More than downloading

Visualization of data produced by sensors in a home environment

IVAN PEDERSEN
and
ALFRED ANDERSSON

KTH Information and Communication Technology

Degree project in
Communication Systems
First level, 15.0 HEC
Stockholm, Sweden

More than downloading

Visualization of data produced by sensors in a home environment

Ivan Pedersen

and

Alfred Andersson

14 June 2012

Bachelor's thesis

Mentor and examiner: Prof. Gerald Q. Maguire Jr.

Communication Systems
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden

Abstract

A home automation system usually contains a set of tools that users use to control devices in their homes, often remotely. These devices often include but are not limited to light switches, thermostats, thermometers, window blinds, and climate controls. The potential for these kinds of systems is huge because of the sheer number of devices that could be controlled and managed with minimal and inexpensive extra hardware. Many of the appliances in a normal home could benefit from being connected to a system that allows the owner to manage and control the devices in their home. Thus the number of potential devices is orders of magnitude larger than the number of homes connected to the system. There are several systems on the market that provide systems to monitor and control a home environment, however these systems only support specific in system devices. This uncovers a problem where a homeowner only has the opportunity to use specific products that fit into these systems. By introducing an open platform for the public that are not bound to any system we can allow more devices to be integrated in the home and contribute to further development of smarter homes.

The goal with this project was to provide a scalable open platform with the possibility of asynchronous updating. This has been done by implementing multiple logical parts to both provide a web interface for the user and to allow us to handle communication and storage of data. All these parts are linked together to form a system of servers that handles all background operations. This thesis discusses and presents implementations of all of these servers, how they are implemented, communicate with each other, provide secure connections and how they can scale with increasing usage. In this process we also discuss and present techniques that were used, how to use them and their benefits, to help us reach our goal.

Sammanfattning

”Home automation” syftar till ett system som låter användaren kontrollera och styra olika apparater i hemmet, ofta sker detta utifrån. Dessa apparater inkluderar, men är inte begränsade till ljusbrytare, termostater, termometrar, persienner eller klimatanläggningar. Potentialen för ett sådant system är enormt då antalet apparater som skulle kunna övervakas med endast minimal och billig extra hårdvara är stort. Många av dessa apparater kan dra nytta av att vara ansluten till ett system som gör det möjligt för ägaren att hantera och styra enheter i deras hem. Antalet apparater är därför mångdubbelt fler än antalet hem som är kopplade till systemet.

Det finns flera system på marknaden som ger användaren ett sätt att övervaka och styra en hemmiljö, men dessa system är ofta låsta och stödjer bara specifika enheter. Genom att införa en öppen plattform för allmänheten som inte är bunden till något system, kan vi tillåta att fler enheter kan integreras i hemmet och bidra till ytterligare utveckling av smartare hem.

Målet med detta projekt var att skapa en skalbar öppen plattform med möjlighet till asynkron uppdatering. Detta har gjorts genom att implementera flera logiska delar för att förse användaren med ett webbgränssnitt och för att tillåta oss hantera kommunikation och lagring av data. Alla dessa delar är sammanlänkade för att bilda ett system av servrar som hanterar alla bakgrundsprocesser. Denna avhandling diskuterar och presenterar implementeringar av alla dessa servrar, hur de genomförs, kommunicera med varandra, ger säkra anslutningar och hur de kan skala med ökad användning. I denna process diskuterar och presenterar vi de tekniker som använts, hur man använder dem och deras fördelar.

Acknowledgements

We would like to thank Prof. Gerald Q. Maguire Jr. for his help and assistance during this thesis project. We would also like to thank Abdel Ahmid for collaborating with us during the feasibility study.

Table of Contents

List of Figures

List of Tables

Glossary

3DES	Triple Data Encryption Standard
AJAX	Asynchronous JavaScript and XML
AES	Advanced Encryption Standard
API	Application Programming Interface
CBC	Cipher Block Chaining
CSS	Cascading Style Sheets
CoS	Department of Communication Systems, KTH, Kista, Sweden
Gateway	A device that receives data from one system and transmit it onto another system
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HW	Hardware
IP	Internet Protocol
ISP	Internet Service Provider
IV	Initialization Vector
JS	JavaScript, a popular scripting language that is widely used on the web
JSON	JavaScript Object Notation
MAC	Message Authentication Code
MCS	Modulation and Coding Scheme
Mbps	Mega Bit Per Second
MHz	Megahertz
MVC	Model View Controller
OS	Operating System
PHP	PHP: Hypertext Preprocessor
PoE	Power over Ethernet, method to supply power to devices via the Ethernet interface port
SSH	Secure SHell
SSL	Secure Socket Layer
SVG	Scalable Vector Graphics
SW	Software
Rails	Ruby on Rails
RGB	RGB colour model, stands for Red, Green and Blue
RTT	Round Trip Time
TCP	Transmission Control Protocol
VPN	Virtual Private Network
UDP	User Datagram Protocol
URL	Uniform Resource Locator
USB	Universal Serial Bus, communication protocol used for communication and power supply between a computer and a device.
XML	eXtensible Markup Language
XOR	eXclusive OR

1 Introduction

This chapter introduces the reader to the purpose of this thesis project and describes the problem to be solved. This chapter also explicitly delimits the scope of this thesis project and outlines the structure of the rest of the document.

1.1 Background

Prof. Gerald Q. Maguire Jr. proposed several Bachelor's thesis projects titled "More than downloading" on his website [1]. The idea was that the Internet could and should be used more to upload data and for users to be able to share certain information. He describes the thesis projects as follows:

"Today many networks are optimized for downloading (i.e., a transfer through the networking infrastructure to a host). Examples include web browsing, audio and video streaming from traditional media providers, etc. One of the coming trends is increasing generation of content (even if it is only a sensor saying "the temperature in the basement has fallen below 8°C"). This will result in increasing amounts of traffic in the uplink direction." [1]

This led us to the idea to develop a system that would present various types of sensor data via a user-friendly interface, accessible from anywhere using a web browser and an Internet connection.

1.2 Problem description

We want to be able to receive data from various sensors and send this data via a gateway to a remote server. This server can be located in the cloud. The user will then be able to view the data via a web interface. Our goal is facilitate users uploading some of their data to other services, such as weight data that could be uploaded to sites such as www.runkeeper.com. Furthermore, we want to create a secure and easy to use website that enables a user to control and/or view the data that was collected within their home. Much of the communication between the various sensors and a local gateway in the home needs to be done over a radio operating in one of the ISM-bands. Communications from the gateway to the Internet will use appropriate Internet protocols; the specific protocols to be used will be examined as part of this thesis project. An overview of the network topology that we consider in this thesis project is shown in Figure 1-1.

The purpose of the project is to create a manageable and easy to use home automation system with low power and low cost components, and to present information collected via these sensors to the user via a web browser. This will allow the user to access their data and to monitor their home from anywhere that they have Internet connectivity.

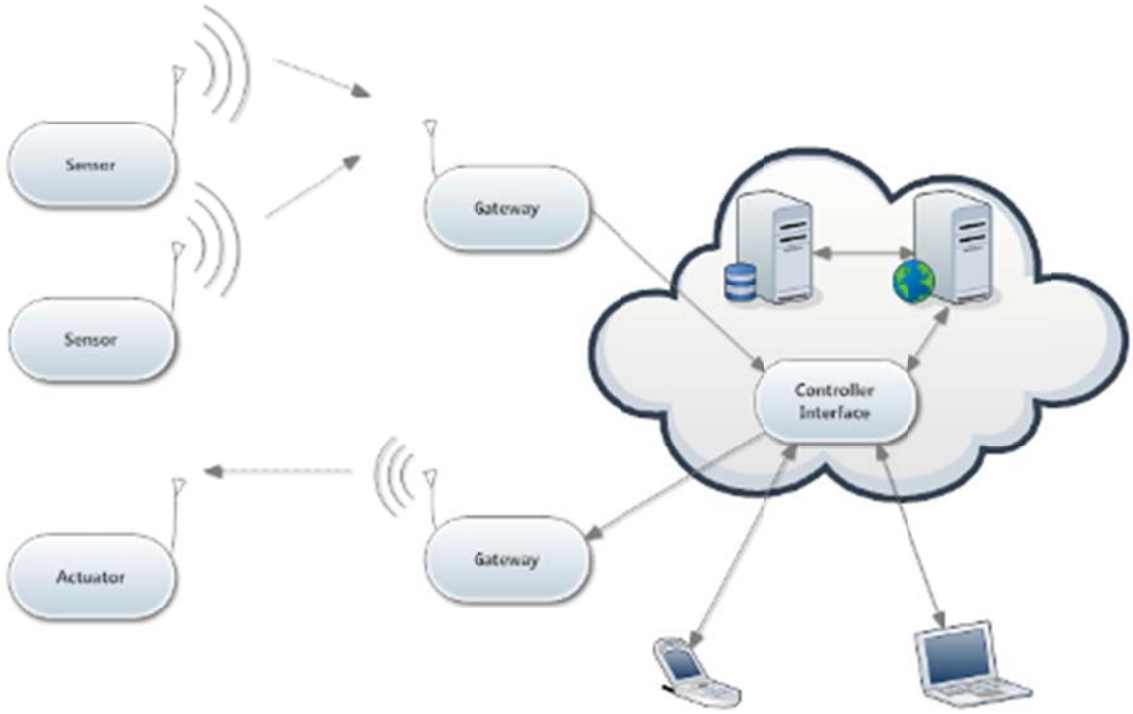

Figure 1-1 System overview

1.3 Sister thesis projects

Simultaneously with this project, Abdel Ahmid [2] is working on another related thesis. His thesis project focuses on the sensor side of the system by collecting and sending data from a scale that has been augmented with wireless communication. Both of our thesis projects are also connected to yet another thesis that Francisco Javier Sánchez currently are working on, this master's thesis project focuses on the development of a gateway that is able to pick up signals from existing sensors in range of their gateway and then relaying this data via an IP network.

These projects are logically separated by this gateway. Abdel will develop a new sensor that is capable of sending data to the gateway and our project will focus on taking data from the gateway, placing this data into a server (which could be in the cloud) and making this data available to the home user anywhere, as long as they are attached to the Internet via a web browser.

1.4 Limitations of this thesis project

This thesis project focuses on sending data to a server and presenting the collected data via a user-friendly interface, thus limiting the scope from the original problem definition to focus on only these aspects. We will design a system that allows others to send information to our system and this data will also be able to be displayed via our website.

Furthermore, our communication with the gateway will be assumed to be simplex, with data only travelling from the sensors to our servers. This limits the security that we will be able to implement because only a pre-configured shared key will be used to protect the information as this data is transferred from the gateway to our servers. This also limits the use of any actuators that could act based on the data that we receive, we leave the extension to actuators and full duplex communications to a future thesis project. Our system will not relay any information to any third parties.

1.5 Structure of the report

Following this introduction, Chapter 2 gives an introduction to the project. In this chapter we will present an introduction to home monitoring, related work, and development techniques. Chapter 3 describes the implementation of our web interface, development of a server that receives data from the sensors (logically via the gateway), and how the security, database, and the push notification system is implemented within the system. Chapter 4 contains the conclusions that we have drawn from this project and suggest some future work.

2 Project introduction

The purpose of this thesis project is to create a platform for visualization of data produced by sensors in a home environment. Our goal is to create a platform to support multiple types of sensor data and allow anyone with data from a wide variety of sensors to use this platform. The project is divided into two major parts, one concerns visualization and the part that handle incoming data, which will be stored in a database.

The visualization part of this project concerns how to present the data that have been collect to a user via a web-interface. Our goal is to allow the user to monitor their home environment via their choice of web browser. Some of this flexible compatibility comes from using web standards that World Wide Web Consortium (W3C) has approved [3].

As some of this data can be quite sensitive it is important that we authenticate the user and ensure that only successfully authenticated users can view information about their own home environment. In order for the user to visualize this data we will present it using graphs and tables, and perform live updates of these graphs and tables.

The part of the project that handles the incoming data requires us to develop a server that can receive messages from different sources and successfully forward these messages to be stored in the database and pushed to the website. Note that we support this push operation in order to have live updates of the data that the user is currently viewing. This part includes security for the data while it is in transit by encryption and use of a hash to ensure that its integrity is maintained. In addition we have to address the issue of concurrency in order to allow scalability of our proposed solution. This means that we have to consider the details of processes of database insertion and querying and pushing of content to allow live updates.

2.1 Introduction to smart home monitoring

The concept of smart homes is based upon devices handling and providing information about the home environment. A smart home provides the homeowner with new ways of interacting with the connected devices in their home environment. The use of home monitoring to provide security for a home is a well-known concept, but the concept of home automation has been extended to include more than just traditional aspects of home security. These other aspects include monitoring of water or power consumption and healthcare. The monitoring of a home can be done internally, thus allowing only authenticated users in the home environment to observe the data that the home automation system collects. Another way of monitoring the home is to provide tools that allow the user to monitor their home externally. By allowing a user to monitor their home without being physically in the home environment, a new and very interesting range of use cases can be explored.

Monitoring of a home involves putting together a puzzle consisting of a large and complex system of sensors, actuators, gateways, etc. As new technology provides the home with new smart devices and broadband Internet connectivity becoming more and more common in homes the possibility of combining these devices become increasingly interesting. Today TVs, game consoles, computers, tablets, media hubs, amplifiers, etc. are already being integrated into the home environment's local network(s). This has at least two effects. The first effect is that controlling devices in the home is becoming more convenient than ever before. Remote controls are swapped with tablets, computers control the volume of the amplifier, and there are many different ways of controlling and communicating between these devices in a smart way. Expanding this network of things by adding new devices could provide the user with even more functionality. It is this expanded functionality that is the second effect of these devices being connected to the home network(s). For example, a user no longer needs to open the fridge to know what is in it. Instead the user might go to a webpage that contains the information about the contents of the fridge. The fridge would with Internet connectivity be able to upload this information by itself. The same might be done for kitchen cabinets. The result is that there is no more guessing while you are out shopping whether you have all the ingredients at home for the meal that you want to make.

There are already many systems that handle monitoring and control of a home environment such as OpenHome [4] and Z-Wave [5]. These systems usually come with their own solutions, protocols, and functions. This means that there are several different protocols, devices and controllers all speaking different languages. This limits the opportunity to truly customize the home environment to fit the user's needs. However, these systems follow a similar structure where multiple sensors and/or actuators wirelessly transmit/receive data to/from a gateway. This data can be collected from these sensors and potentially be handled in some way other than the closed solutions that currently exist.

As noted above we can imagine home automation that exists only within the home, where there is only local storage of data and the user can only monitor and control their home environment from within the home itself. Alternatively we can imagine a system that utilizes external storage of data (potentially storing this data in the cloud), but still only allowing monitoring and control from within the home. The third alternative is to utilize external storage of data and allowing the user to monitor and control their home environment from anywhere via a normal web browser.

2.2 Gateway

Today many existing sensors broadcast their data in the licence free 868 MHz band. In order to intercept these transmission, decode and extract this data, and forward it to the Internet a gateway is needed. The job of this gateway is simply to listen to one or more frequencies, collect data, and relay the data. This gateway could be connected to another wireless link compatible with a 2.4 or 5 GHz Wi-Fi equipped router, a USB connection, or a wired Ethernet connection. By implementing an IP stack directly in the gateway we can eliminate the need for an always-on computer in the home (if we are storing the collected data outside of the home – as only the gateway and router need to be powered on).

Francisco Javier Sánchez and Albert López have during the winter/spring 2012 developed a gateway that uses the 868 MHz band to "sniff" sensors' data and send them to a computer via an Ethernet interface. Their gateway uses a Texas Instruments MSP430 microcontroller and a CC1101 wireless transceiver to communicate with different types of sensors. The device requires very low power and is able to operate from power over Ethernet (PoE). The gateway contains only low cost components. This device is described in detail in their thesis [6].

2.3 Introduction to web development

There are a lot more aspects that must be considered in web development then just a couple of years ago. Today web sites are expected to be more dynamic than ever before and this requires some new technologies. The techniques that allow for the live feeds and data streams that we see in many websites such as Facebook or Twitter simply did not exist a few years ago.

Static websites does not require a lot of the advanced functions that a website today would not work without. Today websites needs to be updated constantly and the old approach of hard coding data into a website makes this difficult. Today websites are generated by a dynamic front end that constantly is receiving updates from some type of backend logic. The backend handles connections to databases and only lets users access the information that they should be able to access based upon some form of authentication and authorization mechanism. This means that a website does not look the same for every user, but instead its appearance depends on the user's credentials.

In this chapter we describe how the structure in Ruby on Rails works and what techniques can be used to create a dynamic and customizable website.

2.3.1 Ruby on Rails

Ruby on Rails or simply "Rails" is an open source web framework developed by the Danish programmer David Heinemeier Hansson. Rails is as its name suggests written in the Ruby language and is designed to make web applications easier and faster to develop. [7]

2.3.1.1 MVC

In the heart of Ruby on Rails development is a Model-View-Controller (MVC) architecture. This architecture allows the program to be separated into three major parts: models, views, and controllers. The model represents the data of the application and how to access/manage this data. The views represent the user interface and in the case of a web application the views render all the HTML content that is displayed on a web page. The controller couples the models and views. Further details of each of these parts are given below.

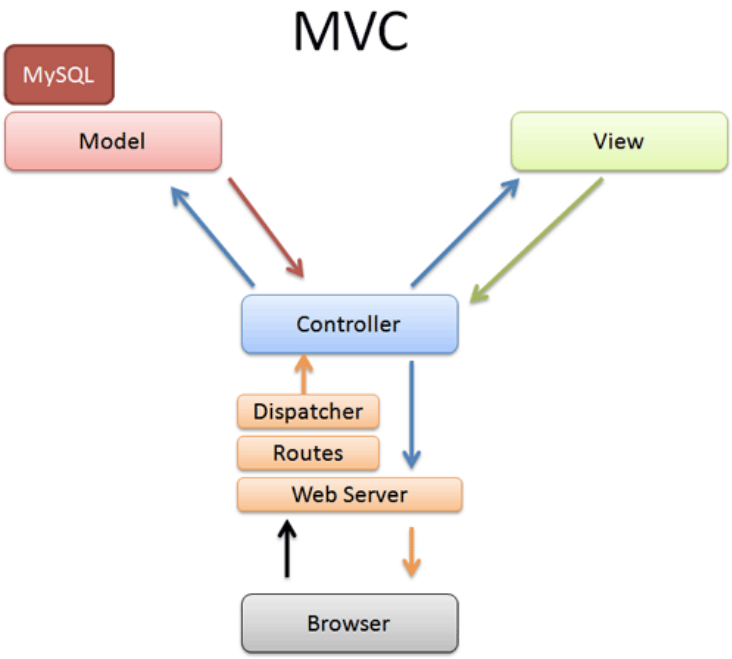

Figure 2-1 Overview of the Model-View-Controller (MVC)

2.3.1.2 Model

In Rails the model represents the data that is present in a database and the rules that govern how to manipulate this information. It is in these models that the associations between the different tables in the database are specified. Rails handles all this and all that we have to do is use keywords such as "belongs_to", "has_many", "has_one" or "has_and_belongs_to_many". A user model could for example specify "has_one :father" and "has_many :children" and that would express the relationship that the "User has one father and has many children." The father of the user will then be accessible in the model as a user object that has a father attribute. Furthermore the information is accessible from the view by using embedded ruby statements.

2.3.1.3 View

The view is what the users see and can interact with. The content shown on the webpage and how it is shown is provided by the view. Creating a view is done by using common web-programming languages such as HTML, CSS, XML, JavaScript, and JSON. However, the development is a bit different than normal web development using these languages. One of the differences is that Rails can combine several of the common web languages and the ruby programming language. The code is written in an .erb (embedded Ruby) file. Rails use the following structure to enable integration of ruby code into the .erb file:

```
<% Ruby code -- inline with output %>
<%= Ruby expression -- replace with result %>
```

The view is just the end point of the entire system and does not operate in the back end as do the model and controller. The view simply uses what the controller says it should use and renders it for the user in a structured way.

2.3.1.4 Controller

The controller can be seen as a middleman between the model and view. Its purpose is to relay information received from the model to the view. This is done by parsing requests from the user and fetching whatever is needed from the model and sending it to the view that displays the new content for the user. The controller also handles work related to data submission, cookies, and sessions. The controller follows a special structure that contains multiple functions, where each function first interacts with the model and then parses the result and calls a view. Common operations in a control function are shown below:

```
@variable = ModelX.find(params[:id])
format.json {renderjson: @variable }
```

2.3.1.5 Scaffolding

Scaffolding is a very useful tool in Rails that allow you to create a good structure upon which further development could be based. It is unlikely that the scaffolding provides exactly the right thing for the application, but it provides models, controllers, and views that you can modify to fit the application's specific needs. This means that you can have a very basic structure up and running together with a database in just a few commands. The following command will provide a sensor model, matching controller, and views for the sensor. The two arguments “name” and “sensor_type” correspond to columns in the database. Rails also creates an auto increment “id” column automatically.

```
$ rails generate scaffold Sensor name:string sensor_type:string
```

Rails follow a linguistic naming convention, this means that the controller represents a sensor (singular) but the database will contain a list of sensors (plural).

2.3.1.6 Managing the database

Rails allows you to create an empty database with the command:

```
$ rake db:create
```

This command will create an empty database based on pre existing models. To be able to update the database, add new tables, and delete old ones a so-called migration is needed. This migration can be automatically generated in a similar fashion as the scaffold (described in section 2.3.1.5). A migration file is also generated when scaffolding that correspond to the model created. A migration can be viewed as both an action, updating the database, and as a file, the migration file. The information about how to update the database is stored in the migration file and can of course be created manually without any automatic generation methods.

To be able to update the database a command written in the ruby console needs be executed. When executing this command, all new migration files in the project are processed. The command to execute the update/migration is:

```
$ rake db:migrate
```

2.3.1.7 Gems

Gems in Ruby are packaged Ruby applications or libraries. Gems are distributed via the “gem” command. The web Ruby on Rails framework is actually a Ruby gem called “Rails” and is installed with a simple command in the ruby console. Other gems can be added as a line of text to a specific “Gemfile” inside Rails and then instantiated by running the “bundle install” command. The “bundle install” command also handles dependencies that a gem might have. Gems can also be installed by

using commands directly on the command line, the same way as installing the Rails gem. Installing gems directly via the command line results in installation of a local gem and the functionality provided by this gem will only work for environments with this gem installed. The syntax for installing new gems is:

```
$ gem install ‘gemToInstall’ –options
```

The preferred way of installing gems is to modify the “Gemfile” to include the desired gem and using the bundle command to install it and its dependencies. This approach allows developers to easily distribute their application to other developers and co-workers.

2.3.1.7.1 Devise

Login-authentication is handled by a ruby-gem called Devise [8]. Devise provides the base, which other gems such as Omniauth [9] then will incorporate. Devise is a Rails engine, which means that it provides its own controllers, models, and views. This will provide the basic structure as well as very useful helper methods such as “current_user”, which return the user that is currently logged in. Of course the initial structure that Devise provides can be modified to suit our needs.

2.3.1.7.2 MySQL

To be able to change the database type from the original sqlite3 to MySQL the MySQL gem is needed. This gem provides an API module for ruby, which has the same functionality as the MySQL C API. The gem has some important dependencies that have to be considered when using it on a Microsoft Windows operating system. The libMySQL.dll used in a Windows operating system needs to have the same version as the one used in the gem.

2.3.1.7.3 Faye

Handling and listening to external events can be somewhat tricky in Rails. By using the Faye gem event handling gets a bit easier. The Faye gem provides functions that can be used to subscribe to specific channels or push to any channel used by a Faye server.

3 Implementation

Our project consists of four main parts. These parts together create a system where we can receive data through UDP datagrams, store data into a database, push current data to a live feed, and visually present the data in a web browser. Because we also wanted to test our system we developed a program that sends UDP messages to our server, just as a sensor would send information to our system. We call this program "SendUDP" and it is described in more detail in section 3.6.

The parts and the relations between them are presented as an overview in Figure 3-1.

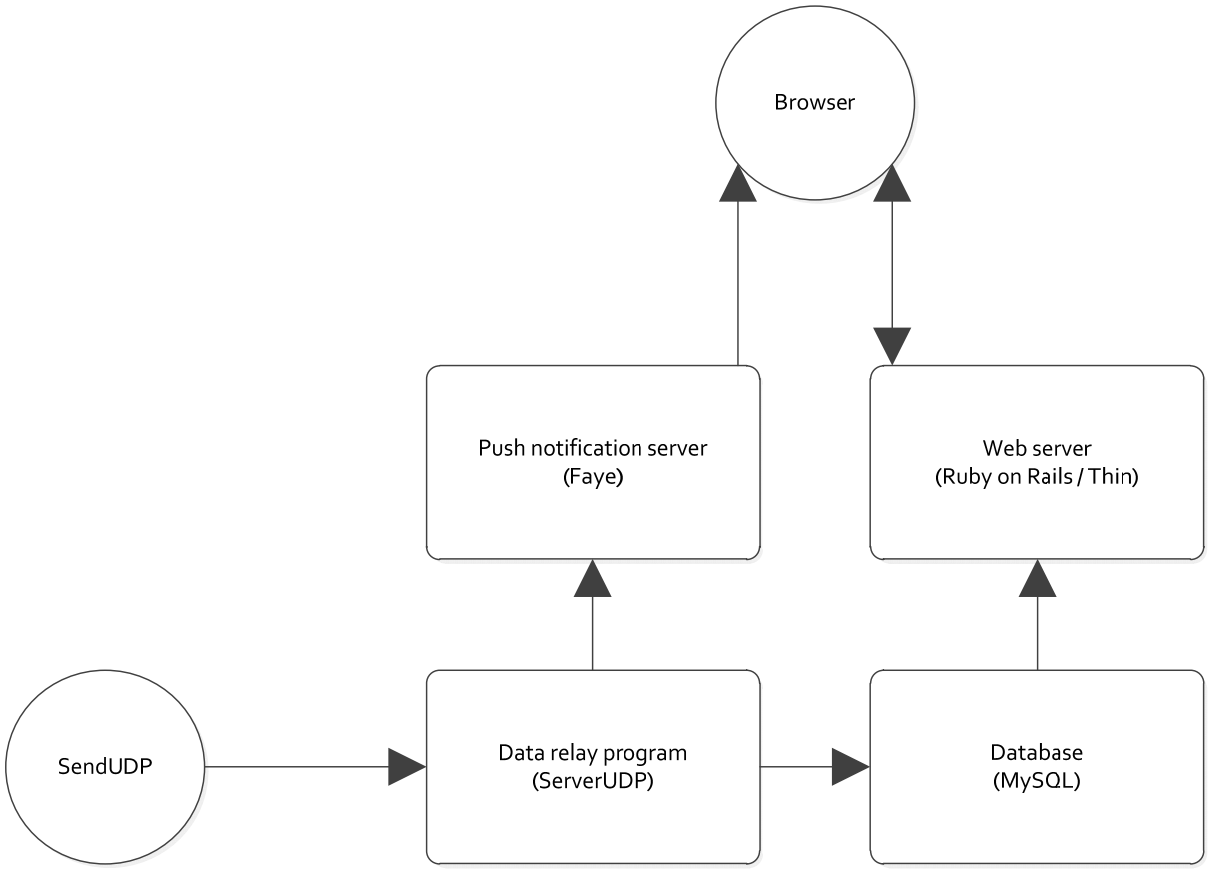

Figure 3-1 Main system structure

3.1 Creating a home visualization interface

To be able to rapidly develop this visualization tool we choose the Ruby on Rails framework. Ruby on Rails is "...optimized for programmer happiness and sustainable productivity." [7], we have also had a little earlier experience working with Rails and wanted to improve our knowledge of this framework. This tool will be the final part of the whole system, and can only operate when previous parts of the system are complete and operating.

3.1.1 Login/authentication

The Devise gem (described in section 2.3.1.7.1) provides the user model along with the logic to protect the user's password. To achieve this Devise uses the gem bcrypt-ruby [10]. This means that the password is never stored in a humanly readable format and can thus never be compromised, even in the event of a security breach. Bcrypt is a hashing algorithm designed by Niels Provos and David Mazières of the OpenBSD Project [11].

Because Devise is a Rails engine there is a built in standard design of Devise. This was a great help to us to quickly start using authentication. However, to be able to fit our needs; we had to modify the design. To modify the standard design of devise we had to expose the files that provide the design. This was done by executing the following command in the console:

```
$rails generate devise_views
```

The generated views gave us access to all of the information that we needed in order to modify the whole layout of Devise. The modifications made to the layout where minimal, but it gave us an understanding of how we could modify Devise and integrate new functions into a layout.

Devise handles most of the essential parts of the authentication process, but not all parts. One vital part is matching the content displayed to the correct user. If one user creates a new gateway on the webpage, this gateway should only be available to that user. To handle this we had to modify the controllers responsible for listing the gateway entries that will be shown. Devise provides a helper function that returns the current user that is logged in. This helper function was used to list only the entries that the current user had created.

3.1.2 Visualization

An important part of this project is to be able to show the data to the user in a user-friendly way. The data that we initially consider supporting included temperature and weight; therefore decided to use gauges and graphs. There are a number of different APIs available that could be used to do this and most of them use JavaScript to load the data into the graphs. In this chapter we describe the different APIs that we use in our system.

3.1.2.1 Google chart

Google provides a free chart API [12] that is easy to use and show a wide variety of different types of charts, maps, tables, and gauges. The charts are rendered with HTML/SVG technology and this provides great cross-browser compatibility, including browsers in Android and iOS without any need for any extra plug-ins.

The use of gauges was a natural step to take when working with temperature data. The API also provides us with some options to customize the gauge. For example, different colour ranges can be displayed on the gauge. The range of the gauge is also customizable, as well as animations when the gauge changes value.

Only three different colours are available in the API: red, yellow, and green. However, there is an option to choose the specific tone of each colour, by specifying a RGB colour code. This came in handy because we wanted the temperature gauges on our site to have a blue range for very cold temperatures and yellow and red for intermediate and very high temperatures. The solution was thus to make the green colour tone a blue colour. Figure 3-2 shows the resulting gauges.

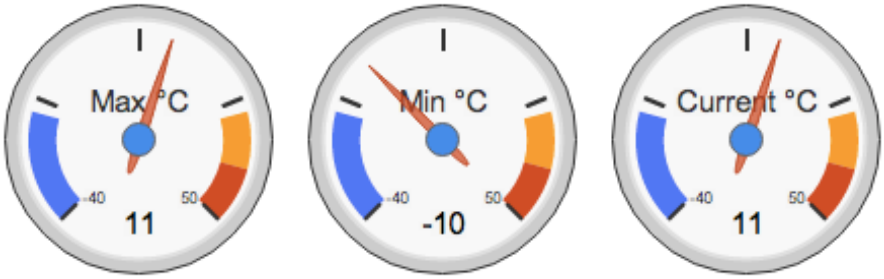

Figure 3-2 Temperature gauges

3.1.2.2 Highcharts

Highcharts [13] provides an API for good-looking interactive charts. Options such as tooltips and zoom functions, and the ability to save and print charts are already built-in. Figure 3-3 shows an example of one of our charts.

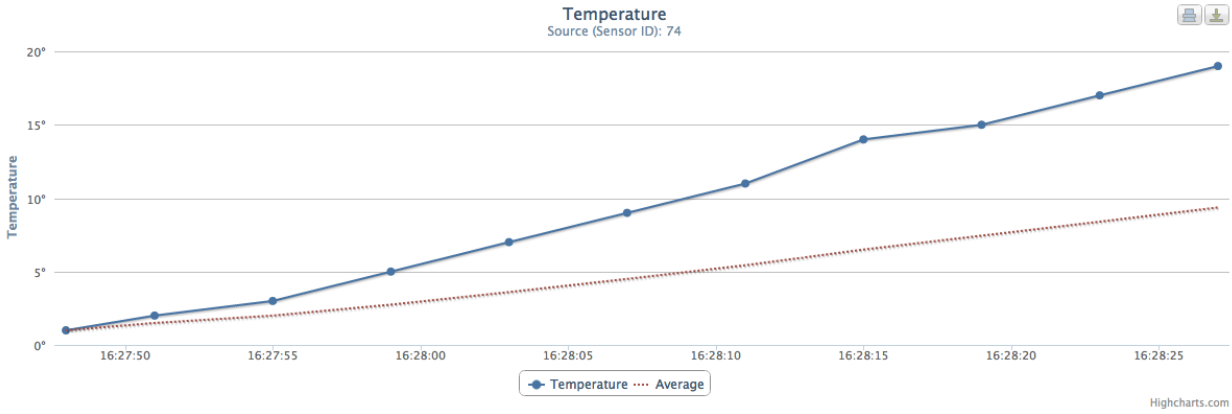

Figure 3-3 Temperature line chart using Highcharts

Highcharts also supports a great number of browsers. This broad support for browsers is of course important in systems such as we are developing because users are expected to utilize many different browsers. The browsers and their versions that Highcharts support are listed in Table 3-1.

Table 3-1 Currently supported versions of the most common browsers

Brand	Versions supported
Internet Explorer	6.0 +
Mozilla Firefox	2.0 +
Google Chrome	1.0 +
Safari	4.0 +
Opera	9.0 +
iOS (safari)	3.0 +
Android browser	3.0 +

Highcharts is implemented by downloading two JavaScript files including the Highcharts API and importing them into the project. This file and its content can be downloaded directly from the Highcharts webpage [13]. To be able to create the chart and include the correct data, a new JavaScript (JS) file had to be created. This JS file uses the downloaded Highcharts API to create and customize a chart. The chart we created to display the temperature data is shown in Figure 3-3. Here we have plotted the temperature as a function of time. The chart contains both the actual temperature and the average temperature. The average temperature is calculated for each new temperature entry and displays the running average from the first temperature displayed up to the current point.

To be able to display live data in our chart we use a subscribe method. This method listens to a Faye channel and receives data pushed by the Faye server. The data received from the channel is added to the chart values and the chart will be updated without having to update the whole page. This method is presented further in section 3.1.3.

3.1.3 Asynchronous message passing

One major feature of our home automation system is to create a truly dynamic and flexible system that could be used for multiple services. Users should be able to upload their sensor data to our servers and immediately be able to view the data on our web site. This requires some kind of service that listens for incoming changes in the database and then updates the web site without the user having to

reload the page. Unfortunately, Rails is not very good at handling these kinds of services, when some process listens on some stream of data, mainly because it is not possible to maintain an open connection to the Rails server. However, our service needs to be able to receive data from our users and at the same time display the new data on the web page.

One solution to this problem is to use Faye [14]. Faye is a push-notification messaging system that allows Faye-clients to listen in on a "channel" that is published somewhere on the web page. The clients will then handle the incoming data and reload the parts of the website that are affected by the incoming data. Each logged in user listens to its own data stream channel and the data will be pushed to the web site for the user to view. The server side of Faye, where the clients are publishing their data is described in section 3.4.

3.2 Creating the data relay program

The data relay program is a vital part of the project and one of its purposes is to act as a server and handle incoming data insertion requests from a gateway. The data relay program handles these requests by checking the validity of the received message and by checking if the requested action should be performed. The program handles requests by adding them as tasks into a queue. Subsequently, threads extract tasks from the queue and evaluate them. The other purpose of the data relay program is to do insertions into the database. The data to be inserted is extracted from the message and checked so that only valid data is inserted into the database.

The data relay program is written in Java because it is easy to use APIs such as JDBC [15] and java.net [16]. Another reason why we decided to create the program in Java was because we already had some experience in creating distributed and parallel programs in Java.

The data relay program is divided into two parts, a server part that handles the incoming messages and a database insertion part that checks the validity of the messages and does the insertions into the database. In order to test the program a gateway emulator (described in section 3.6) was developed.

The data that is sent to our relay program has to follow the XML 1.0 standard with the UTF-8 encoding. Multiple elements can be sent in one package as long as the total size does not exceed 992 byte. If the total size of the XML structure size is lower than 992 byte, padding is needed to reach the desired 992 byte. The IV is added to the end of the package to make a total of 1008 byte. Each of the elements needs to contain the following five attributes: gateway_id, sensor_id, value, type and time. The requirements for a valid message are listed in Table 3-2. Appendix A describes two example messages that follow the correct XML structure.

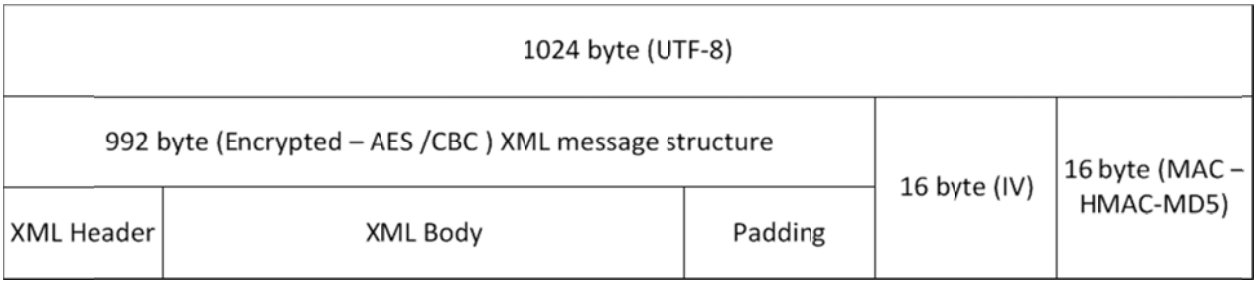

Figure 3-4 Required UDP body structure

Table 3-2 Structural requirements for the UDP body

Requirements
Total size of 1024 Byte
The first 992 byte have to be encrypted using AES-128 and the block-cipher mode CBC
The last 16 Bytes have to be a MAC generated with HMAC-MD5

Byte 992 to 1007 has to contain the IV used to encrypt the message
The message need to have specified all five values
The time value have to be in the format of "YYYY-MM-DD hh:mm:ss"

3.2.1 Handling incoming UDP messages

The interface towards the gateways that sends messages to our system is one part of the data relay program. We call this program "ServerUDP" and its job is to handle incoming UDP messages from gateways and to relay the data received to the database. ServerUDP also forwards the data to a Faye server, which is described in section 3.2.2. The gateway interface is designed to receive a UDP datagram containing an AES encrypted message with the size of 992 byte, a 16 byte IV, and a 16 byte MAC. UDP is used because it provides connectionless communication between a gateway and the relay program. This is suitable because it provides higher speed and requires less work for the gateway in comparison with using TCP. Furthermore, the individual data values that are to be transmitted are assumed to *not* be critically important and data is generally only produced at a relatively slow rate, hence the flow control and error correction provided by TCP are unnecessary. Hence we exchange the guarantee of correct in order delivery provided by TCP for higher speed and less work for the gateway by using UDP. A summary of the comparison between using UDP and using TCP is shown in Table 3-3. If the individual data values were more important, then we would need to add some reliability mechanism, however this was never our goal and has therefore has been left for future work.

Table 3-3 TCP and UDP comparison

TCP	UDP
Reliable	**Unreliable**
Connection-oriented	**Connectionless**
Segment retransmission and flow control through windowing	**No windowing or retransmission**
Segment sequencing	**No sequencing**
Acknowledge segments	**No acknowledgement**

Our program ServerUDP is designed to handle multiple requests in parallel. This parallelism is provided by a number of threads that operate on the list of received requests. By doing so, computationally expensive operations in the program, such as decryption, insertions into the MySQL database, and handling the Faye serve communication are handled by a separate thread, allowing the main thread to continue to listen for incoming UDP messages. However, the lack of flow control in UDP limits the parallelism and sending packets at too high an aggregate rate could result in severe packet loss. This limit is further discussed and presented in section 3.7.2.

We create a shared synchronized object containing the list of tasks that we call "Pool of Tasks". These tasks provide the concurrency in our program. By using a linked list we can easily add and remove elements in a first-in-first-out (FIFO) order. Threads that will handle the task and do the

necessary operations extract elements from the linked list. The number of worker threads that handle the tasks in the list is specified as a parameter at runtime. The default number of threads is 4. The concurrency architecture is presented in Figure 3-5.

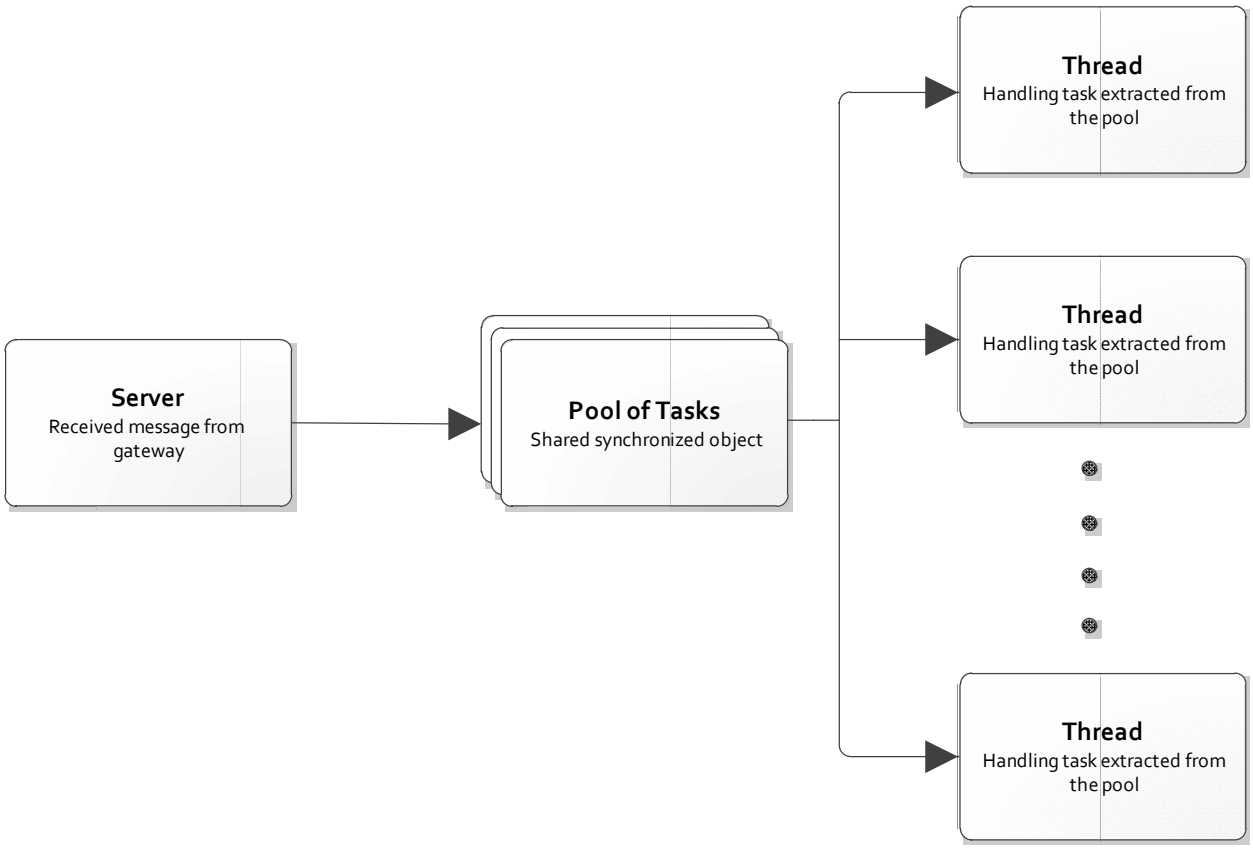

Figure 3-5 Concurrency architecture

To be able to listen for incoming UDP messages we use a datagram socket. This class is provided by the java.net API and the socket is the receiving point for a packet delivery service. The received data is stored in a byte array that has a length of 1024 bytes before it is added to the synchronized linked list.

The socket is bound such that it listens to one port, currently specified as UDP port 50000. This port was chosen because it is in the range of private ports according to IANA [17]. The data received through the datagram socket is assigned to a datagram packet structure, which provides functions that can be used to read the packet's content. If this application were to be extended and adopted for broader use, then an application should be made to the IANA to get an assigned UDP port number.

The UDP packet structure is illustrated in Figure 3-6 and is quite simple, with only four header fields and a data field. All the information in these fields can be accessed using the java.net API [16].

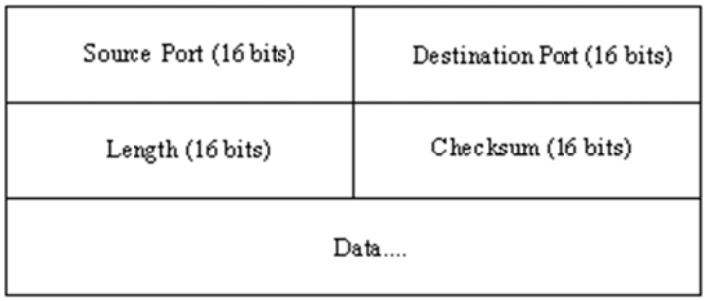

Figure 3-6 UDP packet structure

3.2.2 Connecting to the database and Faye

For each message received from a gateway a new task will be added to the pool of tasks. The threads spawned at runtime will extract tasks from the pool and run methods to decrypt, parse, and push the received data. The program communicates with the database by using a Java-to-MySQL connector that converts JDBC calls into manageable code to be interpreted by the MySQL database. The connector is distributed as a .jar file and needs to be linked together with the Java program at run and compile time. To be able to compose and execute queries in the Java program the JDBC API [15] is needed. This API is used in three simple steps: connecting to the database, sending queries, and retrieving a result from the database. Connecting to the database is a simple line of code in the program:

```
DriverManager.getConnection(”jdbc:mysql://URL”, ”UserName”,
”Password”);
```

Sending queries to the database can be done with two different kinds of queries: Statement and PreparedStatement. The simple Statement lets us send a composed MySQL query as a simple string and the PreparedStatement uses a similar structure, but instead of adding values directly into a string, the statement is prepared with question marks. The query is sent and the database holds the query while we send values to replace all the question marks, when all the question marks have been replaced we execute the query. The PreparedStatement is used for security reasons, as it is resilient against SQL-injections. The PreparedStatement can also be used to execute the query multiple times without having to recompile the query. The API also provides a result object where results from a query are stored. The result object is called a “resultSet” and allows us to use functions to extract rows and entries from the result that we can use for our own purposes.

Each message that is processed by a thread follows the same structured flow. This flow is presented in Figure 3-7.

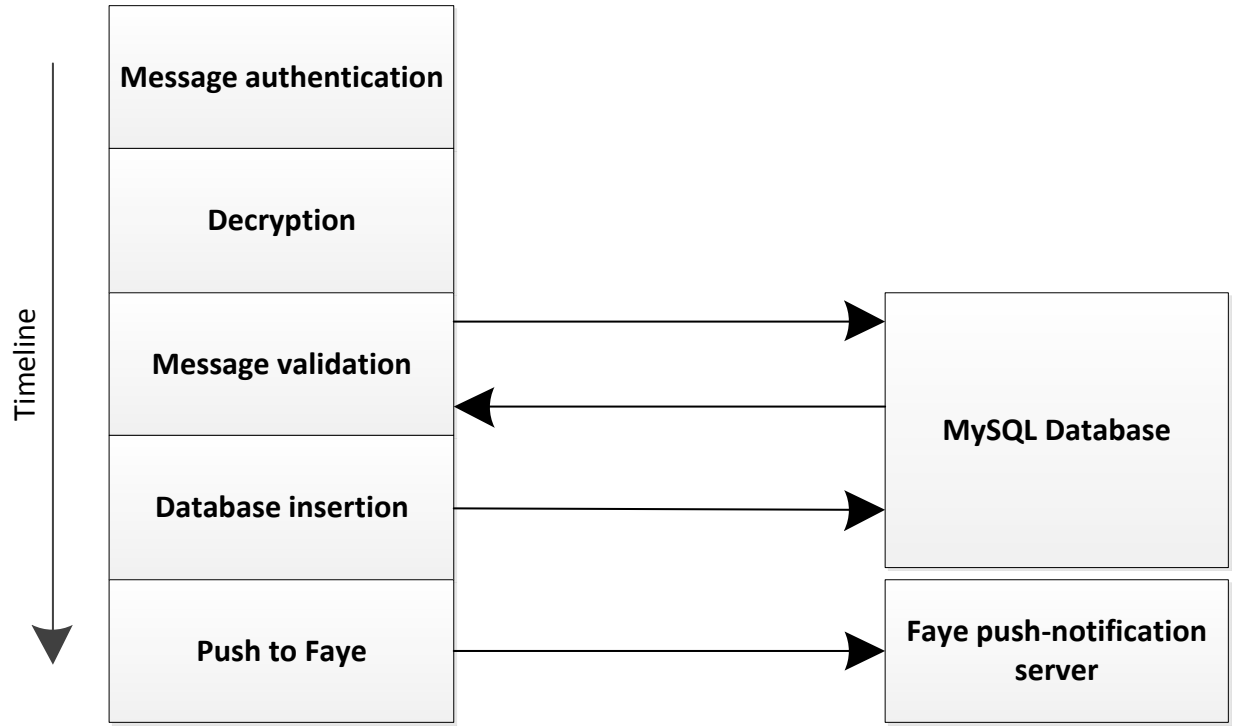

Figure 3-7 Insertion and push program structure

The data that the thread receives is still encrypted and needs to be decrypted in order to be evaluated. Decryption and message authentication are the first step in the process. Message authentication is done by calculating a Message Authentication Code (MAC) and comparing it with the MAC that is appended to the message. The decryption is carried out by using the AES algorithm. More information about message authentication and decryption is presented in section 3.3.1. If the message authentication or decryption fails an exception will be thrown and the thread will ignore this task and select a new one.

If the message authentication and decryption succeed, then the thread will continue to extract the information from the decrypted message, parse the message, and assign values in the message to separate variables. It is essential that the sender properly composed the message (as specified in Table 3-2). If not, the extraction operation will fail and the thread will throw an exception, drop the task, and extract a new task from the pool.

The values extracted from the decrypted message need to be validated before insertion of any data into the database is done. This validation is done in two steps: (1) executing a query on the database to get further information about the values and what they correspond to in the database, and (2) evaluating the reply from the database to decide how and if the data should be inserted. We composed the query so that it returns one row with three columns in it. The purpose of this query is to check if the gateway value extracted from the received message exists in the database and that the sensor value is listed in the database. We also use it to learn the most recent insertion of a sensor reading for this sensor. To achieve this; the query has to contain nestled SQL `select` statements that are more time consuming than a single `select` statement. Even though a nestled `select` statement is time consuming it is still faster than sending multiple queries to the database. The query can return four different types of results. These four types of results are shown in Table 3-4.

Table 3-4 Possible query results

Cases	Gateway	Sensor	Time
Gateway does not exist[1]	-	-	-
Gateway exists **Sensor does not exist**	Gateway ID	NULL	NULL
Gateway and Sensor exists **No readings from the sensor**	Gateway ID	Sensor ID	NULL
Gateway, sensor, and readings exists	Gateway ID	Sensor ID	Timestamp

The gateway column is needed to be able to check if the gateway exists in the database, the sensor column is needed to be able check if the sensor exists in the database, and the time column is used to check when the last entry was inserted. The value from the time column is important because we want to limit the number of insertions that can be done by a sensor within a given interval of time. We have set this limit to allow only one insertion every two seconds.

If the gateway does not exist in the database, then the thread will be interrupted and will ignore the task. This could occur because a user did not add a gateway thru the webpage before this gateway sent data to the database. We require that the user explicitly add gateways in order to be able to link users registered on the webpage with specific gateways. In the case of a gateway existing but a sensor not existing, then we add a new sensor to the database, create a link to the gateway it was sent through by adding a specific “gateway_id” field in the sensor table, and then add the sensor data. In the other

[1] In any case when the gateway does **not** exist; the query will return an empty result.

cases we insert a new sensor reading, with the exception of when the rate reaches the limited number of insertions per second.

To be able to push messages to the Faye push-notification server we have to compose a HTTP POST message. The data sent to the Faye server must be structured as a JSON object or the Faye server will not be able to interpret the data. To be able to create a JSON structured message we used JSON objects. These objects handle the transformation from strings and values to JSON structures. The JSON message has to follow a special structure defined by the Faye server. This structure is presented below:

```
{channel : /subscribe/channel, data : {
   value : sensorValue, created_at : time}
}
```

The channel that we send the data to is dependent on what data is being sent. The channel address is specified by the sensor id, which allows us to only push data to users that own this sensor.

All failures during the operations on a message will cause an exception and will interrupt further operations on the task. This is because we do not want any partially handled request to affect the database or the Faye server. After a thrown exception the thread will continue by extracting and handling new tasks from the pool.

3.3 Security

Security is a vital part of our project, as we want to provide the user with a reliable and secure service. We provide this security by encrypting plain text data and by message authentication. The communication is only as secure as its weakest link and both the sender and receiver have to fulfil the required security preconditions to provide sufficient security. Because we communicate over the unsecure Internet, we have implemented security that is designed to resist several different types of attacks against us. In this chapter we show the security provided by our system.

3.3.1 Encrypting the UDP traffic

The sensor data collected in a home environment can be very personal and a user should not have to worry about that data being compromised. To guarantee that only authorized users can read and send sensor data, encryption algorithms are used to encrypt the data. The gateway interface is designed to only receive messages sent by properly enrolled gateways, which means that it only has to handle decryption of small messages containing sensor data. The gateway that sends the messages needs to implement the same encryption algorithm as the gateway interface and to use the same key. Note that we are assuming the use of a symmetric encryption algorithm, as an asymmetric algorithm would require too much processing time on both the gateway and the receiver.

We are using AES as the encryption/decryption algorithm. AES is a standard issued by the U.S. National Institute of Standards and Technology (NITS) and is approved by the U.S. Department of Commerce. AES uses the Rijndael algorithm developed by Joan Daemen and Vincent Rijmen [19]. In our implementation we encrypt and decrypt a 992 byte array with AES. AES has a fixed block size of 16 bytes, which means that we have to divide our array into multiple blocks. This also means that we can use a block-cipher mode of operation. In our case we use cipher-block-chaining (CBC), which provides extra security by XORing previous cipher text blocks with the next plaintext block. To be able to XOR the first block an initialization vector (IV) is needed to ensure that there is varying content in the first block (to avoid a known plain text attack). The IV is composed by creating a 16 byte array. This array is subsequently used as input to the encryption and decryption algorithms. To ensure that every message is encrypted into a different cipher text the IV is randomized and then appended to the UDP message in plain text. Each new message utilizes a different IV, thus ensures that each message is encrypted in to a different string even if the same data is sent, with the exception of exactly the same data being sent with the same IV. However, this is very *unlikely* as the numbers of permutations in the IV and byte array are very large. Because the IV has a length of 16 bytes, i.e. 128

bits, the number of permutations is or ~ . The CBC encryption and decryption methods are illustrated in Figure 3-8 and Figure 3-9.

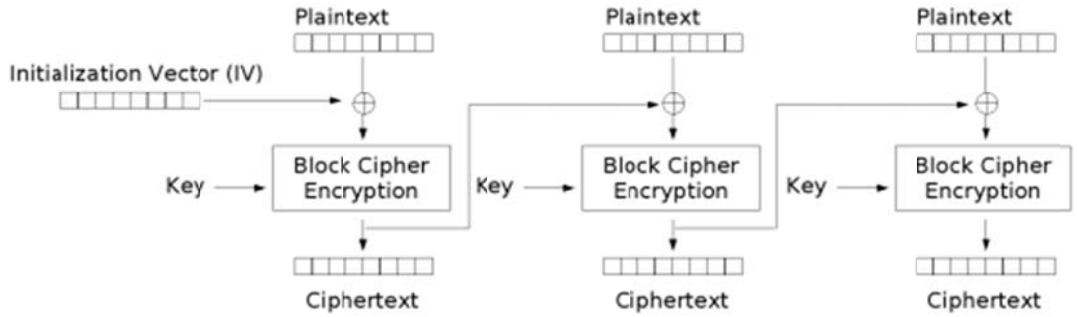

Figure 3-8 Cipher-block chaining (CBC) mode, Encryption

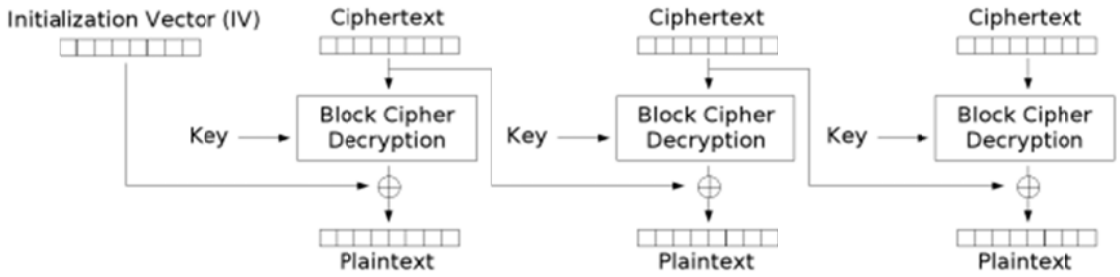

Figure 3-9 Cipher-block chaining (CBC) mode, Decryption

To guarantee the message's integrity we use a 16 byte Message Authentication Code (MAC) that is appended to the end of the message. This MAC is calculated using the MD5 cryptographic hash function to produce a 16 byte hash value. The MAC is generated by using the hash algorithm on the entire encrypted message (in our case this means the first 1008 bytes) together with a secret key.

To be able to verify the integrity of the message we calculate the MAC for the first 1008 bytes together with the secret key and compare the generated MAC with the MAC appended to the message. If these match, then we know that the message has not been tampered with and we can safely open it and evaluate its content.

3.3.2 Internal system security

Security may also be necessary for the communication between servers connected to the internal network of servers, so that within this network the servers should have access to each other but access from others should not be possible. Generally these servers will be located in a single geographical location and therefore their close proximity should result in low delays for this communication.

VPN connections could be used to bind together different instances of the server networks when they are not located at the same physical site. This enables a logical local network to span over a larger area; however the assumption of low delays between servers may no longer be valid. However, the investigation delays in such a VPN network are outside the scope of this thesis and will not be discussed further.

In our test environment we run the different parts of our system on the same server. This gives us a greater level of security, as a firewall only allows connections via certain ports to our system.

3.3.3 Key sharing

In our implementation of the ServerUDP program we need keys to perform encryption with AES and to produce a MAC. These keys are shared secrets between the gateway and the ServerUDP program. These keys have to be secret in order to guarantee the required security. The distribution of these keys needs to be handled in a secure way to avoid compromise of the secret. In our tests of the system we handle both sides of the communication, which makes it easy to manually distribute the keys. Our goal to offer an open interface to which a user can send their sensor data, but properly handling key sharing among the nodes is a more complex project and something that could be integrated in future work.

3.3.4 Problems with decryption of messages sent with UDP

In our implementation of the AES encryption algorithm we encountered some problems regarding the decryption of messages sent over UDP. While experimenting with the algorithm we first implemented the algorithm and ran it within the same program and this worked without any problems. When we separated the encryption and decryption functions to the separate sites of the UDP communication a problem occurred, as we no longer could successfully decrypt the received message. We knew that the algorithm worked because of our initial experiments, which meant that the problem had something to do with the UDP communication between the two hosts.

Trying to decrypt the encrypted message generated this exception:

```
Exception in thread "main"javax.crypto.BadPaddingException: Givenfinal block not   properly padded
```

This problem occurred because messages sent over the UDP channel were specified to be 1008 bytes in size and the encryption algorithm encrypted only the string and not the intended 1008 bytes. We solved the problem by first adding extra padding to the message in order to generate a message of the desired size and then encrypt the whole 1008 bytes.

3.4 Push notification server (Faye)

Faye uses TCP connections to communicate to and from the client and the Thin web server [20]. It does this by using the Ruby library “eventmachine” [21].

Faye is easily installed with the “gem” command and could be deployed on its own separate server. The Faye server pushes messages to all the clients that are currently listening on the correct channel. This allows the browser to receive messages in the background *without* the user having to refresh the whole page. It also gets around the problem that the Rails server does not allow any open TCP connections.

To implement these operations requires adding the following code to a ruby file to set up the Faye server.

```
require 'faye'
Faye::WebSocket.load_adapter('thin')
faye_server = Faye::RackAdapter.new(:mount => '/faye', :timeout => 45)
runfaye_server
```

Running the following command starts the Thin server:

```
$rackup “filename” -s thin -E production
```

In Linux or Unix the server process can be started in the background as a daemon process by adding "&" to the command above. This enables the service to remain running even after the terminal window is closed or if the user disconnects from the SSH connection.

3.5 Database

In our thesis project we have used the MySQL program as our database, since it is freely available and provide the functionality that we need.

3.5.1 MySQL

MySQL is one of the most commonly used database solutions and is used by companies such as Facebook, Google, and Adobe. It is built with performance, reliability, and ease of use in mind. The fact that it runs on multiple platforms also makes it very flexible. Tools such as phpMyAdmin (discussed in section 3.5.2) make interaction with the database very easy and accessible from anywhere via a web browser. A MySQL service was set up on a server running Ubuntu 12.04LTS. The installation is done simply by executing the command:

```
$ sudo apt-get install mysql-server
```

By default the server only listens to the localhost IP address, thus the server has to be reconfigured to listen for remote connections as well. The configuration file that handles this is the "my.cnf" file located in the "/etc/mysql/" folder. The necessary change is made by editing the "bind-address" variable to be the local IP address of the server or by binding it to "*" to allow for connections on all of the host's IP addresses.

```
bind-address = 192.168.0.5
```

or

```
bind-address = *
```

TCP port 3306 is the standard port for MySQL according to IANA [17], so this is the port number that we have used.

The following command was used to verify that the server was up and running.

```
$ sudo netstat -tap | grepmysql
```

3.5.2 phpMyAdmin

Ruby on Rails uses an sqlite3 database by default and although there are visualization programs and interfaces, such as SQLite database browser [22], we thought that it would be easier to switch to a MySQL database and use the phpMyAdmin [23] tool to manage the database. The phpMyAdmin tool is a web-based interface that allows the user to view and edit the database directly in their web browser. This tool is written in PHP with the intention administering MySQL databases.

Furthermore, we wanted to have the database in one place, rather than running on our individual computers. Therefore, we decided to move the database to a remote server. To achieve this an apache2 web server was set up on our Ubuntu 12.04LTS server. The tool phpMyAdmin was then installed and configured to function together with the apache2 server.

We also started a dynamic DNS service on the server that automatically updates the domain name "biifer.mine.nu" to the server's global IP address. This domain is registered to us at the site: http://dyn.com/dns/. Having a domain name means that we can look up this name in order to find the *current* IP address of the server, thus allowing us to contact the phpMyAdmin tool from anywhere by just using the URL: "biifer.mine.nu/phpmyadmin" rather than having to know the server's current global IP address. This also has the advantage that in the event that the server gets a new global IP address from its ISP, then the address will automatically be updated with the associated domain name.

3.5.3 Database security

MySQL supports traffic encrypted with both SSL and SSH. Both of these methods enable secure connections between the MySQL server and any client that is trying to access the database. The SSL method is the preferred connection type to use because it provides a connection *directly* to the MySQL server, the SSH method provides only a secure tunnel to the *host* where MySQL is running. The connections to MySQL are not secure by default, as this insecure option is slightly faster. In our environment we use the same host for our MySQL database as our ServerUDP program, which already receives encrypted traffic. For this reason, encrypting the traffic between the ServerUDP program and the MySQL data is not critical; hence implementation of an SSL based tunnel has been left for future work.

3.6 Gateway emulator

Because we did not have access to the gateway that Albert and Francisco were building we developed a test program that would encrypt and send UDP messages in order to emulate how a sensor would send data. This gave us a platform on which we could test and develop the receiving side of the connection. The program we developed for this emulation is written in Java and uses a UDP socket to send data packets to the receiving program.

At first the UDP connection between our test program and the receiving server was encrypted with a 3DES encryption algorithm, this was later changed to an AES with the cipher-block chaining (CBC) encryption algorithm to increase performance and to increase security. Initially 3DES was chosen because we found a MSP430 compatible implementation of the algorithm on Texas Instruments' web site. Later we implemented our own encryption and MAC calculations to match the receiving side's implementations.

3.7 On a large scale

Because this project aims for a solution for a home environment, the system must be able to scale up to support a potentially large customer base; there are a few potential bottlenecks in our system that must be investigated. The system can be divided into 4 basic parts: the web server, a MySQL server, a Faye server, and a server that handles the incoming UDP messages. To make this system scale well for large numbers of customers and datasets a few actions needs to be taken. The following subsections will consider each of the parts of the system and how it scales.

3.7.1 SendUDP

To be able to investigate the limits in our ServerUDP program we modified it to display the test data. The test was conducted by sending multiple packets to the server to see how it performed.

The first step in the test was to investigate how fast we could send UDP packets. In this test we used our laptops, a Sony Vaio VGN-SR59VG and an Apple MacBook (late 2008). The Sony laptop will be referred to as "Vaio" and the Apple as "MacBook" in the remainder of this report. The specifications of each laptop are listed in Table 3-5.

Table 3-5 Laptop specification

	Sony Vaio VGN-SR59VG	Apple MacBook (Late 2008)
Processor	Intel Core 2 Duo P8700@2.54 GHz	Intel Core 2 Duo P8600@2.40 GHz
RAM	4GB DDR2@800 MHz	4GB DDR3@1067 MHz
OS	Windows 7 Professional SP1	Mac OS X 10.6.8

The test was conducted by creating a simple test program that sent multiple UDP message to the server as fast as possible. The test program (described in Appendix B) is not by itself a test of the

ServerUDP program, but rather test to see if our laptops could be used to test the performance of the server. The test was performed with three different sizes of UDP packets to investigate how the performance was related to packet size. In this test we send 100k UDP packets in a loop as fast as possible and we ran the program 10 times. When each test was finished we compared timestamps to determine how long time it had taken to send these 100k UDP packets. The results of this test with the MacBook are shown in Figure 3-10 and the results of the Vaio in Figure 3-11.

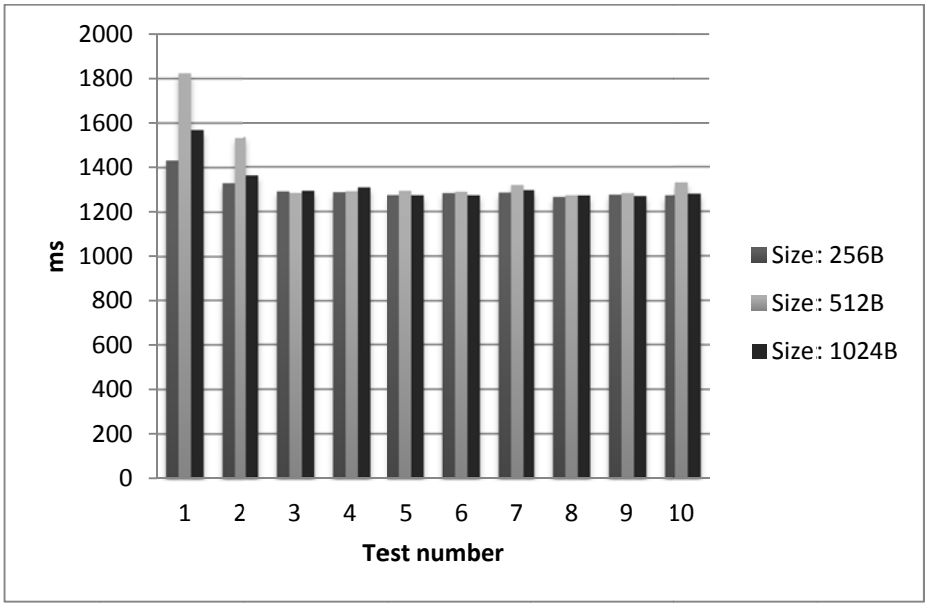

Figure 3-10 MacBook UDP send test results

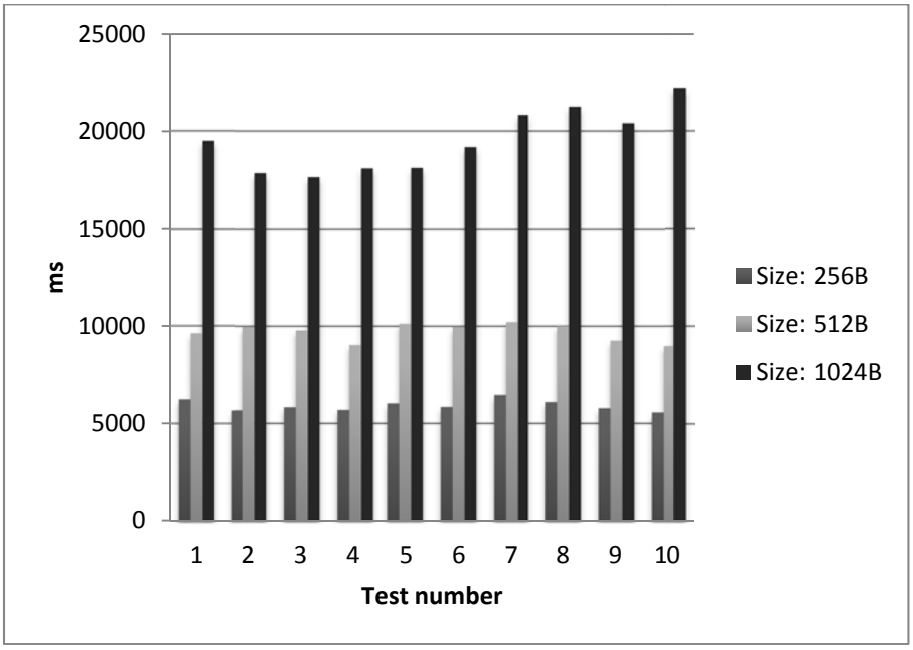

Figure 3-11 Vaio UDP send test results

The results show that there is a quite big difference between the Vaio and the MacBook. The MacBook has a higher average speed of sending packets than the Sony. However, by checking the output in Wireshark we found out that the MacBook dropped a lot of packets, while the Vaio did not drop any packets at all. Additionally, for the MacBook the number of dropped packets increased as the packet size increased. In a test with 1024 bytes in each UDP packet almost 75% of packets were

dropped before even being sent. However, this is somewhat to be expected because the MacBook is trying to send data at a rate of over 650Mbps. This speed is not possible on achieve over the IEEE 802.11n network that the laptop was connected to at the time. An IEEE 802.11n network with the Modulation and Coding Scheme (MCS) index of 15 is only capable of speeds up to 300 Mbps [24]. Moreover, the MCS varies as the signal strength of the wireless network changes. During our tests the transfer speed was hovering around 130Mbps. Figure 3-12 shows a sample reading of the network performance on the MacBook. Figure 3-13 show how many packets that were dropped for each size of UDP package.

Figure 3-12 Transmit rate with a MCS index of 15

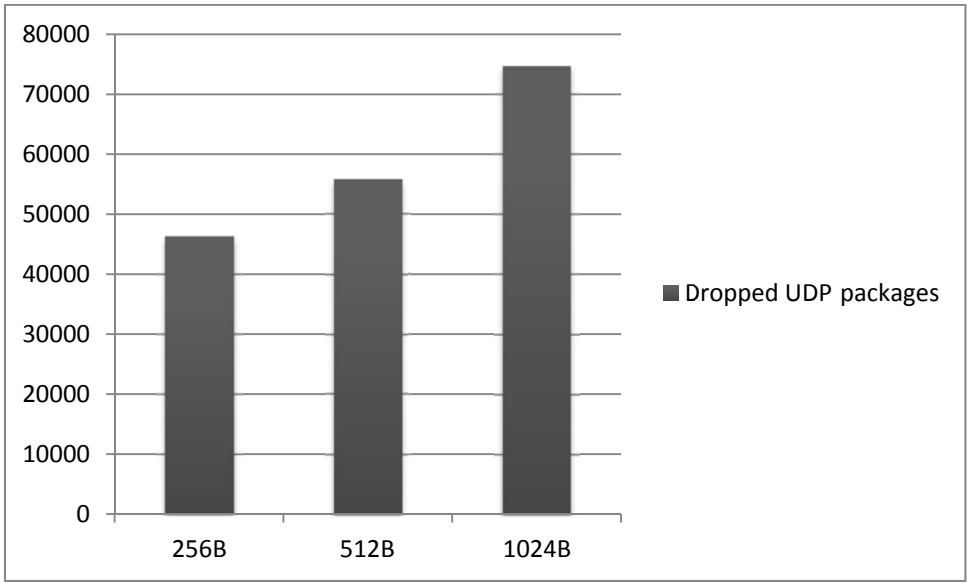

Figure 3-13 Dropped UDP packets as a function of packet size on the MacBook

We think this huge loss of UDP packets is due to the MacBook trying to send the UDP packets too fast and that the hardware simply cannot keep up (although it would seem that the driver should be checking to see of the output queue has room before putting an additional packet into this queue). The Vaio seems to send the packets a lot slower and is therefore able to send without losing any packets.

We used Wireshark to calculate the true throughput from the Vaio and the MacBook and the results are shown in Figure 3-14. These results suggest that even though the MacBook dropped most of the packages the speed is still much higher than the Vaio.

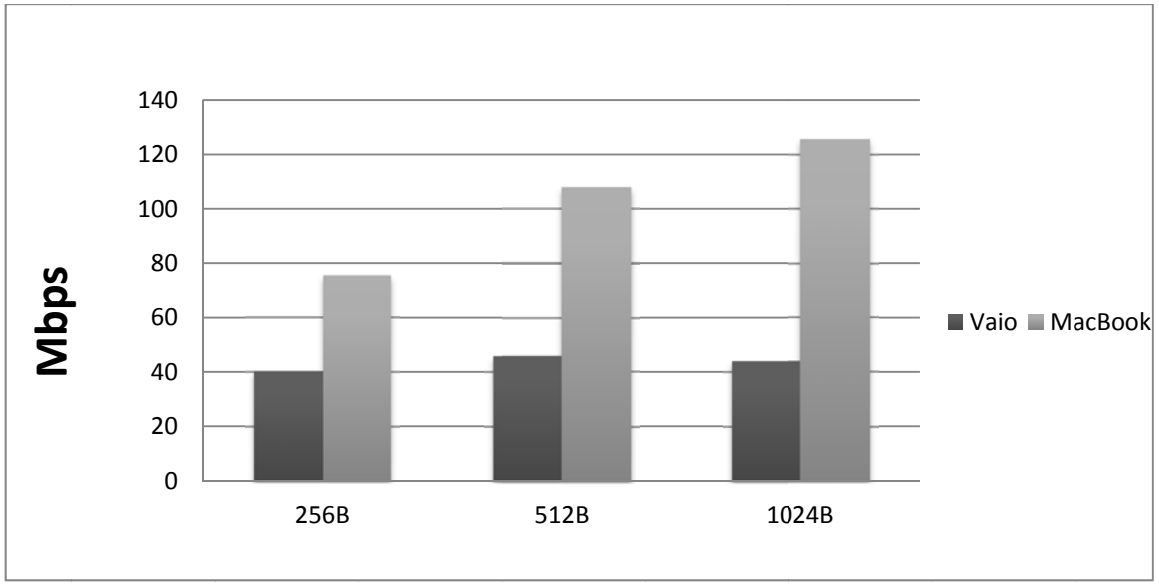

Figure 3-14 True throughput of the Vaio and MacBook

The most interesting result from the UDPSendStressTester program is that we know how fast each computer can send messages to the ServerUDP program. As we do not care about packet losses from the sending side, the MacBook can still be used to send packets because it has a higher actual throughput.

3.7.2 ServerUDP

The second step was to see how many UDP messages the server could receive within a given period of time. We did this by using the UDPSendStressTester program from step one to send packets to our server. The test investigates how many packets our server can successfully receive per second. We wanted to test how long it took the server to receive, decrypt, and send the message to the Faye-server and the MySQL database. We also considered in this test that a larger packet could contain multiple messages, and therefore we could send multiple sensor readings in one UDP packet. The XML message header is quite large in this context, which gives the UDP body a fixed start-up cost. This means that the start up cost for the UDP body is the same even though the number of messages in it varies. Of course UDP and IP headers also must be added to the total number of bytes being sent on the network. Thus a 256 byte raw XML message results in a 288 byte IP datagram after including the UDP and IP headers. Correspondingly a 1024 byte XML message becomes a 1066 byte IP datagram.

An example of a valid XML message is presented in Appendix A to facilitate further understanding of the fixed cost and the structure that we have chosen to follow. The number of messages that can be fit in to a packet depends on the length of each message. A calculation over the total XML structure size containing a variable number of messages is shown in Table 3-6.

Table 3-6 XML structure length with variable number of messages[2]

Header size (byte)	Root element size (byte)	Number of messages	Total message length (byte)	Total structure size (byte)
38	0	**1**	113	151
38	13	**2**	226	277
38	13	**3**	339	390
38	13	**4**	452	503
38	13	**5**	565	616
38	13	**6**	678	729
38	13	**7**	791	842
38	13	**8**	904	955
38	13	**9**	1017	1068

We conducted these tests on a server running Windows server 2008 R2 with a 100/10 Mbps broadband connection and the result from performing a "ping" request to the server showed that we had an average RTT of 26.6ms. The complete results from the ping request are shown in Figure 3-15.

```
ivanp@ccscenter:~> ping itard.dyndns.biz
PING itard.dyndns.biz (77.53.185.61) 56(84) bytes of data.
64 bytes from h77-53-185-61.dynamic.se.alltele.net (77.53.185.61): icmp_seq=1 ttl=51 time=27.3 ms
64 bytes from h77-53-185-61.dynamic.se.alltele.net (77.53.185.61): icmp_seq=2 ttl=51 time=26.4 ms
64 bytes from h77-53-185-61.dynamic.se.alltele.net (77.53.185.61): icmp_seq=3 ttl=51 time=26.5 ms
64 bytes from h77-53-185-61.dynamic.se.alltele.net (77.53.185.61): icmp_seq=4 ttl=51 time=26.6 ms
64 bytes from h77-53-185-61.dynamic.se.alltele.net (77.53.185.61): icmp_seq=5 ttl=51 time=26.4 ms
64 bytes from h77-53-185-61.dynamic.se.alltele.net (77.53.185.61): icmp_seq=6 ttl=51 time=26.4 ms
64 bytes from h77-53-185-61.dynamic.se.alltele.net (77.53.185.61): icmp_seq=7 ttl=51 time=26.4 ms
64 bytes from h77-53-185-61.dynamic.se.alltele.net (77.53.185.61): icmp_seq=8 ttl=51 time=26.4 ms
64 bytes from h77-53-185-61.dynamic.se.alltele.net (77.53.185.61): icmp_seq=9 ttl=51 time=26.4 ms
64 bytes from h77-53-185-61.dynamic.se.alltele.net (77.53.185.61): icmp_seq=10 ttl=51 time=26.4 ms
^C
--- itard.dyndns.biz ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9013ms
rtt min/avg/max/mdev = 26.460/26.588/27.375/0.266 ms
```

Figure 3-15 ServerUDP ping results

We decided to divide the program into multiple parts to be able to pinpoint exactly where the bottlenecks in our system are. At first we simply received UDP packages and printed out how fast we was able to do this for the different sizes of packets. The results are shown in Figure 3-16.

[2] All values presented in the table are calculated from the XML message structures presented in Appendix A. These values can vary depending on the user's implementation (i.e., the total structure size is not guaranteed to be exactly the same length as in the table).

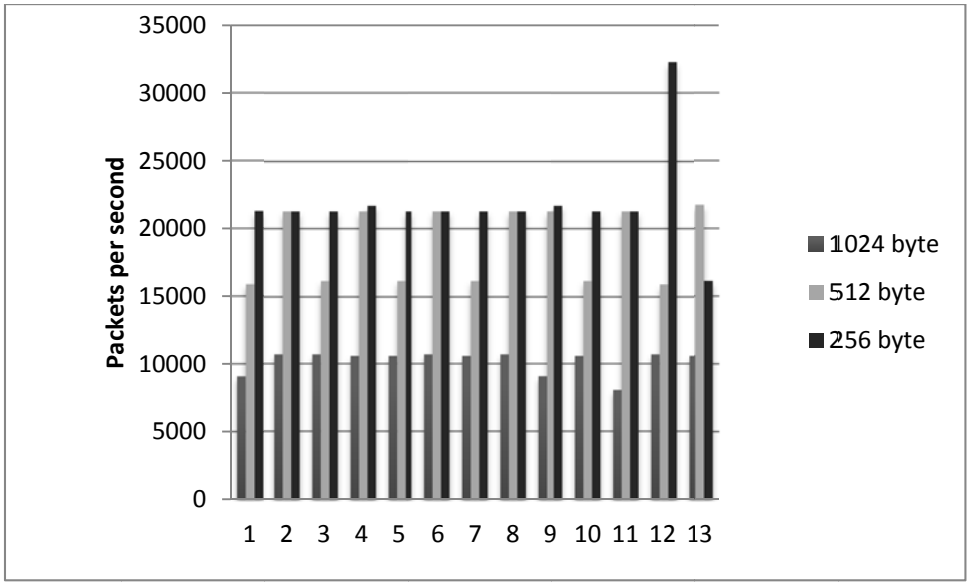

Figure 3-16 Number of received packets per second with varying packet size

The results from Figure 3-16 show that we are not able to receive as many packets per second of the larger size (1024 bytes) as the smaller 256 byte packets. However, the total amount of data is of course greater for the 1024 byte packets and therefore the throughput is better with a larger packet size. We have calculated the throughput according to the following formula, where Ps is the total IP packet size, Pn is the number of packets, and t is the time in seconds.

The throughput was then calculated from the values shown in Figure 3-16 and an average throughput for each packet size was calculated. These results are shown in Figure 3-17.

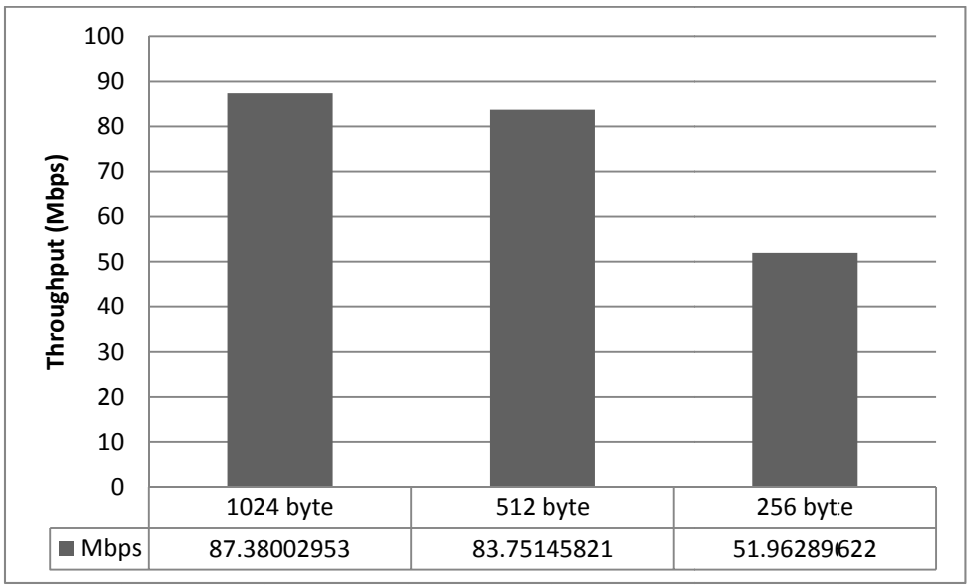

Figure 3-17 Throughput coming into the server as a function of different packet sizes

As seen from the throughput varies with the size of the packets. The best result is of course the one with the highest throughput. As we are able to send up to 8 different sensor readings in a single 1024 byte packet this option is clearly the best with respect to successfully delivering lots of data to the server. To be able to compare these results with what a user might experience, we tested our connection to the site www.bredbandskollen.se and found that the 100/10 Mbps connection actually

delivered about 96/9 Mbps (where the first number is the downlink throughput and the second number is the uplink throughput). From this result we can calculate that our connection uses about 91% of its full potential.

The last step in our tests with the ServerUDP program was to check how long each task took to process each message, i.e. to decrypt the data, check the validity of the data, put the valid data into the MySQL database, and push the data to the Faye server. We modified ServerUDP to only print timestamps when each part of the processing was completed. The results can be seen in Figure 3-18 below.

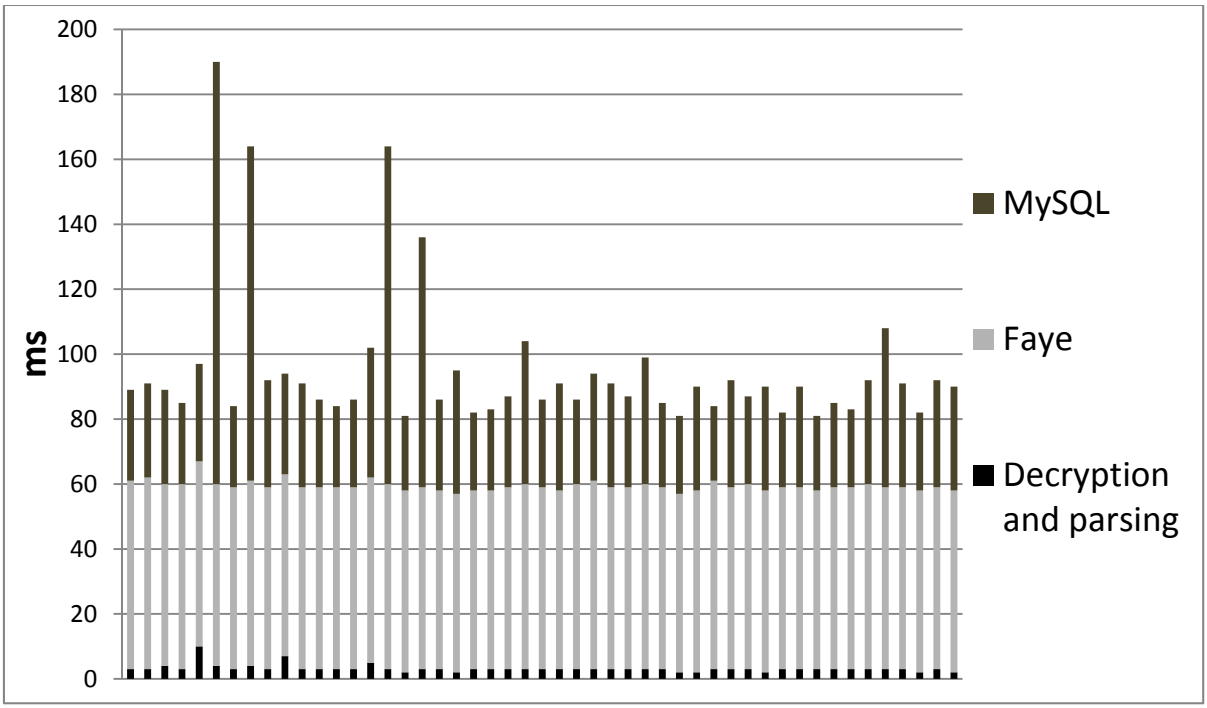

Figure 3-18 Time to process each task

These results show that the most time-consuming parts of our system are the MySQL and Faye portions of the processing. The systems performance is not very good because each task takes about 100ms to complete, and because this is done for every sensor reading that is sent to our system, the system will not able to handle that many readings per second. In the following sections (sections 3.7.3 and 3.7.4) we discuss possible solutions and optimizations to improve this performance.

3.7.3 Faye

In the current implementation each sensor reading that is sent to our system is also pushed to the Faye server, this operation of course takes some time and we have found that this is one of our main bottlenecks. In the tests that we have conducted (described in the previous section 3.7.2) we found that pushing each reading to Faye may not be the best solution. In our future work (section 4.2.6) we propose that in the future multiple readings should be pushed at the same time.

Because Faye is using an HttpURLConnection the connection is dropped between each attempt to connect to Faye. This is unlike the case for the MySQL connection where each thread in the ServerUDP program keeps a connection to the database open. By keeping a connection open and reusing the connection over and over the operations of communicating with the database avoid the processing necessary to establish a new connection each time. There are ways to reuse and keep an HTTP connection for multiple requests [25], but this has been left for future work.

3.7.4 MySQL

The MySQL database that we are using in this system is a potential bottleneck because it is located at a single point and every request has to at some point access the database. The most common

operation that the system performs is to add sensor data to the database; therefore it is important that the system is able to perform this request as fast as possible. In order to check how fast the database was able to insert a single row, i.e. a single sensor reading we did a number of insertions with the tool "mysqlslap" [26]. This is a diagnostic program that emulates a client's load on a MySQL server and returns a set of values of how long the average database request took to be processed. We used the following command to emulate how a real sensor value would be inserted. We ran this test with the iterations option in order to perform the operation 1000 times in the interest of getting a good estimate of the average value.

```
$mysqlslap --user=username --password=* --create-schema=development3 --query="INSERT INTO sensor_readings(sensor_id, gateway_id, value, created_at, updated_at) VALUES (3,27,10,1337932798774,1337932798774);" --number-of-queries=1 --iterations=1000
```

The output from the mysqlslap program was:

```
Average number of seconds to run all queries: 0.019 seconds
Minimum number of seconds to run all queries: 0.015 seconds
Maximum number of seconds to run all queries: 0.065 seconds
```

These results show that each database insert request will take on average of 0.019s. This poses a bit of a problem for us because our system sends each sensor reading to the database separately. This means that each reading that is produced by a sensor has to be sent to the database and then inserted individually. Looking at the results above, the average number of readings that can be inserted into the database is only 52 per second. This is clearly not enough as our system must be able to handle thousands of updates per second.

The solution to this problem could be to insert multiple rows with the same insert request. This means that we should collect a number of values and insert them into the database at the same time. We did another test with the "mysqlslap" tool and instead inserted 1000 rows of sensor readings with a single SQL insert statement. The results are shown below.

```
Average number of seconds to run all queries: 0.040 seconds
Minimum number of seconds to run all queries: 0.023 seconds
Maximum number of seconds to run all queries: 0.391 seconds
```

These results show that even though we are inserting a thousand times more values each time, the time to do so only increases by 21ms on average. This performance increase was bigger then what we expected, so big in fact that we suggest the system's performance can be improved by exploiting this behaviour. Instead of sending each sensor reading to the database individually we should collect values in a buffer and after some time or when the buffer reaches a certain size, then we should send an update to the database. There are a however a disadvantage in using this solution which occurs when the server that have bulked up the readings crashes. If the server crashes, all the readings that have been bulked will be lost. However the data bulked up will hold readings from different sensors as the gateways do not send information that often to the server which will only result in considerably small looses for each home. This idea is presented in more detail under further developments in section 4.2.4.

3.7.5 Test conclusions

As we have seen from the tests we have conducted there are several possible scaling bottlenecks. The operations that have to been done within a task to successfully store and push a sensor reading are time consuming and as a result we cannot reach the desired system throughput. There are possible solutions to this problem, mainly concerning the database and Faye communications. As we have seen from the results from the database test (described in section 3.7.4) the difference in time it takes to push one or multiple messages is not that large. The solution proposed is to gather multiple messages and not update the database every time we get a new reading. This solution is proposed and discussed

further as future work in section 4.2.4. However, the biggest bottleneck in the system is the Faye server, more specifically the establishing of a connection to the Faye server from the ServerUDP program. A connection established with the Faye server to send a push message can only be used once, which means that we have to establish a new connection every time we push a message. The establishment of this connection is a time-consuming operation and drastically lowers the overall performance of the entire system. A proposed solution to this problem is presented as future work in section 4.2.6.

The difference in throughput depends very strongly on the size of the packets and sending a larger packets result in a high throughput. As a result the number of packets that we will receive are lower and thus we can sustain a higher throughput. To take advantage of this higher throughput, multiple messages should be sent in one packet. A 1024 byte UDP payload has space for a 992 byte XML structure, thus we expect that it can contain 8 individual messages. This means that we can lower the number of packets sent by having the gateway store messages locally and send eight sensor readings in each message. Lowering the number packets sent to us reduces the processing cost and allows for even higher scalability. An additional advantage is that we can handle each message faster if they are contained in one packet. By handling multiple messages in one task we can also lower the cost of connection establishment to the Faye server and other onetime costs associate with a task.

4 Conclusions and Future work

This chapter suggests some future work that could be done. It summarizes work that we have thought about, but have not done. This chapter also contains a summary of our results and conclusions that we have drawn during this thesis project.

As we move in to the age of the computers and the Internet being something that we take for granted, a wide range of new devices and new implementations of old devices are becoming available for homeowners. These devices strive to facilitate the everyday life for end users and to automate the tasks of the internal systems. As these devices become more common in home environments the data produced and consumed in the home environment is increasing. Data produced within the home is rarely stored for further use and it is even more unusual for the data to be passed outside of the home. Today there are several companies that provide systems for controlling and monitoring of a home environment. However, these systems have their own closed protocols that limit the possibility of introducing new creative solutions by which the user can input their own home's data. The potential uses of the data from home environments are many, but we believe that in order to be able to move forward within this area we need to extend the home's system, out into the Internet.

In the preceding chapters we have introduced the possibility of creating a web interface for the purpose of monitoring data produced in a home environment. We have also described how we can create a scalable system by which anyone with data from their home environment could monitor their data. We have looked into the development process of the system and have reviewed several aspects of the problem that have to be considered in order to develop scalable, secure, and robust systems. The first major aspect discussed concerns web development and how to create an easy to use interface that the user can interact with wherever they are. The discussion of the implementation process of creating a web interface introduced many techniques and how they could be integrated into the service. These techniques involve asynchronous updates using a push-notification server and the use of external APIs to provide for a rapid development process. The second major aspect discussed concerns the implementation of a system of servers to manage communication, store data, and to allow asynchronous updating. The communication server provides the system with a communication point to which users can push data. We have looked into how this communication server can be developed, what is needed, and its limitations. We have also introduced techniques and APIs needed within the server's structure, such as details of the interaction between the servers, implementation of security, error handling, parsing, concurrency, and scalability.

For the last step in our development we discussed and presented the results obtained in tests on each part of the system. We discussed the likely scalability limitations in our system, what they depend on, and how we can modify the system implementation in order to scale up even further.

All these steps add up to a complete open system that has the potential to grow along with increased usage. Investment in servers, other equipment, and further software development are needed in order to be able to scale the system up to become a widely used service. As of now the possibility of launching such as service for the public in a near future is not that farfetched, however a good-looking graphical interface is believed to be needed to attract potential users.

Creation of the web interface introduced us to Rails and the easy to use interface it offered. Rails proved to be a great framework that facilitated our development. It provided us with easy to use plug-ins in the form of gems and helped us structure our code following the MVC structure. Rails gave us the opportunity to develop a fully functional web application in a short period of time, without requiring extensive knowledge about the framework. For a first time web developer, Rails is perfect as it is easy to get started and there is a lot of support along the way from the Rails community, such as via the RailsCasts website [27].

The current implementation with each individual sensor value being added into the MySQL database and pushed to Faye has some limitations with regard to scalability. The current system is able to receive messages so rapidly that the worker threads are not able to keep up. As a result our linked list (List Of Tasks) grows when there are many readings simultaneously sent to our ServerUDP program. Based upon tests of our system we have been able to isolate the bottlenecks and have

proposed some solutions to the problems that we have observed. These solutions are described in more detail in sections 4.2.4 and 4.2.6. Further conclusions regarding the scalability can be viewed in section 3.7.5.

4.1 Reflections

In this section we will present our reflections about how our service possibly could affect the user and the society in multiple ways such as economical, environmental and ethical aspects.

As we have created a service that aims to allow a user to monitor almost everything in a home environment, changes in the lifestyle of a user is possible as the user can gain further information about the home environment in an easy to access way. Furthermore, the possibility of affecting the society positively regarding the environmental sustainability is also possible.

4.1.1 Economics and environmental aspects

Our service enables the user to monitor almost anything. This means that the user can gain additional information about for example the power and water consumption. Based on the information gained, the user could make decisions to lower the overall cost of the home environment by for example lowering the power consumption.

Our service in itself does not directly affect the economics for a user but instead aims to raise awareness about what is going on in the home environment to possibly change the lifestyle of the user. A greater understanding of the power usage in every home would also benefit the society and help to educate the user on how to lower the carbon footprint.

The economical aspects go hand in hand with the environmental aspects as decreasing for example the power and water consumption will both affect the economics and the environment. By lowering the power consumption of a home the combined savings of the whole system of many thousands of homes would be noticeable, even though the system would require additional low-power devices in every home.

4.1.2 Ethical aspects

Because we are working with sensor data that might be sensitive for the user we have implemented security to prevent any data from being compromised. As we also work with user accounts special care has been taken to protect the users' passwords by never storing them in a humanly readable format. Furthermore we store a hash of each password instead of the actual passwords. This is very important because users often use the same passwords for multiple services and if the passwords were compromised in our system, other services that the user uses might be affected.

4.2 Further developments

As we took on a quite vast project that spanned across several systems from web development in Rails to socket implementation in Java, there are several things that we did not have time to implement. As we developed each part of the system as isolated parts the possibility of adding extra functionality in the form of new programs can easily be done without having to modify all parts of the program. In this section we present areas that have not explored yet and things we have not had time to investigate or implement yet. We will also discuss possible solutions to some of these future implementations.

4.2.1 Ability to control the home environment

The current implementation of our system only allows monitoring of data pushed from a home environment. Something that we would have liked to incorporate in our system is the ability to control devices in the home via the web interface as well. This would require modifications to the developed

web interface, as one must translate user driven request to actions. This would also require the implementation of a new program that is able to push a request to a device by communicating with a home gateway.

4.2.2 Key exchange

Even though we have implemented secure communication between a gateway (in our case the emulation of a gateway) and our system the key handling needed in a large system is not covered in this project. The implementation of a suitable mechanism for key sharing is necessary if we wish to scale up the solution. The current method of directly orally sharing a key will not be viable at a larger scale. Using other key sharing methods that can guarantee better security in large systems is necessary to be able to scale the system further. This could possibly be done using third party methods to share keys among users of the system.

4.2.3 SSL-encrypted MySQL connection

The connection to and from the MySQL server could be secured using SSL. This requires the use of keys and certificates, as well as adding a "SSL" requirement to the database profile for all users that connect to the database. The MySQL documentation pages provide a guide on how to set up SSL encrypted traffic to a MySQL server. [28]

4.2.4 Bulk data

One of the major bottlenecks in our system is that the system sends each sensor reading to the database individually. This is a bottleneck because the database is not able to handle that many insertions per second, therefore we propose that our ServerUDP program collects the data into bigger chunks before performing an update on the database. The implementation would then work as follows. Each thread takes a task out of the "Pool of Tasks" as usual, decrypts the data and adds the different values to some kind of data structure, which then is added to another synchronized list. When the list is full or if a certain time has passed, then the complete list is sent to the database. The thread that discovers that the list is full can send the data to the database or it could notify a separate thread that only handles this task. Figure 4-1 demonstrates how this method might work. This method of collecting the data into bigger chunks will greatly decrease the number of database insertions that our program has to perform.

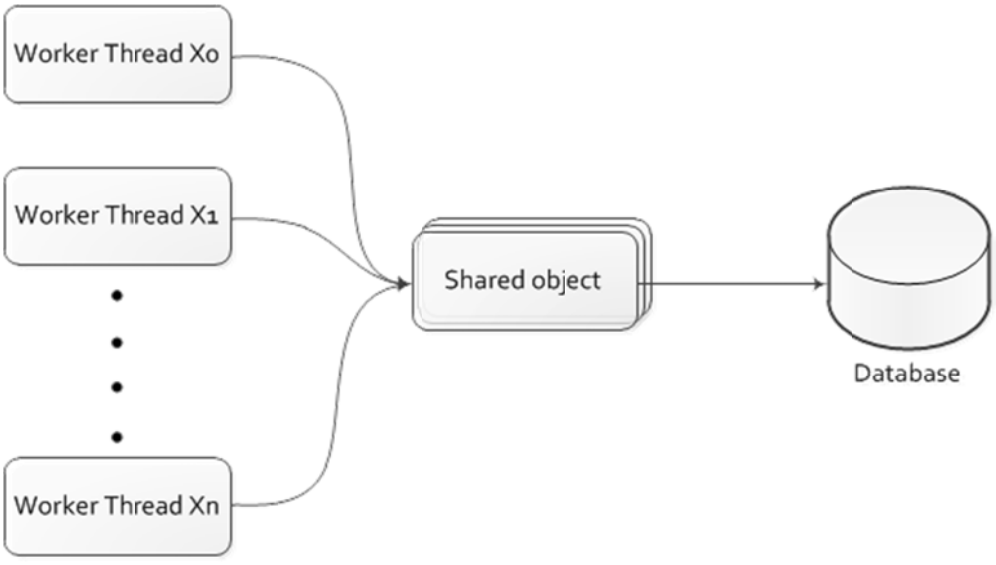

Figure 4-1 Possible bulk data extension structure

4.2.5 Working with real data

It was always our goal to include some real data in our system, i.e. connecting to the gateway that Albert López and Francisco Javier Sánchez built or to take some readings from the device that Abdel Ahmid worked on during his thesis project. However, due to time constraints this was never accomplished, but it would be easy for a future thesis project to take readings from these devices and send data to our system.

4.2.6 Faye optimization

The use of the asynchronous push messages greatly adds to the user's experience and accounts for a substantial part of the workload in developing this thesis project. However, the way Faye is implemented at the moment is somewhat inefficient. The whole operation of pushing to Faye is something that the ServerUDP program does with all the tasks that are coming in. This is not necessary since not all of the users will be looking at their graphs all the time, hence only those graphs that someone is currently looking at require the constant updates that Faye provides. A solution to this problem is to run another select query to check if a user is currently logged in, then based on this information decide if pushing to Faye is necessary. Another part that makes the Faye usage inefficient is the establishing of a connection from the SendUDP program to the Faye server. As of now we cannot keep a connection to Faye alive and have to establish a new connection for every task. Keeping the connection alive for a longer period is possible [25] and would probably significantly reduce the total time required for pushing new data to Faye.

4.2.7 HTTPS

HTTPS could relatively easily be added to the log in procedure on the web page. The advantage of this would be that the communication between the web server and the browser would be encrypted and secure. Because of this encryption, HTTPS will use more resources then a normal HTTP session; therefore it might be useful to use HTTPS for the login credentials and then normal HTTP for the rest of the web site.

The disadvantage of this would be that the rest of the communication is in the clear. The level of security is however always a compromise between speed and security, less security will require fewer resources and therefore be faster. A solution where all the traffic is sent with HTTPS is also possible but this approach will require more resources.

4.2.8 Switch from XML to JSON

JSON [29] has during the past years grown to be a more and more widely used standard. JSON has a slightly simpler syntax than XML and would require a smaller header in every message that is sent. This would make the communication more efficient. Switching from XML to JSON would be preferable in our case as the size of the UDP body is quite small, hence the percentage reduction in overhead would be larger.

References

[1] Prof. Gerald Q. Maguire, 'Examples of thesis projects'. [Online]. Available: http://web.it.kth.se/~maguire/maguire-exjobbs-examples.html. [Accessed: 18-April-2012].

[2] A. Ahmid, 'More than downloading', Bachelor's thesis, KTH, Royal Institute of Technology.

[3] World Wide Web Consortium, 'World Wide Web Consortium (W3C)'. [Online]. Available: http://www.w3.org/. [Accessed: 30-May-2012].

[4] iControl Networks inc, 'OpenHome home management', *Solutions, OpenHome*. [Online]. Available: http://www.icontrol.com/solutions/index.php. [Accessed: 30-May-2012].

[5] Z-Wave Alliance, 'Z-Wave solutions'. [Online]. Available: http://www.z-wave.com/modules/Z-WaveSolutions/. [Accessed: 30-May-2012].

[6] Albert López and Francisco Javier Sánchez, 'Exploiting Wireless Sensors', Master thesis, KTH, Royal Institute of Technology, Stockholm, Sweden, Not yet published.

[7] D. Heinemeier Hansson, 'Ruby on Rails'. [Online]. Available: http://rubyonrails.org. [Accessed: 10-April-2012].

[8] C. A. da Silva and J. Valim, *Devise*. plataformatec, Available at https://github.com/plataformatec/devise, [accessed April 9, 2012].

[9] M. Bleigh and J. Rosen, *Omniauth*. Intridea, Inc., Available at https://github.com/intridea/omniauth, [accessed April 9, 2012].

[10] C. Hale, 'bcrypt-ruby', 12-August-2009. [Online]. Available: http://bcrypt-ruby.rubyforge.org/. [Accessed: 18-May-2012].

[11] N. Provos and D. Mazières, 'A future-adaptive password scheme', in *Proceedings of the annual conference on USENIX Annual Technical Conference*, Berkeley, CA, USA, 1999, pp. 32–32, Available at http://dl.acm.org/citation.cfm?id=1268708.1268740.

[12] Google, 'Google Chart'. [Online]. Available: https://developers.google.com/chart/. [Accessed: 18-May-2012].

[13] Highsoft Solutions AS, 'Highcharts JS', *Highcharts JS*. [Online]. Available: http://www.highcharts.com/. [Accessed: 18-May-2012].

[14] J. Coglan, 'Faye'. [Online]. Available: http://faye.jcoglan.com/. [Accessed: 18-May-2012].

[15] Oracle, 'JDBC API Documentation'. Available at http://docs.oracle.com/javase/1.3/docs/guide/jdbc/index.html, [accessed April 27, 2012].

[16] Oracle, 'Java.net'. [Online]. Available: http://docs.oracle.com/javase/6/docs/api/java/net/package-summary.html. [Accessed: 18-May-2012].

[17] The Internet Assigned Numbers Authority (IANA), 'Service Name and Transport Protocol Port Number Registry', 07-May-2012. [Online]. Available: http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.txt. [Accessed: 08-May-2012].

[18] Wireshark Foundation, 'Wireshark'. [Online]. Available: http://www.wireshark.org/. [Accessed: 18-May-2012].

[19] 'Advanced Encryption Standard (AES)'. National Institute of Standards and Technology (NIST), 26-November-2001, Available at http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[20] M.-A. Cournoyer, 'Thin, A fast and very simple Ruby web server'. [Online]. Available: http://code.macournoyer.com/thin/. [Accessed: 18-May-2012].

[21] F. Cianfrocca, 'eventmachine'. [Online]. Available: http://rubyeventmachine.com/. [Accessed: 30-May-2012].

[22] M. Piacentini, 'SQLite Datebase Browser', 12-September-2009. .

[23] Tobias Ratschiller, Marc Delisle, Michal Čihař, Dieter Adriaenssens, Madhura Jayaratne, and Rouslan Placella, 'phpMyAdmin', 03-May-2012. [Online]. Available: http://www.phpmyadmin.net/home_page/index.php.

[24] The Institute of Electrical and Electronics Engineers, Inc, '802.11n'. IEEE, Available at http://standards.ieee.org/getieee802/download/802.11n-2009.pdf, [accessed May 28, 2012].

[25] Oracle, ‘Persistent Connections, HTTP keep alive’. Oracle, Available at http://docs.oracle.com/javase/6/docs/technotes/guides/net/http-keepalive.html, [accessed May 29, 2012].
[26] Oracle, ‘mysqlslap’. [Online]. Available: http://dev.mysql.com/doc/refman/5.6/en/mysqlslap.html. [Accessed: 18-May-2012].
[27] R. Bates, ‘RailsCasts’. [Online]. Available: http://railscasts.com/. [Accessed: 30-May-2012].
[28] Oracle, ‘Using SSL for Secure Connections’, *MySQL Documentation*. [Online]. Available: http://dev.mysql.com/doc/refman/5.6/en/secure-connections.html. [Accessed: 28-May-2012].
[29] D. Crockford, ‘json Media Type for JavaScript Object Notation (JSON)’. The Internet Engineering Task Force, Available at http://www.ietf.org/rfc/rfc4627, [accessed May 29, 2012].

Appendix A

A.1. Single message

```
<?xml version="1.0" encoding="UTF-8"?>
    <sensor_reading
            gateway_id="27"
            sensor_id="12"
            time="2012-05-24 16:45:31"
            type="Temperature"
            value="-8.911619"
    />
```

A.2. Multiple messages

```
<?xml version="1.0" encoding="UTF-8"?>
   <root[3]>
         <sensor_reading
               gateway_id="27"
               sensor_id="12"
               time="2012-05-24 16:45:49"
               type="Temperature"
               value="-9.571651"
         />

         <sensor_reading
               gateway_id="27"
               sensor_id="12"
               time="2012-05-24 16:45:58"
               type="Temperature"
               value="-10.101651"
         />
   </root>
```

[3] The name of the root element has no impact on the validity of the structure.

Appendix B

```
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.security.SecureRandom;
import java.util.Date;


public class UDPSendStressTester {

	public static void main(String[] args) throws Exception {
		String address = args[0];
		int UDPPacketSize = Integer.parseInt(args[1]);
		int iterations = Integer.parseInt(args[2]);
		float totalPacketSize = Integer.parseInt(args[3]);
		int socket = Integer.parseInt(args[4]);

		DatagramSocket clientSocket;
		clientSocket = new DatagramSocket();
		InetAddress IPAddress = InetAddress.getByName("localhost");
		SecureRandom random = SecureRandom.getInstance("SHA1PRNG");

		IPAddress = InetAddress.getByName(address);
		byte[] randomBytes = new byte[UDPPacketSize];
		random.nextBytes(randomBytes);

		long startTime = new Date().getTime();

		for(int i = 0; i < iterations; i++){
			DatagramPacket sendPacket = new DatagramPacket(randomBytes,
randomBytes.length, IPAddress, socket);
			clientSocket.send(sendPacket);
		}
		float floatDate = ( new Date().getTime() - startTime) / (float)
1000;
		System.out.println("Time to send: " + floatDate + "s");
		float throughput =  ((totalPacketSize / 10) / floatDate);
		System.out.println("Throughput: " + throughput + "Mbps");
		clientSocket.close();
	}
}
```

TRITA-ICT-EX-2012:084

www.kth.se