&:chr).join
      #element[:provider_name] = stop_info[5].map(&:chr).join
      #element[:created_at] = stop_info[6].map(&:chr).join
      #element[:updated_at] = stop_info[7].map(&:chr).join
      #element[:url] = stop_info[8].map(&:chr).join

      stop_id = stop_info[4].map(&:chr).join
      provider_name = stop_info[5].map(&:chr).join
```

```
        tobeparsed = array[i][1]
        loader =
               BusStopLoader.new("Provider::#{provider_name}".constantize)
        parsed = loader.parse_forecast(
                               tobeparsed.force_encoding("UTF-8"),
                               stop_id,
                               element[:name].force_encoding("UTF-8"))
        element[:forecast] = parsed
        result << element
        i += 1
      end

      f.send!([:result, result.to_json])

      #f.send!([:result, [].to_json])
      f.receive_loop
    end
  end
```

File: bus_stop.rb

```
class BusStop < ActiveRecord::Base
  validates_presence_of :name, :lat, :lng, :stop_id, :provider_name, :url

  after_initialize :extend_module

  private
  def extend_module
    self.extend "Provider::#{provider_name}".constantize
  end
end
```

File: bus_stop_loader.rb

```
class BusStopLoader
  def initialize(provider)
    extend provider
  end
end
```

File: provider.rb

```
module Provider
  module Base
    require 'timeout'
    require 'net/http'
    require 'uri'

    def delete_obsolete_stops(provider_name)
      BusStop.delete_all ["provider_name = ? AND updated_at < ?",
                                                      provider_name,
                                                       24.hours.ago]
    end

    def fetch_xml_from_url_string(given_url)
      retries = 2
      response = nil
      begin
        Timeout.timeout(2) do
          response = Net::HTTP.get(URI.parse(given_url))
        end
      rescue Timeout::Error
        if retries > 0
          retries -= 1
          retry
        else
          return nil
        end
      end

      xml_tree = Nokogiri::XML.parse(response, nil, 'UTF-8')
      return nil if xml_tree.css("string").size == 0
      encapsuled_xml = xml_tree.css("string").text
      xml_tree = Nokogiri::XML.parse(encapsuled_xml, nil, 'UTF-8')
    end

    def sort_lines(lines)
      lines.sort! do |a,b|
        if a[0].size == b[0].size ||
                 (a[0] =~ /[^\d]/ && b[0] =~ /[^\d]/)
          # Normal sort for lines of equal length or
          # if both are lines with names.
          a[0] <=> b[0]
        else
          # Sort by size if line names has different length.
          a[0].size <=> b[0].size
        end
      end
      return lines
    end
  end
end
```

File: vasttrafik.rb

```
module Provider::Vasttrafik
  include Provider::Base
  def update_stop_database
    provider_name = "Vasttrafik"
    use_identifier

    xml_tree = fetch_all_stops()
    return nil unless xml_tree

    rt90grid = SwedishGrid.new(:rt90)

    xml_tree.css("all_stops item").each do |element|
      stop_id = element.attribute("stop_id").text.to_s
      bus_stop =
           BusStop.find_or_initialize_by_stop_id_and_provider_name(
                            stop_id,
                            provider_name)

      rt90_x = Float(element.attribute("rt90_x").text.to_s)
      rt90_y = Float(element.attribute("rt90_y").text.to_s)
      latlng = rt90grid.grid_to_geodetic(rt90_x, rt90_y)

      bus_stop.provider_name = provider_name
      bus_stop.name           = element.css("stop_name").text.to_s
      bus_stop.lat            = latlng[0]
      bus_stop.lng            = latlng[1]
      bus_stop.url            = get_forecast_url(stop_id)
      bus_stop.touch # ?
      bus_stop.save
    end

    delete_obsolete_stops(provider_name)
  end

  def get_forecast_url(poi_id)
    use_identifier
    url = "http://www.vasttrafik.se/External_Services/" +
          "NextTrip.asmx/GetForecast?identifier=" +
        @identifier +
        "&stopId=" +
        poi_id
end

def parse_forecast(response, poi_id, poi_name)
  provider_name = "Vasttrafik"

  xml_tree = Nokogiri::XML.parse(response, nil, 'UTF-8')
  return [] if xml_tree.css("string").size == 0
  encapsuled_xml = xml_tree.css("string").text
  xml_tree = Nokogiri::XML.parse(encapsuled_xml, nil, 'UTF-8')

  # Extract data
  lines = []
  xml_tree.css("forecast item").each do |element|
    attributes = {}
    attributes[:line_number] =
              element.attribute("line_number").text.to_s

    attributes[:color] =
              element.attribute("line_number_foreground_color").
                    text.to_s

    attributes[:background_color] =
              element.attribute("line_number_background_color").
                    text.to_s

    attributes[:destination] =
              element.css("destination").text.to_s

    attributes[:next_trip] =
              element.attribute("next_trip").text.to_s

    alternate_text =
              element.attribute("next_trip_alternate_text").text.to_s

    attributes[:next_handicap] =
              (alternate_text == "Handikappanpassad".
                      force_encoding("UTF-8"))
```

```
      attributes[:next_low_floor] = (alternate_text == "Low Floor")

      attributes[:next_next_trip] =
                element.attribute("next_next_trip").text.to_s

      alternate_text =
                element.attribute("next_next_trip_alternate_text").
                        text.to_s

      attributes[:next_next_handicap] =
                (alternate_text == "Handikappanpassad")

      attributes[:next_next_low_floor] =
                (alternate_text == "Low Floor")

      lines << [attributes[:line_number], attributes]
    end

    return sort_lines(lines)
  end

  private
  def fetch_all_stops
    get_poi_list_url = "http://www.vasttrafik.se/External_Services/" +
                       "TravelPlanner.asmx/GetAllStops?identifier=" +
                       @identifier

    xml_tree = fetch_xml_from_url_string(get_poi_list_url)
  end

  def fetch_forecast(poi_id)
    get_poi_forecast_url = "http://www.vasttrafik.se/" +
                           "External_Services/NextTrip.asmx/" +
                           "GetForecast?identifier=" +
                           @identifier + "&stopId=" + poi_id

    xml_tree = fetch_xml_from_url_string(get_poi_forecast_url)
  end
end
```

File: bus_stops.rake

```
namespace :bus_stop do
  desc 'Load bus stops from all providers'
  task :load_all => :environment do
    [Provider::Vasttrafik, Provider::StorstockholmsLokaltrafik].
                                                    each do |provider|

      loader = BusStopLoader.new(provider)
      loader.update_stop_database
    end
  end
end
```

File: database.yml

```
# SQLite version 3.x
development:
  adapter: sqlite3
  database: db/development.sqlite3
  pool: 5
  timeout: 5000

test:
  adapter: sqlite3
  database: db/test.sqlite3
  pool: 5
  timeout: 5000

production:
  adapter: sqlite3
  database: db/production.sqlite3
  pool: 5
  timeout: 5000
```

File: create_bus_stops.rb

```
class CreateBusStops < ActiveRecord::Migration
  def self.up
    create_table :bus_stops do |t|
      t.string :name
      t.float :lat
      t.float :lng
      t.string :stop_id
      t.string :provider_name
      t.string :url

      t.timestamps
    end
  end

  def self.down
    drop_table :bus_stops
  end
end
```