CHALMERS | UNIVERSITY OF GOTHENBURG

A modular system for smart energy plug stream analysis

Bachelor of Science Thesis in Computer Science and Engineering

JONAS GROTH
ERIK FORSBERG
JOHAN JINTON
IVAN TANNERUD
ISAK ERIKSSON
ANTON LUNDGREN

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, June 2016

Bachelor of Science Thesis

A modular system for smart energy plug stream analysis

JONAS GROTH
ERIK FORSBERG
JOHAN JINTON
IVAN TANNERUD
ISAK ERIKSSON
ANTON LUNDGREN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
University of Gothenburg

Göteborg, Sweden 2016

A modular system for smart energy plug stream analysis

JONAS GROTH
ERIK FORSBERG
JOHAN JINTON
IVAN TANNERUD
ISAK ERIKSSON
ANTON LUNDGREN

Examiner: Arne Linde

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
SE-412 96 Göteborg
Sweden
Telephone: +46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg 2016

A modular system for smart energy plug stream analysis

JONAS GROTH
ERIK FORSBERG
JOHAN JINTON
IVAN TANNERUD
ISAK ERIKSSON
ANTON LUNDGREN

Department of Computer Science and Engineering,
Chalmers University of Technology
University of Gothenburg

Abstract

This report documents the development of a system capable of gathering energy consumption data from multiple different brands of smart energy plugs. The problem today is that firstly the manufacturers' software is not general and provides a limited set of functions. For example, one brand's software may provide forecasting of energy consumption while another does not. Secondly, it is not possible to use different brands of plugs together.

The system presented in this report consists of a plug data parser, a message broker and a data processing engine. The plug data parser can gather data from one or multiple different brands of energy plugs at once. Using a message broker opens the possibility to gather data from a large number of plugs at the same time and in real-time. A data processing engine enables processing of the data through which use cases are implemented. It provides functions such as calculation of a moving average for the energy consumption, the total power consumption for all plugs and provides alerts for the energy consumption. Lastly, it provides a foundation to forecast future consumption.

The resulting system is capable of processing more than 500 plug readings per second in real time, from two different plug brands.

Keywords: smart energy plugs, energy plug, stream analysis, forecasting, statistics, alarms, big data

Sammanfattning

Denna rapport dokumenterar utvecklingen av ett system för att samla in energianvändningsdata från flera olika märken av smarta energipluggar. Problemet med detta idag är främst att tillverkarnas mjukvara ej är generell samt endast erbjuder ett begränsat antal funktioner. Till exempel kan ett märkes mjukvara erbjuda prognoser om framtida energianvändning medan ett annat inte gör det. Dessutom är det i dagsläget inte möjligt att använda smarta energipluggar av olika märken tillsammans.

Systemet som presenteras i denna rapport består av en “plug data parser”, en “message broker” samt en “data processing engine”. Parsern samlar data från ett eller flera märken samtidigt. Användningen av en message broker öppnar för möjligheten att samla data från ett stort antal pluggar samtidigt, i realtid. Med användandet av en data processing engine implementeras behandling av data som beräknar ett glidande medelvärde för energianvändningen, den totala energianvändningen för alla pluggar samt förser användaren med alarm vid onormal energianvändning. Slutligen utgör systemet en bas för att förutse framtida energianvändning.

Det resulterande systemet kan behandla mer än 500 energiavläsningar per sekund i realtid, från två olika märken av energipluggar.

Nyckelord: smarta-energipluggar, energipluggar, strömanalys, prognostisering, statistik, alarm, big data

Acknowledgements

We would like to thank our supervisors Vincenzo Gulisano and
Marina Papatriantafilou for their invaluable help with this bachelor thesis.

Contents

1 Introduction **1**
1.1 Purpose . . . 1
1.2 Scope . . . 2
1.3 Related work . . . 2
1.4 Method . . . 3

2 Problem **4**
2.1 Problem analysis . . . 4
2.1.1 Using different energy plug brands together . . . 4
2.1.2 Processing of streamed energy data . . . 5
2.1.3 Usages for streamed energy data . . . 5
2.2 Task specification . . . 6
2.2.1 Performance . . . 7
2.2.2 Fault tolerance . . . 7

3 Technical Background **8**
3.1 Smart Energy Plugs . . . 8
3.1.1 Plugwise . . . 8
3.1.2 Z-Wave . . . 9
3.2 Message Brokers . . . 9
3.2.1 Apache Kafka . . . 9
3.2.2 RabbitMQ . . . 10
3.3 Data processing engines . . . 10
3.3.1 Apache Storm . . . 10

3.4 Forecasting 11
3.4.1 Exponential Smoothing 11
3.5 Node.js 12

4 Feasibility study 13
4.1 Message broker 13
4.2 Data processing engine 13
4.3 Forecasting 14

5 Model of the system architecture 15
5.1 Gathering of energy plug data 16
5.2 Processing of energy plug data 16

6 Implementation of the system model 17
6.1 Acquiring data from energy plugs 18
6.1.1 Plugwise 18
6.1.2 Z-Wave 19
6.1.3 Other brands 20
6.2 Using streamed energy data 21
6.3 Building the stream processing architecture 21
6.3.1 Statistics 22
6.3.2 Alarms 23
6.3.3 Forecasting 24
6.4 Extracting data from the stream processing engine 24
6.4.1 User Interface 25
6.5 Fault-tolerant system 26

7 Testing and evaluation 27
7.1 Hardware and software setup 27
7.2 Tests 28
7.2.1 Latency through Kafka and Storm 28

7.2.2 Plug data parsing 29
7.2.3 Maximum number of plugs - processing engine 29
7.2.4 Maximum number of plugs - plug parser 30
7.2.5 Fault tolerance 30

8 Results 31

8.1 Plug data parser 31
8.2 Processing of energy data 31
8.3 Performance testing 32
8.3.1 Polling speed 32
8.3.2 Message broker and processing engine 35
8.4 Fault tolerance 36
8.5 User interface 36

9 Discussion 38

9.1 Method 38
9.2 Plug data parser 39
9.3 Message broker 40
9.4 Forecasting, statistics and alarms 40
9.4.1 Improving forecasting of future consumption 40
9.4.2 Efficiency when calculating a moving average 41
9.5 Performance evaluation 41
9.6 System improvements 42
9.7 Implications for society 43

10 Conclusion 45

A Plugwise protocol 50

Glossary

Apache Zookeeper	A program for maintaining configurations and synchronisation of distributed services
Baud rate	Specifies how fast data is sent over serial
Javascript	A programming language commonly used to create interactive web pages
JSON	JavaScript Object Notation, a standardised format for sending data
Mesh network	In a mesh network all nodes relay messages to other nodes, the most famous mesh network is the Internet
ODROID-XU4	A single board computer
OpenJDK	An open-source Java Development Kit that also contains a Java Runtime Environment
Smart energy plug	A device that reads electricity consumption from an electricity socket

Chapter 1

Introduction

Regardless of where in the western countries you live, it is of interest to lower the energy consumption, both from an economic and environmental standpoint. Households in the USA have an average annual electricity consumption of 11,000 kWh [1] while the corresponding number in Sweden is 14,000 kWh [2]. This is a substantial part of the expenses for a family. There are devices which are plugged in between an electricity outlet and some appliance, that measures the electricity consumption. These devices are called smart energy plugs and can help households get insight into what appliances consume the most electricity. This allows them to make informed decisions about their consumption [3].

The smart energy plugs come in different forms by different companies. These plugs often differ in the way they handle data and in their utilisation of communication protocols. Because of this, the different brands have their own software with different capabilities. Therefore, when buying new plugs, the software included might not give the users information about the plugs that they are interested in. This project aims to provide users with software that offers the same functionality no matter which plug brand is used. With a general system such as this, the user can buy energy plugs without having to base the purchase on software properties.

1.1 Purpose

The purpose of this project is to create a prototype of a general-purpose modular system for gathering and processing of electricity consumption data, from a number of different smart energy plugs. A data processing engine is to be used to process the data and present it to the user.

The prototype should be able to display statistics and forecasts describing the electricity consumption, as well as to send out alerts depending on the current consumption. There are other systems that provides some of these functionalities, but they are limited to one type of plug and one corresponding protocol. The resulting prototype of this project is meant to act as a general system that small-scale, as well as large-scale, users of energy plugs can utilise regardless of underlying hardware.

1.2 Scope

As mentioned, the project aims to create a program to consolidate data from several different energy plugs, including plugs of different brands. There are many different plugs available to choose from and implementing support for them all would be too time-consuming for the scope of the project. For that reason, we choose to work with two plug brands. These brands are chosen as they use different protocols to communicate, which is in line with what the project is aiming to achieve.

Since the goal of the project is to acquire and process data, a user interface is created only to showcase to what ends the system can be used.

1.3 Related work

The paper by Monacchi et al. [4] describes an integration of households into a so called smart grid. Smart grids mainly focus on large scale monitoring infrastructure for measuring of electricity load to optimise the production and consumption of electricity.

Already back in 2002 Tsuyoshi Ueno et al. [5] wrote a paper about having a system to measure and visualise a households energy consumption. They found that a reduction in energy consumption of 9% was achievable with their system. A similar study was made by Xudong Ma et al. [6] but they focused on measuring both temperature and electricity consumption to optimise the usage of heating, ventilation and air conditioning systems, HVAC. Both of these studies show that visualising energy consumption can decrease the energy consumption, which could be a consequence of this project.

An abstract framework for the Internet of things allowing users to implement their own algorithms without concern for how data is transferred is discussed in [7] by Kamburugamuve et al. They used the message broker Apache Kafka among others as well as Apache Storm in their framework.

1.4 Method

The methodology applied in the project consisted of two phases when adding a new component to the system. The first phase being research of the available technology to make sure that the properties fit project's desired functionality, with phase two being the actual implementation of the system component in question.

Meetings were held every week on which progress and problems were discussed and different tasks were handed out to be worked on over the coming week. To be able to develop the system in a parallel fashion, the *Git* version control system was used. This meant that everyone in the group always had access to every version of the code regardless of which component of the system was being worked on.

Chapter 2

Problem

The main goal of this project is to create a system to fetch, process and analyse data from the smart energy plugs. To find a suitable solution, the different parts of the problem need to be analysed. In this section, the problem will be broken down, analysed and turned into a specification. Answers will be given to the questions: What needs to be done? What are the pieces to the puzzle? How do they fit together?

2.1 Problem analysis

The goal is to create a system that offers the same functionality regardless of energy plug brand. As a result of this, the system must solve the problem of individual plug brands not always providing the functionality desired by the user. In addition, the system should be expandable so that users can add new features to the system. These factors create a number of more specific problems in need of solving.

2.1.1 Using different energy plug brands together

The difference in software mentioned previously often lies in the proprietary communication protocols of different brands. Also the content of the respective brands protocols may differ slightly. One may give current power consumption in kilo-watt and another in watt, for example. This presents a problem when trying to use all the different brands of plugs together and later use the acquired data in some calculations. The solution to this problem is to create a poller or plug data parser to extract data from the proprietary protocols. This data is then arranged into a new open format that can be easily used in other applications.

2.1.2 Processing of streamed energy data

Processing of streamed data makes for a few problems. The main problem that arises here is the sheer amount of the data, along with the fact that it is streamed and as such needs to be processed in real time. Considering that the project's system should be able to be used on a larger scale, for example monitoring the electricity consumption of a small town, the number of plugs can be in the thousands. Each plug will send out large amounts of data and with a large number of plugs, this data will need to be processed by a data processing engine for streamed data.

2.1.3 Usages for streamed energy data

To be able to show examples of what the system can achieve in terms of processing and handling of energy plug data, some use cases needs to be implemented. These are explained below.

Statistics
Overall statistics should be available to the user. The point of these statistics is to give a clear overview of the system while also allowing the user to go into detail about individual plugs. The total power that the system is using should be available, as well as the total consumption since system start up. Another feature that should be implemented is a moving average per plug over the last few hours.

Alarms
The system should send a notification of some kind when a plug is reading abnormal values. For example, a user should be able to receive a notification when the readings of a plug has been zero for a long time. This probably means that something is wrong with the device connected to the plug, either it is broken or the plug might have been pulled. Also, the user should be alerted when the total consumption of a plug system reaches a value higher than a certain multiple of the moving average.

Forecasting
Simple predictions reminiscent of forecasting should be available to the user, providing information of possible future values based on the past collected values. Forecasting can be done in a number of different ways, many of

which are outside the scope of this project. Accordingly a trade-off needs to be made between complicated solutions with accurate values, and simpler solutions with less accurate values.

2.2 Task specification

This section will discuss specific requirements placed on the system, and describe these in more detail. Once all the requirements described in the specification below are met by the system, the goal of the project is fulfilled.

- Data is gathered from multiple different brands of energy plugs i.e. the system should be hardware agnostic. The system shall be able to gather data from at least two different types of plug brands.
- The gathered data must be arranged into a standardised format and sent to a message broker in real-time.
- A setting for alerts shall be available to the user; notifications will be sent to the user based on predefined conditions.
- Real-time statistics ought to be calculated and presented to the user. This includes average power and energy consumption, for the hour for the system as a whole.
- Individual plug readings shall be available to the user in real-time.
- The system shall do calculations that could be used to create a forecast for the electricity consumption over the next hour.
- It shall be possible to add new plug brands to the project's system. For this reason, the energy plugs' data need to be parsed in a modular fashion. Also the parser shall parse plug data in real time even if there are a large number of plugs.
- The system shall not fetch data slower than the fastest brand of energy plug. More details can be found in section 2.2.1.
- The message broker and data processing engine shall be able to process 200 energy plug readings per second. Read about the reasoning behind this number in section 2.2.1.
- The system shall have ways of dealing with errors that can occur during service. Read more about this in section 2.2.2.

2.2.1 Performance

Difference in performance among the brands should not affect the system as a whole, meaning that the modules for the different brands have to work independently of each other. The number of different brands and total number of plugs the system should be able to handle is partly determined by the maximum number of plugs for each brand's system. The data processing part can receive data from multiple pollers at once, so if the problem of one plug brand reaching its maximum amount of plugs arises, it should be solved by simply adding another poller.

A possible user of the system could be someone reminiscent of a landlord. This means that the user might not only want to measure one household but many households together. With this example in mind it was decided that the system should be able to handle 10 energy plugs in 20 households which equates to 200 readings per second. Some users of the system might have specifications more demanding that of a landlord and this will function as point of comparison.

2.2.2 Fault tolerance

The system should be able to handle events such as plugs being disconnected or added to the system without suffering any downtime. Also the system has to handle corrupt messages without crashing. If the data processing engine crashes or experiences any problems, it should not impact the collection of consumption readings from the energy plugs. Messages sent from the plug data parser should be queued up and processed once the system is available again.

Chapter 3

Technical Background

The project consists of many different components with their own uses and limitations, both in terms of software and hardware. Below follows a brief description of all components.

3.1 Smart Energy Plugs

A smart energy plug is a small device that plugs into an ordinary power outlet, creating an outlet extension as the plug itself has an outlet. The plug measures the electricity consumption and current power usage of the outlet and sends data to an application, offering a way to keep track of energy needs [8].

3.1.1 Plugwise

Plugwise is a brand of energy plugs that creates a mesh network between a maximum of 64 plugs, these plugs are called ”Circle”, and a master plug called ”Circle+”. The ”Circle+” communicates with a USB-dongle, called ”Stick”, for the communication between all components a wireless communication protocol named ZigBee is used [9]. It is a high-level communication protocol mainly used to create personal area networks. The mentioned USB-dongle is used with a computer to communicate with the rest of the network. The dongle communicates with the computer via a serial interface with a baud rate of 115200 bits/s. The stick accepts and sends HEX encoded commands using a closed protocol [10].

3.1.2 Z-Wave

Z-Wave is a wireless technology that enables smart devices to communicate. It creates a mesh network between devices and a controller, each network can contain a maximum of 232 devices [11]. Adding new devices to a network is done by pressing a button on the device to be added and then waiting for the inclusion process to finish. There are more than 1400 Z-Wave certified products from more than 330 manufacturers [12]. The energy plugs utilising Z-Wave used in this project is manufactured by Greenwave Systems [13]. All Z-wave certified products are capable of communicating with each other. The certification ensures backwards compatibility and future proofing.

3.2 Message Brokers

A message broker is an intermediate manipulator of messages between a sender and a receiver [14]. A producer sends messages to the broker application, which then stores incoming messages in some way until they are requested by a consumer. A broker may perform some operation on the incoming message, for example preparing it for the receiver by reformatting the data, or routing the data to one or more destinations. It can be seen as a building block in a bigger scheme, used for connecting two dots of the scheme together.

3.2.1 Apache Kafka

Apache Kafka is an open-source messaging system designed for persistent messaging and high throughput. Kafka provides a distributed real-time system for publishing and subscribing to messages. The processes that publishes messages to Kafka are called producers and the processes subscribing to messages are called consumers. Messages are published to different topics which essentially act as categories [15]. Topics can be divided further into partitions which can be used to differentiate the messages within a topic. In order to receive the messages, the consumer can then subscribe to the topic and partition of interest. Kafka uses an application called Apache Zookeeper to keep track of it's synchronisation and configuration information. In conclusion Kafka is a message broker, providing a way to send a large number of messages from several producers to several consumers.

3.2.2 RabbitMQ

RabbitMQ is an open-source message broker software that implements the Advanced Message Queueing Protocol (AMQP) [16]. The features provided by the AMQP, e.g. message orientation, queueing, routing, reliability and security are important when working with message brokers [17].

3.3 Data processing engines

Data processing engines are systems designed for processing of big data. Such systems are suitable when handling large amounts of data and/or streaming data. There are a number of engines available to process streamed data such as S4, Storm, and Flume [18][19][20].

3.3.1 Apache Storm

Apache Storm is a free data processing framework that is open-source. It provides a data processing engine that processes unbounded streams of data in real-time. One of the main advantages with using Storm is that it can be used with any programming language, making it simple for most programmers to use. Storm uses, similarly to Kafka, Apache Zookeeper for maintaining configurations and distributed synchronisations. Applications in Storm are designed as a topology, with bolts and spouts as seen in Fig. 3.1. The spouts act as sources of data streams, they can emit messages from any type of message brokers or get data from other sources. Bolts are where all processing happens, they receive data from spouts or other bolts and do some processing and can then emit the result to another bolt or somewhere else [19]. The same bolt can be run in several different instances at once opening for possibilities for doing parallel processing.

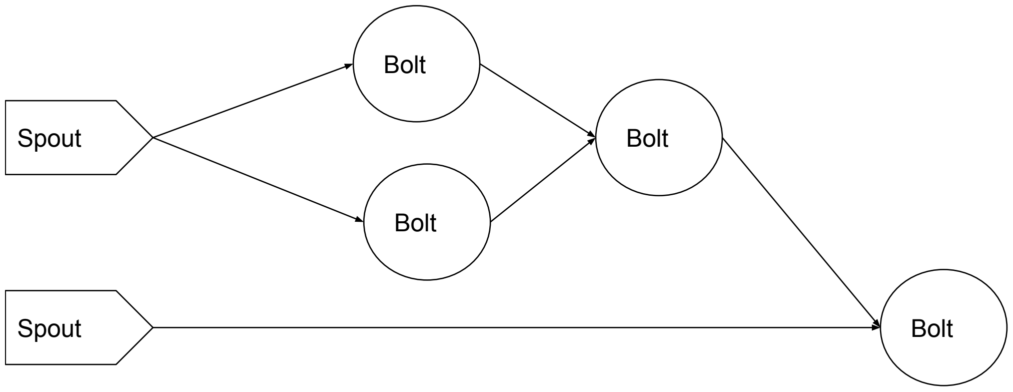


Figure 3.1: An example of a Storm topology, with spouts and bolts.

3.4 Forecasting

As stated in 2.2 the system developed in this thesis should do calculations that could be used to create a forecast. Forecasting makes use of historical or current data to predict future scenarios and trends. This can be done in several ways with different suitability depending on what is to be forecasted. One type of forecasting consists of predicting future values by analysing previous values. Predictions about data where historical values are unavailable can also be made, but instead by observing previous data found in other areas. For example, upcoming electricity demand can be predicted by taking population, time and electricity pricing into consideration.

3.4.1 Exponential Smoothing

Exponential smoothing is a simple method used to create approximate forecasts. More advanced forms of exponential smoothing, taking trends and seasonality into account, have been used with good results in [21]. A basic form of the method forecasts its values as

$$s_0 = x_0$$

$$s_t = \alpha x_t + (1 - \alpha) s_{t-1},$$

where s_t is the forecast for x_{t+1} and α is the smoothing factor which determines the effect older historical values will have on the forecast [22]. Essentially this is a weighted average where the impact of a value on the final result decreases exponentially with the value's age.

3.5 Node.js

Node.js is a runtime environment for developing cross platform applications written in JavaScript. Node.js is also event driven which makes it easy to create real-time applications [23]. Another advantage is that there are a lot of premade libraries for handling everything from serial port communication to interfacing with the web. These libraries are available through a package manager, called *npm* [24].

Chapter 4

Feasibility study

Message brokers, data processing engines and forecasting algorithms all come in different versions with varying advantages. In order to make informed decisions about what best fits the system it is important to research these areas.

4.1 Message broker

The message broker is an important part of the system. Different message brokers had to be evaluated in order to find a message broker that caters to the system's needs. Apache Kafka and RabbitMQ were the two main candidates. One difference between the two is that RabbitMQ is a message queueing system while Kafka gathers the data in a non sorted fashion [25]. This is however a nonissue as the data sent in this project's system contains a time-stamp, as explained in section 6.1. Kafka was designed specifically to solve the problem of having messages in large quantities. It has been used by many different companies such as LinkedIn, Netflix and Spotify [26]. RabbitMQ on the other hand, is an older message broker and wasn't designed for high throughput-volumes [25]. This makes Kafka the better choice as the project is looking to make a system appropriate for large-scale use.

4.2 Data processing engine

To realise the goal of the project, the system needs a data processing engine that is scalable while providing high availability. A popular option is Apache Storm, which is used by companies such as Spotify, Baidu and Alibaba [27]. The Storm engine handles fault tolerance by guaranteeing processing of tuples while also restarting dying processes [28]. With these properties the system can remain available while encountering errors. Storm tackles the issue of system scale by offering tools to alter parallelism of processes [29]. This allows the system to become more parallel with heavier loads of data without disrupting the system. Since these are issues of interest to the project, it was decided to use Storm. Both Kafka and Storm have also been used previously for collection and processing of data from air quality sensors in [30], this further cements the choice to use these tools.

4.3 Forecasting

The use cases in general and the forecasting specifically is not the main focus of the project. There are many different algorithms with different suitability for different situations [21]. The scope of this thesis is only to showcase the possibility to create a forecast for future consumption. As such the accuracy or suitability of this algorithm has not been taken into account in the selection process. Exponential smoothing was therefore chosen merely as an example of a forecasting like algorithm.

Chapter 5

Model of the system architecture

There are many ways of making energy plug data available to the user while also providing processing of the transferred data. The system architecture of this project is only one of many solutions. This section describes a model that can be used to interconnect the various parts of the system with respect to the problem specification in section 2.2. In Fig. 5.1 the system model and the flow of data can be seen in full.

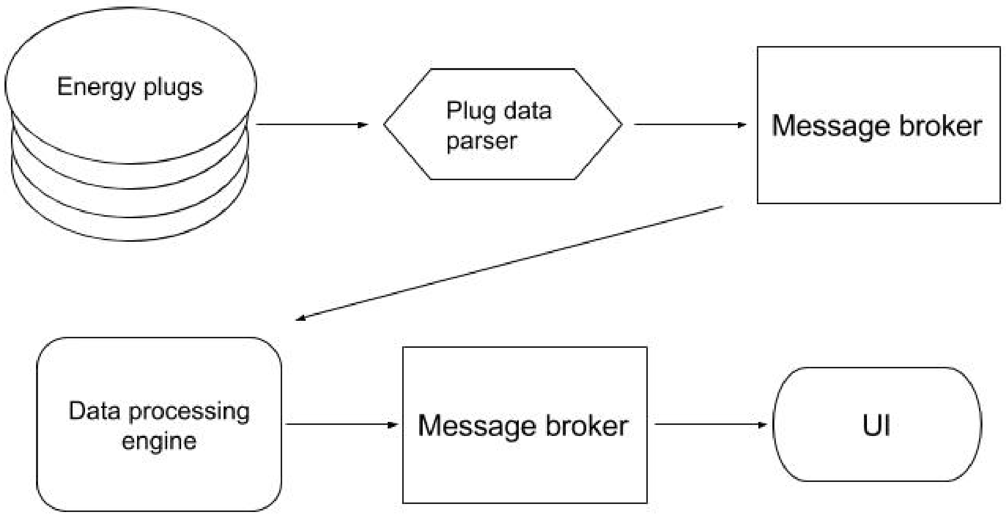


Figure 5.1: The system model displayed in its entirety.

This model contains a plug data parser, explained in the section below, so that the data from different energy plugs is sent to the message broker in the same format. By the use of a message broker as an intermediary entity the system can transmit data from different energy plugs into the data processing engine. This way the same processing can be applied to all data sent through the system while also providing a temporary storage location, in the form of a message broker, for the results. The UI is in the model as an example of showing where a consumer can access the processed data.

5.1 Gathering of energy plug data

Data from different plugs need to be collected and consolidated in order to be utilised in the use cases. This is the purpose of the plug data parser, which will be further divided into modules that specifically handles each brand and gathers all the acquired data. The plug data parser will also pull the data readings from the plugs as they do not send the data without requests. After the data has been acquired it will be sent onwards to a specific topic on the message broker in the system, giving consumers centralised access to the data readings.

5.2 Processing of energy plug data

To handle the potentially large amounts of data produced by the energy plugs, the system needs an efficient model for processing streamed data. The use of a message broker in conjunction with a data processing engine gives the system capability to cope with the problem of processing streamed data in real-time. This type of architecture also makes the system scalable as the message broker and stream processing engine can be deployed on several machines to increase performance. The data gathered from the energy plugs is consumed by the data processing engine and is processed according to the specifications. Subsequently the processed data is sent to a message broker so that it is accessible by end users.

Chapter 6

Implementation of the system model

This chapter covers how the the system model was implemented as well as the underlying reasons for the approach. See Fig. 6.1 for an overview of the solution, with the inner components of the message broker and data processing engine displayed.

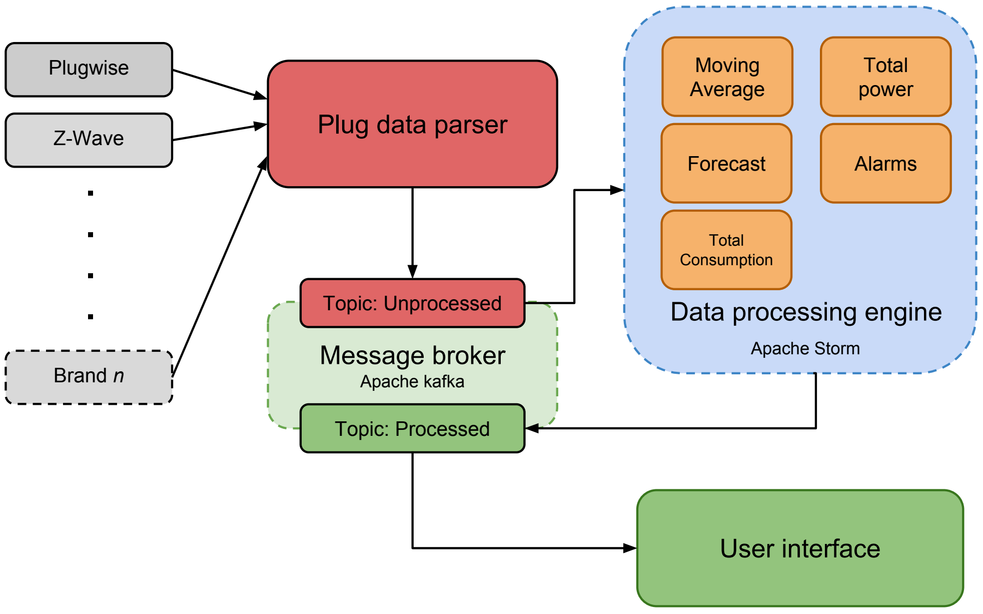


Figure 6.1: System overview

6.1 Acquiring data from energy plugs

The data that the different plug system receive from their plugs is formatted in different ways. Therefore a plug data parser is needed, in order to convert the data into a format recognisable by the data processing framework. The parser was split up into one main module and one sub module for each of the plug brands. Programming of the plug data parser was done in JavaScript together with Node.js, as it is a language that is well suited to creating real-time applications. These sub modules work independently from the rest of the system, continuously collecting data. The main module then compiles the data, and passes it on to the message broker.

There are common denominators in every plug system, namely the power output and the electricity consumption. These two components, along with a timestamp, are used to compose a standardised data object. For this, JSON-objects (JavaScript Object Notation) was chosen since there is support for handling the JSON format in both JavaScript and Java, which was used when implementing the data processing. See table 6.1 for an overview of the JSON-object used to transfer data from the energy plugs.

Table 6.1: JSON object used to carry data inside the system

Variable name	**Explanation**
timeStamp	Timestamp in ms for when data was read from plug
power	Current power reading
energy	Energy in kWh since last reading
plug_id	A unique id for the sending plug

The sub modules communicate with the main module via event emitters, emitting events once per second with the current data from the plugs. This design means that most of the processing is done in the sub modules leaving the main module mostly idle. The data sent off to the message broker needs to be formatted in a standardised way as seen in table 6.1, otherwise the data will be thrown away instead of being processed once it reaches the processing engine.

6.1.1 Plugwise

Plugwise's serial protocol is not open, meaning it needed to be reverse engineered in order to acquire the data. To be able to implement the module for requesting data from the Plugwise network, an extensive documentation of the Plugwise's serial protocol [10] was used. Communication with the Plugwise network is done with the help of the Plugwise USB-stick, which communicates with a computer through serial communication using a baudrate of 115200 bit/s. The module sends a request to one plug, wait for a response and then proceed with another plug. To send a request, the program generates a command string according to the protocol specified in table A.1. It then sends this string to the Plugwise stick's serial port, which responds. Once the response is received, it is parsed according to the protocol in table A.2. In Fig. 6.2 are two examples, with brief explanations, of strings of the format specified in appendix A.

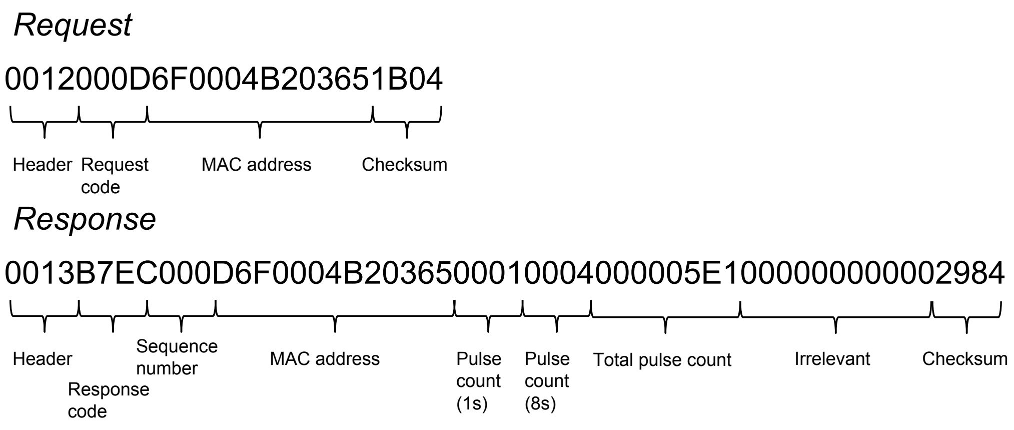


Figure 6.2: An example of a request and response string from the Plugwise protocol.

Data in the response contains both the current power and the total energy consumption. In order to get the energy consumption for the last second, the newest value is subtracted from the previous one. These values can then be sent onwards to the main module.

6.1.2 Z-Wave

The Z-Wave protocol is not open either but it is much more widely used. Notably there is an open-source project for a library to interface with Z-Wave devices called OpenZWave [31], or OZW for short. The library contains a wrapper available for Node.js, which was used to create the data plug parser.

To interact with Z-Wave devices some device to communicate on the same wireless protocol is needed. For this a Z-Wave.me UZB stick [32] has been used, which is compatible with the OpenZwave library. Polling data from Z-Wave device with OpenZwave can be done by setting a poller interval for each plug as well as a callback function. The callback function is there to receive the data and emit events to the main module.

To differentiate the different kinds of data, Z-Wave uses something called "Command Classes" which identifies the data with hexadecimal value. The relevant command classes is detailed in table 6.2.

Table 6.2: Z-Wave command classes

Name	**HEX**
COMMAND_CLASS_METER	0x32
COMMAND_CLASS_SWITCH_BINARY	0x25

Each command class contains a number of different values, such as:

```
Energy=5.7265
Previous Reading=5.7265
Interval=1
Power=7.9
Previous Reading=7.3
Interval=1
Exporting=false
Reset=undefined
```

As seen previously the command class contains different types of data, most notable is the "Energy" and "Power" values as these are the ones being processed in the project's system. All the module has to do is to put the power and energy readings into a JSON string and send it off to the main module.

6.1.3 Other brands

Other than Z-Wave and Plugwise there are some other brands of smart energy plugs. The other ones considered in this project all use a proprietary protocol and communicate either over Wi-Fi or Bluetooth. Both would require extra hardware to communicate with the system and a significant effort to reverse engineer their respective protocols. The project's time frame did not allow us to work with any of these plugs.

6.2 Using streamed energy data

The data gathered from the energy plugs is sent from the main module of the plug data parser to the Apache Kafka message broker. There is one common topic that all data from the energy plugs is sent to. To be able to process the data, Kafka is integrated with the processing engine Apache Storm. This is done by using a Spout in Storm that emits data from the mentioned Kafka topic in real-time. Kafka and Storm are both installed on an ODROID-XU4 which is used as a server. This small computer is used merely to show that the system can be implemented on a small and energy efficient device. It does not mean that system has to be implemented using this type of computer.

The data sent from the plug data parser to the message broker consists of a JSON-String which can be processed by bolts in the Storm topology. Bolts were implemented for each use case mentioned in 2.1.3. This means one bolt for each type of statistic as well as for the alarms and forecasting. These bolts are described in detail below.

6.3 Building the stream processing architecture

Some processing is needed for several use cases. In order to avoid doing the same calculations twice, this kind of processing can be moved to a separate bolt. For example, the moving average for the energy consumption is used in both forecasting and alarms, as well as being a statistic itself. Several Storm bolts can be connected, so that a value calculated by one bolt can be sent to multiple different bolts. See Fig. 6.3 for a sketch of the used Storm topology.

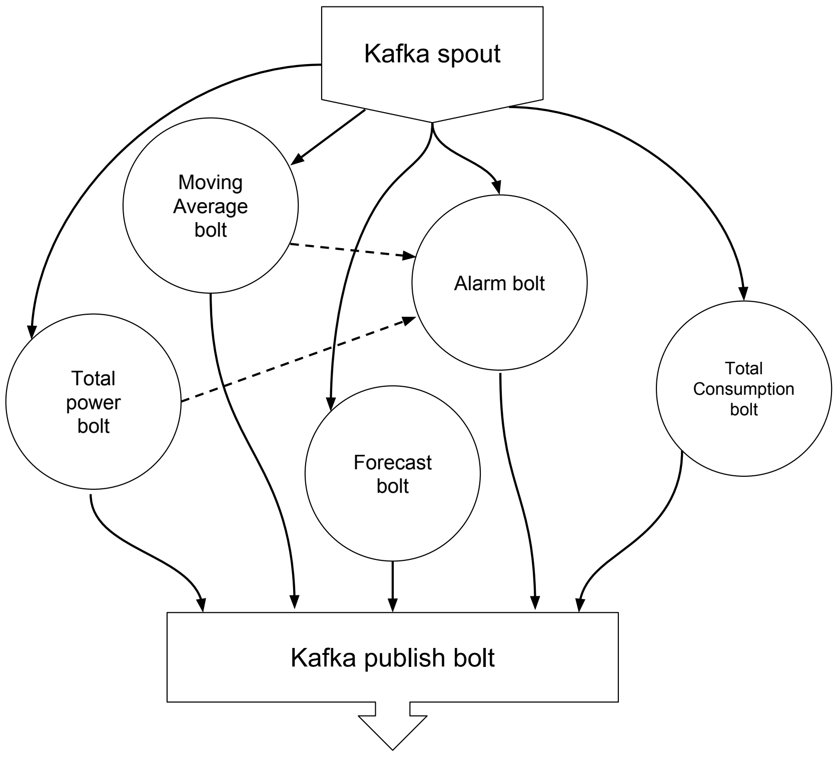


Figure 6.3: The systems stream processing topology.

All coding related to Storm was done in Java as this is a language that all members of the project have prior experience with. The Storm topology consists of a spout and multiple bolts, which process and prepare the data for the use cases. In the case of statistics, one bolt per statistic was implemented. The Storm spout emits the data collected from the Kafka message broker, and each bolt that uses the the data will receive it. Each time a message is received by the bolt, it will trigger a function in the bolt that will take care of the data in a desired way. This function is implemented differently in each bolt, depending on what use case the bolt is used for. The different bolts will be described in detail below for each use case.

6.3.1 Statistics

There are several types of statistics that the system produces. These calculations are made in the different bolts that exists within the topology seen in Fig. 6.3.

The first statistic is the moving average per plug. This will indicate how much each plug has been reading on average over the last arbitrary time frame n. The basic theory behind a moving average is that there is a sliding window covering the last n minutes, in other words a list in which the values of the last n minutes are stored. Continuously taking an average of all these values is the equivalent of a moving average. Every time a new value is received, the older stored values are checked to see if there are any values that are too old. Any values that are too old are removed and the new value is stored in the list. Also the total count and sum of the values are stored in order to avoid having to iterate through the whole list every time. This way the new average can be calculated with the formula below.

$Average_{new} = Average_{old} + \frac{Value_m}{n} - \frac{Value_{m-n}}{n}$

Another statistic is the current total power in watts that the system is reading. The bolt that calculates the total power takes the most recent value from each energy plug and adds these values to a total sum. When this new value is received and added, the old value from that energy plug is removed from the sum and a new total sum is calculated. The result of the calculation is emitted from the bolt once every second. As this bolt keeps track of all the current power readings from the plugs, it will also forward these values directly. The individual plug data can then be printed in the user interface to give an overview of the current readings from each energy plug.

The last statistic is the total energy consumed since system start up. When an energy plug reads a value, it also calculates the energy consumed since it's last reading. This difference in energy used is added to a total sum. The total sum is then sent to the user interface and is displayed as the energy used in kWh.

6.3.2 Alarms

There are two cases for when an alarm should fire.

1. The current average power consumption is more than 50% than the moving average usage during the last hour.
2. One or more plugs has zero power consumption for more than 10 seconds.

For the first case the moving average and total power consumption are obtained through other bolts. The difference in percent between the two values is calculated and if the calculated percentage is above the threshold an alarm is triggered. For the second case a list of times each plug last read a non zero value is maintained. This list is continuously checked to see if it has been more than 10 seconds since the last non zero reading.

When an alarm is triggered a message is sent back to the message broker and then to the user interface to alert the user.

6.3.3 Forecasting

The method presented here does in no way produce a trustworthy prediction or forecast. The aim is only to show that the system can be used for predicting future energy consumption, which is why exponential smoothing is used to do this. When using exponential smoothing one has to decide how much historical data to use and choose a suitable α value. In this project, energy data for the last four hours are used along with an α value of 0.8. These values are then used to indicate the total electricity consumption in the coming hour. The first thing that had to be done in the implementation, was to calculate the sum of all total energy consumption readings during the first, second, third and fourth last hours. This was accomplished through the usage of four sliding windows. To get the first hours readings, values between 0 and 1 hour old were summarised. For the second hour values between 1 and 2 hours old were used and so on. The system essentially maintains four sums which are then used with the formula detailed in section 3.4.1.

In the formula, s_4 is the forecasted consumption for the next hour. This formula was applied to the data that the bolt received and the forecast value, s_4, was then passed on to the user interface.

6.4 Extracting data from the stream processing engine

All processing bolts in the Storm topology produce some sort of output that needs to be sent back to the message broker in order to make it easily accessible. To avoid doing this operation in every bolt, the choice was made to create a separate publishing bolt to offload all the other bolts of this task.

An advantage of doing this is that this final bolt can be parallelised without the need to do any changes to cope with the concurrency. There is some added overhead to this in comparison to doing the sending to message broker operation in each and every bolt. But the fact that it can be parallelised without doing any major changes to the other bolts compensates for the added overhead.

The resulting data from the processing is sent back to the message broker on a second topic via the publishing bolt. This result topic is split into several partitions, one for each calculating bolt. The publishing bolt decides which partition on the topic the data should be sent to, depending on which bolt the data came from. There is one partition for each bolt type. The data sent back to the message broker is formatted as a JSON object with the syntax seen in table 6.3.

Table 6.3: JSON object used for results from data processing

Variable name	**Explanation**
timeStamp	Timestamp in ms for when data was read from plug
value	Processed result from calculating bolt.
plug_id	(optional, not used in most bolts) ID of the plug.

The timestamp from the incoming message that triggered the bolt is kept. This is mainly done to calculate the delay in the system.

Using the message broker as both a data inlet and outlet of Storm, makes it possible to capture both unprocessed and processed data by consuming messages from any of the two topics on the message broker. The unprocessed data from the plug parser can for example be used for other tasks and calculations. The processed data can be saved to a database or be used by any number of other applications. In this case, the processed data is sent on to a user interface.

6.4.1 User Interface

The processed data has to be made available to the user in some way in order to show the results of the processing. A local web page showing statistics, see Fig. 8.4, was set up to be the interface for a user of the system. Current forecasted values as well as a graph describing the electricity and energy consumption is on display. A local web server is used to host the web page while also pulling the statistical data from the message broker so that it can be used for the web page. The web server was implemented using the Express library [33] to deliver HTML and data consumed from the message broker. The graphs are made with a library called SmoothieCharts.js [34], the rest of the web page is made up of standard HTML and JavaScript.

6.5 Fault-tolerant system

As outlined in section 2.2.2 the system should handle unexpected events such as plugs being disconnected and messages being corrupted. Handling problems with plugs being unexpectedly disconnected is a fairly straightforward process. If there is no response from the transceiver, the connection is reset to start over again. Note that for Z-Wave all of this is handled automatically by the OpenZwave library itself.

As for problems related to corrupt messages, Kafka can not ensure that the messages it receives are not corrupted. However, the plug data parser throws away corrupt data it reads from the plugs. This way the message broker in use only needs to look for corruption in the data it stores. Kafka does this and guarantees that the messages available to consumers are without corruption. While a crash will not result in corrupt messages, it can still result in lost messages if Kafka has not yet written the data to disk [35]. Storm can prevent messages from being lost though. When a crash happens it can replay the tuples that had not been processed prior to the crash [36]. This is also a way of avoiding corrupt messages if the data processing framework crashes.

Chapter 7

Testing and evaluation

During and after the development of the system, different parts and the system as a whole were tested and evaluated. This was done to assert that the system is up to par with the specifications. The projects goals are met when the system meets all the criteria described in section 2.2. The results are of course highly dependent on the hardware used for the tests. To give a more balanced view of the performance of the system, tests were carried out on two separate and very different setups.

7.1 Hardware and software setup

The hardware used during testing was the single board computer ODROID-XU4 and the plugs as well as their respective receivers, detailed in table 7.1. The plug systems receivers were connected to the ODROID running Ubuntu 15.04. Software wise the setup included the plug data parser and the software listed in table 7.2.

Table 7.1: Hardware for the plug systems used during testing

Name	**Quantity**
Z-Wave.me UZB	1
Greenwave Systems PowerNode (Z-Wave)	1
Plugwise Stick	1
Plugwise Circle	10
Plugwise Circle+	1

Table 7.2: Software used while testing

Software name	**Version**
Apache Storm	0.9.5
Apache Kafka	2.11-0.9.0.1
Apache Zookeeper	3.4.6
Oracle Java	1.8.0_91

For reference purposes, a laptop with 8 GB ram and an Intel i5 2.7 GHz CPU with four cores is used. The test results from the ODROID can then be compared to the results from running the system on a more powerful computer.

7.2 Tests

To assess the system and determine its performance and characteristics, three different tests were developed and conducted. It is imperative that the system reads the correct values from the energy plugs, thus this was tested. Both the processing engine and the plug data parser were individually tested to determine which part of the system might pose a bottleneck, in terms of the maximum amount of plugs. If the processing engine had turned out to be the bottleneck that would have limited the whole system. The plug data parser on the other hand can not bottleneck the whole system, as increasing the number of plugs can be achieved by adding yet another parser.

The conducted tests are described in detail below, and results are available in chapter 8. All tests were conducted with the setups described in section 7.1 on both the ODROID computer and the mentioned laptop.

7.2.1 Latency through Kafka and Storm

In order to call the system a real-time system, the processing time through Kafka and Storm has to be equal to or lower than the rate at which the plugs are polled. If the latency is higher, that would result in congestion at the message broker. Some measurements on latency are provided by Storm itself but in order to get accurate data for Kafka and Storm together some tests had to be conducted. A script that sent randomly generated data to Kafka and then collected the processed data from Storm was created. This was then used to take two different types of measurements. The first used timestamps from the parser and compared them to the ones given after calculations in Storm was finished, estimating the total time this journey took. The second used timestamps internal to the script, again measuring the total round-trip time. The second measurement however, unlike the first, takes the total time from plug reading to user interface into account. This type of measurement is also called end-to-end latency.

7.2.2 Plug data parsing

It is essential to the system that the plug parser returns the correct data. To assure that the data is correct the original software from the manufacturer was used to take measurements. This was then compared to measurements taken with the systems plug data parser.

7.2.3 Maximum number of plugs - processing engine

One of the things to be tested is the maximum amount of data the system can handle. There are two factors to take into consideration when determining the total number of plugs the system can handle at once. Testing the processing part is done through writing a program that sends random values to the system. With the program it's possible to generate random data at a given interval. There might be differences between the number of messages the message broker and the processing engine can handle. The message broker may keep accepting new messages and simply enqueue them waiting for the processing engine to get ready.

A limit on the maximum latency was placed and the speed of the random data generator was successively increased. If the total latency does not stabilise around a single value, it is most likely a situation where messages are queued in Kafka at a higher rate than they are consumed. That would indicate that the rate of incoming readings are too high for the system to handle, which in turn suggests that the number of plugs are too high. This should happen if any latency within Storm reaches more than one second. Looking at the latencies in Storm will give a more precise indication of when the system has reached the peak of what it can handle.

7.2.4 Maximum number of plugs - plug parser

Testing the limits of the plug data parser is more complex, as this part is totally dependent on the hardware. To thoroughly test this, a large number of plugs would be needed, a number far beyond the scope of this project. In order to test performance data, the parser measurements for a small number of plugs was extrapolated to get indicative data for a larger number of plugs.

7.2.5 Fault tolerance

Detailed in section 2.2.2 are different scenarios that the system should be able to handle. Below are description of tests to test that the fault tolerance goals are met.

1. Disconnecting and reconnecting one plug.
2. Manually sending messages that are not of the correct format.
3. Manually shutting down the data processing engine.

To meet the project goals the system should not crash for any of the three scenarios. For the third scenario messages should be queued and saved at the message broker ready to be processed when the data processing engine comes back online.

Chapter 8

Results

The goal of this project was to create a system capable of reading and processing electricity data from a number of different smart energy plugs (see Fig. 6.1 for an overview). The system of this project can handle two brands of energy plugs namely Plugwise and Z-Wave. Calculation of a moving average, a forecast and total power consumption for all plugs have been implemented and is functioning. Alarms are triggered whenever a plug reads zero for more than 10 seconds, and when the total power consumption peaks 50% above the moving average of the system. One goal was also to parse the energy plug data in a general way to allow for easy integration of more plug systems. In line with this, the system allows plug systems to be added in two ways; either through the creation of a Node.js modules, or by simply sending JSON strings to the message broker.

8.1 Plug data parser

The energy plugs were tested with their respective software that came with each brand of plugs. Readings from these applications were then compared to the readings from the plug data parser. Running the same appliances connected to the plugs with the bundled software and the plug data parser yielded the same power readings.

8.2 Processing of energy data

In section 2.1.3 three different use cases are described, these are all implemented using Storm. An overview of the topology used is available in Fig. 6.3. Getting readings from the plug data parser to Storm are done through the use of a message broker, in this case Kafka. The system makes use of the bolts listed below.

- Calculation of a moving average power consumption per plug for the last hour.
- Total current power consumption for all energy plugs.
- Total energy used for the system since start up.
- Sending out alarms when a plug reads zero watts for 10 seconds or the current average power consumption is more than 50% above the moving average.
- Exponential smoothing on the last four hours readings, could be used to predict future values.
- Sending messages to Kafka.

The results from these bolts are sent to Kafka to be consumed. A simple consumer then publishes the data on a web server to make it available for the user interface.

8.3 Performance testing

The testing methodology, hardware and software setup is described in chapter 7. All tests were done using the described methodology and setup. In this section the result from the conducted tests are presented.

8.3.1 Polling speed

Thoroughly testing the polling speed for a different number of plugs would need access to a lot of plugs, which was not the case in this project. Measurements have been taken on a smaller number of plugs and have then been extrapolated. Below are details on the exact number of plugs used in tests along with the extrapolated results. It is important to note that both Plugwise and Z-Wave are mesh networks. A message may have to travel through a large number of plugs before ending up at the receiver, which would yield a higher delay than direct communication with the receiver. This could potentially cause a significant delay, with a maximum of 65 plugs in a Plugwise network 65 hops would be required in the worst case. All testing has been conducted with all plugs within range of the receiver.

For Plugwise a theoretical maximum polling speed based on some observations can be calculated. These observations are that Plugwise uses serial communication with a baud rate of 115200 bit/s and that each request is 120 bits long. Also, each acknowledgement response is 64 bits and each data response is 224 bits. This would lead to a theoretical max speed of $\frac{115200}{120+64+224} \approx 282 \quad polls/s$ while only considering transfer speed. However, in reality, the maximum polling speed differs from this theoretical value. The test results below aim to show the actual, non theoretical, polling speed.

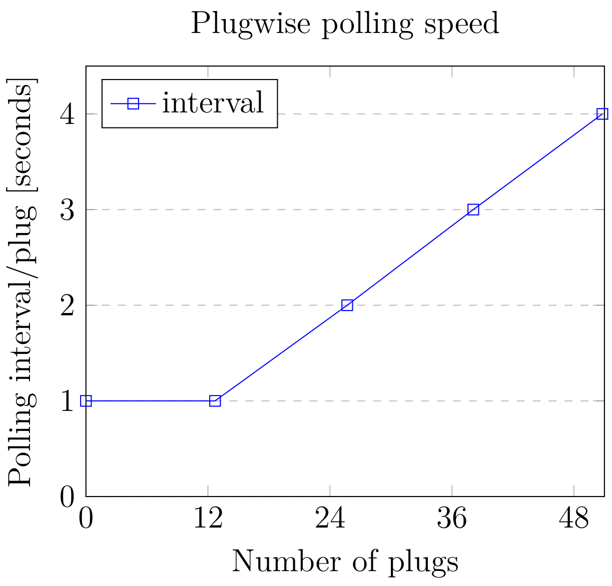


Figure 8.1: Graph of how often one plug is polled depending on the total number of plugs in the system.

The test described in section 7.2.4 show that a maximum of 12.7 requests per second can be processed. This means that the processing speed is the limitation rather than the transfer speed for the transceiver. On the other hand, the plug performs a new energy reading every second so polling the data more often than once a second is unnecessary. That is why the polling speed is the same for one through 12 plugs, as seen in Fig. 8.1. For a higher number of plugs the polling speed increases and the time between polls for some plug increases. Note that in the real tests only ten plugs were used.

Like Plugwise, Z-Wave relies on serial communication with a baud rate of 115200 bit/s through a USB transceiver. Z-Wave does however not rely on a single master node for communicating with the rest of the network. Instead the USB transceiver acts as a master node and can communicate directly with the network. The maximum number of nodes that can be in any Z-Wave network is 232 so the maximum numbers that can be polled per second is definitely not higher than this.

Again the tests described in section 7.2.4 show that the minimum polling interval that can be used is around 80 ms. It is possible to set a smaller interval than this in the software but that doesn't impact the real polling interval. As with Plugwise the readings on the plug only change about once a second hence there are no point in polling any single plug more than once a second. Given an 80 ms interval the maximum number of polls per second would be $\frac{1000}{80} = 12.5 \quad polls/s$. The polling speed dependent on the number of plugs can be seen in Fig. 8.2.

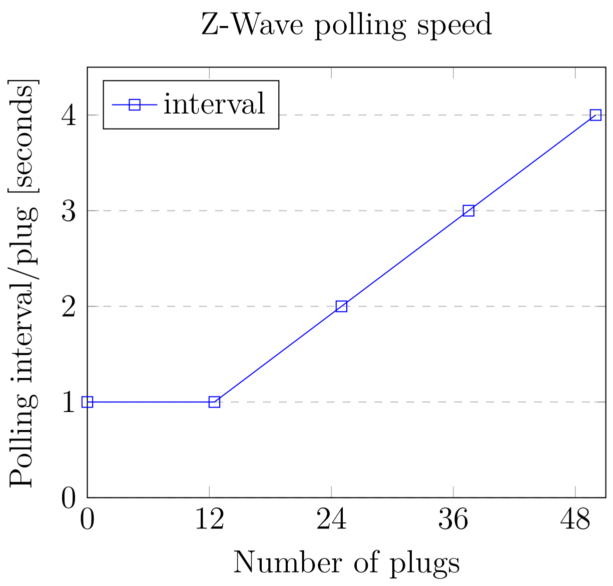


Figure 8.2: Graph of how often one plug is polled depending on the total number of plugs in the system.

This is very similar to that of Plugwise, described earlier, and is discussed in section 9.2.

8.3.2 Message broker and processing engine

The maximum number of plugs the message broker and processing framework can handle when running on the ODROID-XU4 was tested as described in section 7.2.3. Total latency through Kafka and Storm was measured using the method described in section 7.2.1. To start with, ten readings per second were used to get a base reading. The number of plug readings per second was then successively increased in steps of five, with 300 readings taken from each step. To get a fair reading the system was left to run for about one minute before the 300 samples were taken. The average of these samples are displayed in Fig. 8.3.

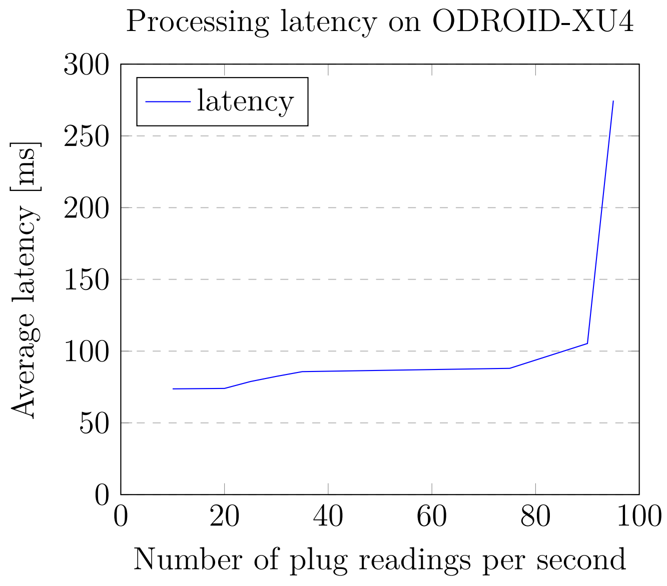


Figure 8.3: Graph of the average latency from poller to user interface for a varying number of plugs.

Looking at Fig. 8.3 with 30 to 75 readings a second the latency does not increase by more than 3-5 ms. Up to 20 plugs reading once per second, the tests shows negligible differences and suggests that 75 ms is the lowest possible latency that the system can achieve. From 75 plug readings/s the system starts to show significant increases in latency as the speed is increased. At 95 plug readings a second the system hits a latency of 274 ms, going higher than this results in an ever increasing latency.

For reference, tests were also carried out with the system running on the laptop mentioned in 7.1. This meant running the system with 100, 200, 300, 400 and 500 simulated plugs sending one message every second. These different quantities of messages per second yielded the same delay, around 5 ms. The program which generates these simulated readings does not respond well to over 500 generated messages per second. Therefore the limit was set to 500.

8.4 Fault tolerance

In section 7.2.5 three different tests were outlined, the results from these tests will be presented in this section.

After disconnecting and reconnecting one plug, the system returns to its usual state, nothing changes. Disconnecting one plug will lead to the individual value of that plug not being updated, while the rest of the system keeps functioning normally. Once the plug is reconnected, the individual plug value will continue updating.

Messages that are not in the correct format or has unreadable data is discarded at the processing bolt. This means that the bolt does not perform any processing and no result is emitted. The bolt then continues with the next message it receives. In other words, data formatted incorrectly, or simply corrupted data, will not affect the system in any way. No unnecessary calculations will be made and corrupted messages will be discarded.

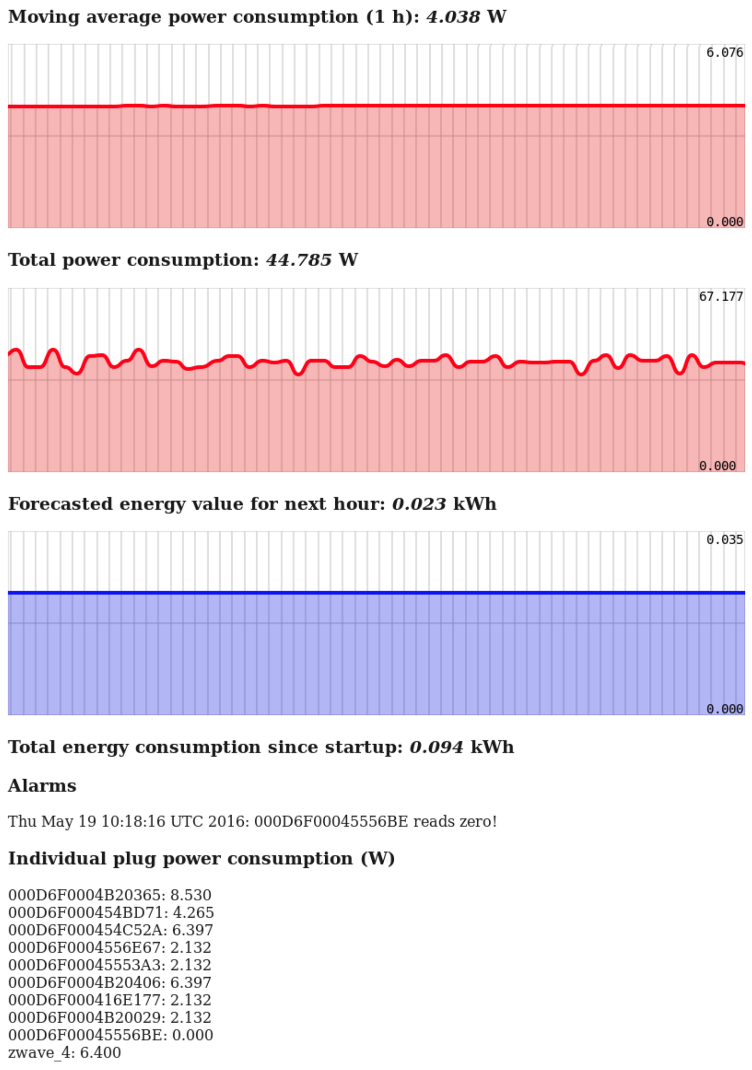
If the processing engine is shut down for any reason, data from the plug will be queued in the message broker so that it can be processed at a later time. There are settings available in the message broker that determines how long messages should be kept after being queued. When the processing framework is started again, it will start consuming the queued messages at the maximum rate it can handle until it is reading messages in real time. Shutting down the engine will as such not have any implications on the system, which will work normally once it is started back up.

8.5 User interface

Creating a user interface was not a priority in this project, hence the interface is very simple. When a bolt has some value that is to be presented, the bolt sends this value back to the corresponding partition of the Kafka topic. A small Node.js-application then consumes the messages from Kafka and presents the values through a web API. A simple web page, made in

JavaScript with the help of a chart library, then fetches the data from the API and plots charts for the user. A screenshot of the UI can be seen in Fig. 8.4.

The UI displays graphs of the moving average for power consumption, the total power consumption, a predicted energy consumption for the next hour as well as the last five alarms. It also displays the current power readings from each energy plug. All information is updated in real time and the graphs display the last minute of data. The graphs scales are automatically adjusted based on the maximum value for the specific statistic.



Moving average power consumption (1 h): ***4.038*** **W**

Total power consumption: ***44.785*** **W**

Forecasted energy value for next hour: ***0.023*** **kWh**

Total energy consumption since startup: ***0.094*** **kWh**

Alarms

Thu May 19 10:18:16 UTC 2016: 000D6F00045556BE reads zero!

Individual plug power consumption (W)

```
000D6F0004B20365: 8.530
000D6F000454BD71: 4.265
000D6F000454C52A: 6.397
000D6F0004556E67: 2.132
000D6F00045553A3: 2.132
000D6F0004B20406: 6.397
000D6F000416E177: 2.132
000D6F0004B20029: 2.132
000D6F00045556BE: 0.000
zwave_4: 6.400
```

Figure 8.4: The user interface for the system.

Chapter 9

Discussion

The system was developed through an iterative process. When problems were encountered, certain parts of the system had to be remade throughout the development. In this chapter, different design choices made during the project as well as the final result will be discussed.

9.1 Method

As mentioned in 1.4, the addition of components to the system was handled in two phases. This eased the development as all parts of the system were given some research time before implementation, which made it possible to plan for parallel development of the system's different parts. The research of which frameworks and software to use was of some help, however it would probably have helped to do more in depth studies, and even setup a trial system before settling on what to use. It also proved quite hard to actually develop the different parts in parallel. In reality the system could more or less only be split up into two parts; the plug data parser and everything else. With more planning the user interface could have been developed in parallel. That would however require precise definition of how communication between the processing engine and the user interface was to be done to be defined early on. The development of the use cases also proved hard to actually carry out in parallel, since all development was done on one single machine instead of two or more. One conclusion that can be drawn from this is that the implementation phase was the problematic one. Perhaps by extending the research phase the problems with the implementation could have been eased. This could have given enough time for planning of the implementation to allow for a more parallel implementation of the system's components.

9.2 Plug data parser

Using data from two different brands could present problems with regards to polling intervals. If the polling intervals differ it might have consequences the rest of the system. Large differences in polling interval would not have a direct negative impact but the system would appear slower. A moving average or total power consumption would still be calculated correctly but would approach the correct value at a slower pace. The same goes for alarms and forecasting, they will lag behind.

Maximum polling speed could also differ internally for a brand. This is the case for the chosen plug brands in this project, since both Plugwise and Z-Wave are mesh networks. A message may have to travel through a large number of plugs before ending up at the receiver. This can cause a significant delay and has not been part of the tests. A plug far away from the receiver could potentially hold up the whole polling and thus decrease the polling speed for an entire brand. One way to avoid this problem would be to ensure that all plugs are in range of the receiver. However, this approach would limit the range of the energy plugs to an extent that may be unacceptable to the average user. For example Plugwise only has a range of five meters and that is in an open space, walls would most likely limit this even further.

The results, available in section 8.3.1, regarding polling speed show similar results for both plug brands. With values of 12.7 readings/s for Plugwise and 12.5 readings/s for Z-Wave, there could be a problem with the testing methodology leading to incorrect results. The similarities could be due to limitations in the computer that was used while testing or in the Node.js framework which was used for the parser. Further tests could be done to strengthen what is claimed in the project's results. Example of such tests could be to use a different computer and/or use another, more low level, programming language.

Another difference between energy plug brands is the decimal precision. At a low energy consumption, Z-Wave would appear to update at a slower pace. The reason for this is that the precision of the energy consumption results in read values being equal to or higher than 0.01 kWh. Hence, having a power draw of for example 5 W would only give a nonzero reading every $0.01/\frac{0.05}{3600} = 720s$ or every 12th minute. Whereas Plugwise with a precision resulting in values larger than $5.92354363 * 10^{-7}$ kWh being shown, would give a reading every $5.92354363 * 10^{-7}/\frac{0.05}{3600} \approx 0.043s$. However, precision of this value is redundant, considering that new readings are sent once a second.

9.3 Message broker

The purpose of message brokers in the project's system is merely to provide the ability to gather data from many different sources. Even then it is reasonable to consider how the message broker will affect the system when large amounts of data traverse the system. There are certain configurations that perform better than others. However, there was not any extended amount of time spent configuring Kafka to ensure that it performs as well as possible considering the circumstances. When running the system on the ODROID-XU4, the data processing framework was the bottleneck rather than the message broker. Because of this, more time needed to be spent configuring Apache Storm rather than Apache Kafka.

9.4 Forecasting, statistics and alarms

The main purpose of the use cases in this project was to showcase what could be done while providing a foundation for building of more advanced processing algorithms. Below the use cases are discussed briefly, forecasting and statistics are discussed in more detail.

If the use cases presented in this project are found lacking by the user, additional functionality can be implemented without losing the interconnectivity of the system's different components. Properties of the alarm use case can be changed by altering the alarm bolt in the storm topology. The alarm functions as an example of how the project's system can be used to implement a feature that alerts the user of unwanted behaviour in appliances. The system can be used to alert the user when the current total power usage exceeds the moving average for the last hour by more than 50%. A galloping power consumption may incur higher costs than the user originally expected.

9.4.1 Improving forecasting of future consumption

As seen in section 3.4.1 there are a few other forms of exponential smoothing that can be used for forecasting. All of these take more variables into account than the form which has been used in this project. In [21] one of these other forms predicted the future power consumption with an error margin between 1.14% and 9.59%. It is reasonable to believe that the same results can be achieved with the project's system. However none of the algorithms are perfect and succeeding in forecasting energy/power consumption would be a significant achievement. Doing this is far beyond the scope of this report.

9.4.2 Efficiency when calculating a moving average

Calculating a moving average involves saving all readings during the time window in memory, which would mean that the available memory limits the size of the window and/or the number of plugs that can be used. The JSON string used in the system is about 93 bytes in size, depending on the format of the plug id this can vary somewhat. Assuming a size of 93 bytes, the total memory usage for one single plug with a sliding window of an hour would be $93 * 3600 = 334800 \quad bytes$ or $0.3348 \quad MB$. Given a situation where the system has 100 megabytes of available memory that would be sufficient to support $100/0.3348 \approx 299 \quad plugs$, as seen in section 8.3.2 the system is not able to handle more than about 95 plugs. As such the memory usage for the moving average calculation would not be a limiting factor in terms of how many plugs the system can handle.

9.5 Performance evaluation

There are several factors that comes in to play when testing performance of a software. The hardware used when testing the system has by far the most impact on the final result. Still, getting satisfactory performance may require more than just additional processing power. There is a cost to performance ratio to consider as well as the fact that there are no infinitely fast processors available on the market. To go further without improving the system with expensive hardware, the system will need to utilise several cores and even machines at once. Apache Storm can handle all the intricate parts such as distributing work across several cores and machines. For that to work, logic focused on parallelism is needed to perform these computations.

Throughout the project the idea has always been to run the whole system on an ODROID-XU4 to test hardware limitations of the system. At first the system's performance on the ODROID, when running with 10 energy readings per second, resulted in maximum usage of the CPU and system freezing from time to time. OpenJDK was used to run the message broker and the data processing engine but was replaced in an attempt to increase performance. Switching to the official JDK version Oracle Java 8 brought the performance to a more acceptable level. The CPU usage problem disappeared and the internal processing times in Storm went from 30+ ms to around 5-10 ms when performing tests with ten energy readings per second. This indicates that the java runtime environment of OpenJDK caused the problem.

As seen in 8.3.2, the systems capability of handling larger amounts of plugs is tested. When reaching close to 100 messages per second, the end-to-end latency of the system increases drastically. The reason for this is either that the processing engine can not process messages at a desired speed or that the user interface get overwhelmed with data and causes it to fall behind. Both of these possible causes are most likely related to the same underlying problem. Namely that the ODROID computer does not provide enough computing power when running all system components locally while at the same time producing 95+ messages per second. As this system can be distributed between machines, adding more ODROIDs or even more powerful computers can increase the performance greatly. The plug data parser can be run on one machine, the message broker on a second and so on. This makes the system highly scalable.

The goal set in section 2.2 of being able to handle around 200 plugs is not met at the current state and hardware configuration of the system. The delays are simply too high. As stated in 8.3.2, the performance greatly increases when running the system on more competent hardware. When testing the system on the reference laptop described in 7.1, the internal processing times in Apache Storm went from around 10 ms on the ODROID to a little less than a millisecond. The end-to-end latency also decreased from around 75 ms to somewhere between 5 and 10 ms. This is a significant difference in performance. By using the reference laptop computer with more computing power, the goal of handling 200 plug readings per second can be realised. Additional tests with up to 500 simulated plugs on the reference laptop resulted in about the same latency as with 200 plugs. This indicates that the maximum number of plug readings per second that the system can handle is greater than the tested 500 readings per second, if more powerful hardware is used.

9.6 System improvements

Several new features can be added to the system including modules for other plug brands; calculation of a median power consumption; more advanced forecasting algorithms; and statistics for individual plugs. However, these changes are more of the shallow type. The more interesting improvements, following the purpose of the project, would apply to performance and fault tolerance.

Improving the performance could be done in several ways. Finding a bottleneck in the system is of interest in order to know which type of system improvements that can be made. There is not really any limit to how many plugs the plug data parsing can handle. If one data plug parser reaches the maximum amount of plugs it can handle, another parser process can be started to compensate for this. The message broker, Apache Kafka, does not present itself as a bottleneck either as it has functionality to solve performance issues; for higher performance it is possible to allocate more processing power by distributing the message broker's work load across several computers. That leaves the processing framework, Apache Storm, which in itself does not present any real limitation as it can be scaled up to more nodes on the premise that the processing can be done in parallel. This is not the case of all the bolts in the system as some calculations, such as summarising, can not be done fully in parallel.

In case just lower latencies in the project's system are desirable, a potential improvement could be achieved by swapping Apache Kafka for RabbitMQ. In [7] RabbitMQ is found to have lower latencies than Kafka for messages smaller than 100KB. Considering that the messages sent through through the system are around 93 bytes in size, a switch to RabbitMQ could potentially reduce the end-to-end latency for the system.

9.7 Implications for society

The system presented in this report could contribute to smarter use of energy in the power grid. With smart energy plugs offering different types of energy data processing, some users might want the same type of processing for all plugs to not be dependent on the default processing for a brand. The project's system can make the utilisation of energy plugs more appealing to these users as the system can be tailored to cater to user's specific demands in terms of processing. As mentioned in a report by Tsuyoshi Ueno et al. [5], monitoring and presenting data to residents, could results in a decreased energy consumption. By making energy plugs more appealing, the system can contribute to more efficient consumption of energy among users of the system, resulting in less electricity expenditure for the user as well as reducing their carbon footprint. The system could also be used as a stepping stone towards building a smart energy home. The data collected by the system could be used to program home appliances to use less electricity based on various factors. For example, when the price of electricity is high, the home appliances can be programmed to enter a energy saving mode, which can then be released when the price is back to normal. This functionality of course implies that the system has to be modified with additional use cases, which is beyond the scope of this thesis. However, if implemented, it could help make the vision of smart energy homes a reality [37].

Chapter 10

Conclusion

The purpose of this project was to create a general purpose modular framework for collecting and processing data from smart energy plugs. This system was to enable the implementation of data processing algorithms without having to consider the underlying smart energy plugs. The system implemented in this project consists of: a plug data parser, a message broker, a data processing engine. A plug data parser is needed in order to send the data from the energy plugs to the message broker. Energy plug brands that are not supported by the system can be added by developing additional brand specific modules for the plug data parser. By implementing the system with a message broker, a focal point for data originating from different producers was created. The message broker accepts data regardless of the sender, which means that the type of incoming data does not matter. A data processing engine pulling its data from the message broker then performs processing on the energy plug data despite the type of brand. Processing of the energy plug data is done with Apache Storm which framework allows for the user to change the functionality of the processing without having to change the components in the rest of the system. This means that the system can be used for other applications than smart energy plugs. For example, a user in need of analysing air quality can connect air sensors with the system presented in this project. With these factors in mind, the project's system as a whole can be used to apply the same type of processing to not only energy plugs of different brands, but other types of data producers as well.

Bibliography

[1] Svensk Energi. *Svensk Energi Elanvändning*. 2012. URL: `http://www.svenskenergi.se/Elfakta/Elanvandning/` (visited on 02/02/2016).

[2] U.S. Energy Information Administration. *U.S. Energy Information Administration How much electricity does an American home use*. 2015. URL: `https://www.eia.gov/tools/faqs/faq.cfm?id=97&t=3` (visited on 02/02/2016).

[3] Tom Hargreaves, Michael Nye, and Jacquelin Burgess. "Making energy visible: A qualitative field study of how householders interact with feedback from smart energy monitors". In: *Energy policy* 38.10 (2010), pp. 6111–6119. Elsevier, 2010.

[4] Andrea Monacchi, Dominik Egarter, and Wilfried Elmenreich. "Integrating households into the smart grid". In: *Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES), 2013 Workshop on*. IEEE. 2013, pp. 1–6.

[5] Tsuyoshi Ueno et al. "Effectiveness of an energy-consumption information system on energy savings in residential houses based on monitored data". In: *Applied Energy* 83.2 (2006), pp. 166–183. Elsevier, 2006.

[6] Xudong Ma et al. "Supervisory and Energy Management System of large public buildings". In: *Mechatronics and Automation (ICMA), 2010 International Conference on*. IEEE. 2010, pp. 928–933.

[7] Supun Kamburugamuve, Leif Christiansen, and Geoffrey Fox. "A framework for real time processing of sensor data in the cloud". In: *Journal of Sensors* 2015 (2015). Hindawi Publishing Corporation, 2015.

[8] D. Kranzlmüller and A.M. Tjoa. *Information and Communication on Technology for the Fight against Global Warming: First International Conference, ICT-GLOW 2011, Toulouse, France, August 30-31, 2011, Proceedings*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, p. 104. ISBN: 9783642234477.

[9] Plugwise BV. *FAQ|Plugwise*. 2016. URL: `https://www.plugwise.com/faq` (visited on 05/10/2016).

[10] Maarten Damen. *Plugwise unleashed.* 2015. URL: `http://maartendamen.com/wp-content/uploads/downloads/2010/08/Plugwise-unleashed-0.1.pdf` (visited on 03/08/2016).

[11] Mikhail T Galeev. "Catching the z-wave". In: *Embedded Systems Design* 19.10 (2006), p. 28. CMP MEDIA LLC, 2006.

[12] Z-Wave Alliance. *About Z-Wave Technology.* 2016. URL: `http://z-wavealliance.org/about_z-wave_technology/` (visited on 03/16/2016).

[13] Greenwave Systems. *Greenwave PowerNode NS310-F.* URL: `http://manuals.zwaveeurope.com/make.php?lang=en&type=&sku=GWRENS310-F` (visited on 05/12/2016).

[14] Inc. Gartner. *Message Broker - Gartner IT Glossary.* 2016. URL: `http://www.gartner.com/it-glossary/message-broker/` (visited on 05/12/2016).

[15] Apache Software Foundation. 2016. URL: `http://kafka.apache.org/documentation.html#intro_topics` (visited on 04/01/2016).

[16] Inc. Pivotal Software. *RabbitMQ - Compatibility and Conformance.* 2016. URL: `https://www.rabbitmq.com/specification.html` (visited on 05/12/2016).

[17] John O'Hara. "Toward a Commodity Enterprise Middleware". In: *Queue* 5.4 (May 2007), pp. 48–55. New York, NY, USA: ACM, May 2007. ISSN: 1542-7730. DOI: `10.1145/1255421.1255424`. URL: `http://doi.acm.org/10.1145/1255421.1255424`.

[18] Apache Software Foundation. *S4: Distributed Stream Computing Platform.* 2013. URL: `http://incubator.apache.org/s4/` (visited on 05/12/2016).

[19] Apache Software Foundation. *Concepts.* 2015. URL: `http://storm.apache.org/releases/1.0.0/Concepts.html` (visited on 03/30/2016).

[20] Apache Software Foundation. *Apache Flume.* 2012. URL: `http://flume.apache.org/` (visited on 05/12/2016).

[21] Nur Adilah Abd Jalil, Maizah Hura Ahmad, and Norizan Mohamed. "Electricity load demand forecasting using exponential smoothing methods". In: *World Applied Sciences Journal* 22.11 (2013), pp. 1540–1543. Citeseer, 2013.

[22] Charles C Holt. "Forecasting seasonals and trends by exponentially weighted moving averages". In: *International journal of forecasting* 20.1 (2004), pp. 5–10. Elsevier, 2004.

[23] Node.js Foundation. *About Node.js.* 2015. URL: https://nodejs.org/en/about/ (visited on 02/16/2016).

[24] npm Inc. *What is npm?* 2016. URL: https://docs.npmjs.com/getting-started/what-is-npm (visited on 03/21/2016).

[25] B. Ellis. *Real-Time Analytics: Techniques to Analyze and Visualize Streaming Data.* Wiley, 2014. Chap. 4. ISBN: 9781118838020.

[26] Apache Software Foundation. *Companies Using Apache Kafka.* 2015. URL: http://cwiki.apache.org/confluence/display/KAFKA/Powered+By (visited on 03/30/2016).

[27] Apache Software Foundation. *Companies Using Apache Storm.* 2015. URL: http://storm.apache.org/Powered-By.html (visited on 03/30/2016).

[28] Apache Software Foundation. 2015. URL: http://storm.apache.org/releases/0.10.0/Daemon-Fault-Tolerance.html (visited on 03/30/2016).

[29] Apache Software Foundation. 2015. URL: http://storm.apache.org/releases/1.0.0/Understanding-the-parallelism-of-a-Storm-topology.html (visited on 03/30/2016).

[30] Daniel-Octavian Rizea, Alexandru-Corneliu Olteanu, and Dan-Stefan Tudose. "Air quality data collection and processing platform". In: *RoEduNet Conference 13th Edition: Networking in Education and Research Joint Event RENAM 8th Conference, 2014.* IEEE. 2014, pp. 1–4.

[31] Elias Karakoulakis. *OZW utilities.* 2016. URL: http://www.openzwave.com/home (visited on 04/13/2016).

[32] Z-Wave.Me. *UZB.* 2016. URL: http://www.z-wave.me/index.php?id=28 (visited on 04/13/2016).

[33] Express. *Fast, unopinionated, minimalist web framework for Node.js.* 2016. URL: http://expressjs.com/ (visited on 04/13/2016).

[34] Joe Walnes & Drew Noakes. *Smoothie Charts: A JavaScript Charting Library for Streaming Data.* 2016. URL: http://smoothiecharts.org/ (visited on 04/13/2016).

[35] Apache Software Foundation. 2016. URL: http://kafka.apache.org/documentation.html#log (visited on 04/06/2016).

[36] Apache Software Foundation. 2016. URL: http://storm.apache.org/releases/1.0.0/Guaranteeing-message-processing.html (visited on 04/06/2016).

[37] M. Jahn et al. “The Energy Aware Smart Home”. In: *2010 5th International Conference on Future Information Technology*. May 2010, pp. 1–8. DOI: `10.1109/FUTURETECH.2010.5482712`.

Appendix A

Plugwise protocol

Table A.1: Plugwise protocol for requests

Example data (HEX)	**Explanation**
5533	Header
0012	Request code
000D6F0004B20365	The mac address for some plug
1B04	CRC16 checksum
0D (carriage return)	Footer

Table A.2: Plugwise protocol for responses

Example data (HEX)	**Explanation**
5533	Header
0013	Response code
B7EC	Sequence number
000D6F0004B20365	The mac address for some plug
0001	Pulses during the last second
0004	Pulses during the last 8 seconds
000005E1	Total number of pulses
0000	Unknown and not relevant
0000	Unknown and not relevant
0002	Unknown and not relevant
9845	CRC16 checksum
0D (carriage return)	Footer

One pulse is 468.9385193 kWs (kilo watt seconds). Getting the consumption in kWh can be done like this $\frac{pulses}{3600}/468.9385193$.