CHALMERS

Energy efficient data synchronization in mobile applications

A comparison between different data synchronization techniques

Master of Science Thesis in Computer Science and Engineering

MAGNUS LARSSON
MARCUS STIGELID

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Gothenburg, Sweden, September 2015

Energy efficient data synchronization in mobile applications
A comparison between different data synchronization techniques

MAGNUS LARSSON
MARCUS STIGELID

Examiner: PHILIPPAS TSIGAS

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

Department of Computer Science and Engineering
Göteborg, Sweden September 2015

Abstract

The mobile phone is the most widespread computing platform in today's society, having over 2 billion users. With each year mobile devices becomes increasingly powerful and are also able to synchronize more data. However, the battery capacity of a device remains the same, thus limiting the time of use. This thesis investigates the energy efficiency of different network synchronization techniques to reduce energy consumption when synchronizing data between a mobile phone and a data provider. Prototypes implemented using different network synchronization techniques are used to evaluate which technique is the most energy efficient. Furthermore, the prototypes are implemented using different frameworks, user interfaces, and update frequencies to find how other factors affect energy consumption in comparison to network data retrieval.

The results show that using a network protocol with a smaller header consumed less energy. It is also shown that using a constantly open connection is the most energy efficient choice of network technique compared with techniques that frequently opens new connections. However, when the frequency of creating new connections decreases it is more energy efficient to close the connections. Furthermore, it is demonstrated that the energy consumption of a mobile application mainly depends on the frequency with which the phone updates the graphical user interface. Another important factor is a graphical interface, which consumes more energy than network data retrieval does. The thesis is concluded with a suggestion that to reduce energy consumption, a mobile application should optimize its update interval for which data is visualized.

Keywords: Energy efficiency, energy consumption measurement, mobile applications, computer networks, data synchronization

Acknowledgements

We would like to thank our academic supervisor Aras Atalar for his useful feedback and constructive support. We would also like to thank our examiner Philippas Tsigas. Furthermore, we send a thank you to our supervisor at i3tex, Per Lundberg, for his technical expertise as well as his commitment and dedication during the course of this thesis. We are also grateful for all the help, support, and positive comments we have received from other people at i3tex. Lastly, we would like to send a thank you to the people who supported us from Volvo; Erik Israelsson for implementation details and feedback, and Karl-Johan Jönsson for his technical expertise and help with debugging as well as technical features.

Magnus Larsson and Marcus Stigelid, Gothenburg, September 2015

Dictionary

.NET	*A software framework used to develop applications.*
API	*Application Programming Interface. An interface that a programmer can use to call underlying system functions.*
GUI	*Graphical User Interface. An interface consisting of graphical components that handles input and output in an application.*
IDE	*Integrated Development Environment. An environment where a programmer can write code, debug and release an application.*
IP	*The Internet Protocol is a network protocol for sending data from a source to a destination.*
JavaScript	*An interpreted computer programming language used mainly for interactive web content.*
PCL	*A Portable Class Library is a library that contains code that can run on multiple platforms.*
SDK	*Software Development Kit. A set of tools that enables the creation of applications for a specific development platform.*
TCP	*Transmission Control Protocol. A network protocol for reliable packet transmission.*
UDP	*User Datagram Protocol. A protocol for sending network packets without delivery guarantees.*

Contents

1 Introduction 1

1.1 Purpose . . . 2

1.2 Problem description . . . 3

1.2.1 Energy efficiency . . . 3

1.2.2 Challenges in prototype design . . . 4

1.2.3 Functionality and energy efficiency trade-off . . . 4

1.3 Limitations . . . 5

1.4 Structure of report . . . 5

2 Related work 6

3 Background 8

3.1 Platforms . . . 8

3.1.1 Android . . . 8

3.1.2 iOS . . . 9

3.2 Frameworks . . . 9

3.2.1 Native applications . . . 9

3.2.2 Frameworks using web technology . . . 10

3.2.3 Xamarin . . . 11

3.2.4 Corona . . . 12

3.3 Electrical theory . . . 13

3.3.1 Measuring power . . . 13

3.3.2 Calculating energy consumption . . . 14

3.4 Network protocols . . . 15

3.4.1 TCP . . . 15

3.4.2 UDP . . . 16

4 Methodology 17

4.1 Energy measurement approach 17
4.1.1 Hardware logging 18
4.1.2 Sanity checks 19
4.2 Implementation approach 20
4.2.1 Data synchronization techniques 21
4.2.2 Prototypes 21
4.2.3 Consistency 22
4.3 Acquiring the results 22
4.3.1 Prototypes 23
4.3.2 Energy measurement protocol 24
4.3.3 Models 25

5 Design and Implementation 26
5.1 Mobile application 26
5.1.1 Architecture of the mobile application 26
5.1.2 Base prototype 27
5.1.3 Prototype 1 27
5.1.4 Prototype 2 28
5.2 PowerLogger 30
5.2.1 Functionality 30
5.2.2 User Interface 32
5.3 Data provider 33

6 Result 36
6.1 Energy measurement 36
6.1.1 Base prototype 37
6.1.2 Prototype 1 38
6.1.3 Prototype 2 40
6.2 Factors affecting energy consumption 41
6.2.1 Update frequency 42
6.2.2 Graphical user interface 43
6.2.3 Frameworks 44
6.3 Energy models 45
6.4 Additional findings 46

7 Discussion 49
7.1 Results 49
7.2 Cross-platform frameworks 51
7.3 Measuring energy 53
7.4 Mobile application 55
7.5 Future work 55

8 Conclusion **56**

References **58**

Appendix A Related projects **I**

Appendix B Requirements **IV**

B.1 Functional requirements IV

B.2 Nonfunctional requirements V

List of Figures

1 The architecture of mobile applications using web technologies . . . 11
2 Shared runtime on top of Android's Linux kernel 12
3 Scheme for measuring current through current resistive sensing . . . 13
4 Example of power consumption for a Samsung GT-S7275R 14

5 Hardware logging setup 19
6 Sample log from sanity check 20

7 Prototype 1 29
8 Prototype 2 30
9 User interface for the mobile part, with settings and main view . . . 32
10 Output for original (left) and smoothed graph (right) 33
11 Relation between a running process and energy consumption 34

12 Relationship between synchronization techniques in Prototype 1 . . 39
13 Relationship between synchronization techniques in Prototype 2 . . 41
14 Difference in energy consumption between factors 43
15 Energy consumption of graphical user interfaces and network techniques 44
16 Energy consumption in different frameworks shown as average power. 45
17 Average power for different frequencies 46
18 Start and tail energy 47
19 Start and tail energy stacked on run time energy 48

List of Tables

1 Sanity check results 20
2 Samsung GT-S7275R hardware specification 23
3 Prototypes divided into different factors 23
4 Prototypes with techniques and update intervals 24

5 Energy consumption of the base prototype shown as average power (W) 37
6 Energy consumption in Prototype 1 shown as average power (W) . 39
7 Energy consumption in Prototype 2 shown as average power (W) . 40
8 Constant values for start and tail energy (Wh) 48

1

Introduction

A few decades ago, a mobile phone was nothing more than an embedded device. Dimarzio describes the community of mobile application developers as a small sect, and states that development was difficult because ”embedded device manufacturers were notoriously stingy on feature support” [1]. Over the course of the last ten years, the mobile phone has evolved from a static machine to an almost full-fledged computer. This has been made possible through extensive development of mobile platforms and frameworks, as well as through adding sensors for additional input. The rapid change in functionality and usability has made mobile phone application development increasingly popular. Today, the mobile phone is the most widespread computing platform in the world with over 2 billion users [2]. As the market for mobile applications continues to expand, more applications retrieve data through mobile phone components such as WiFi or 3G [3].

Because of the mobile nature of a phone, it needs to be powered by a battery. This severely limits the computational power since energy resources are finite. According to Moore’s law, the processing power of a mobile device is doubled every two years, which can be compared with the capacity of the battery which is doubled every ten years [4]. This means that an application has the possibility to run heavy calculations, but doing so carelessly will drain the battery. In addition to this, using components such as WiFi or 3G consumes a lot of energy [5]. This puts constraints on each individual application to use the phone’s resources in the most conservative manner possible. One way of minimizing energy consumption is to make sure that the used synchronization techniques, which are different ways for an application to retrieve data through a component, are as energy efficient as possible.

The main goal when creating energy efficient mobile applications is to increase the user experience by extending the battery life. However, energy efficiency is not only useful as a way of improving user experience but also as a way of making sure that a user can use an application as much as possible. Thus, there is a need from a developer's perspective that all applications should use the mobile phone's energy resources sparingly.

There exists a number of studies in energy efficiency targeting the two most popular mobile operating systems, Android and iOS [6]. These studies [7–10] show different coarse-grained ways of measuring power consumption in a mobile phone. However, Li et al. [11] describe that it is difficult for a developer to easily find "quantitative and objective information about the behavior of apps with respect to energy consumption".

1.1 Purpose

This thesis aimed to develop a number of mobile networking application prototypes, each using a different synchronization technique. The prototypes have been developed at the company i3tex AB and have been used in real-world scenarios at Volvo Cars to visualize sensor data from a car in real time. Together with Volvo Cars requirements have been developed, and both Volvo and i3tex have supported this thesis with feedback as well as testing equipment.

The prototypes were used to evaluate the main purpose of this thesis, which is to investigate the energy efficiency of different data synchronization techniques. A comparison between the techniques was made in order to determine which technique is more energy efficient. To gain further insight of different synchronization technique's impact on energy consumption, the general elements of a mobile networking application were used to determine other possible factors.

As different synchronization techniques have different advantages and drawbacks, the functionality of a mobile application can prove to be the decisive factor when deciding which technique to use. Another purpose was therefore to investigate the correlation between an application's functionality and its energy efficiency. Each prototype application will thus include different functionality so that a comparison between the prototypes and their respective energy consumption can be made.

Another aim of the project was to investigate different frameworks that can cross-compile code written in one language into code for different platforms. Since developing native applications for different platforms takes a lot of time [12], it is interesting to investigate if these tools can be used to create energy efficient, complex applications. It is also interesting to compare the accuracy in functionality, and its correlation to energy efficiency, provided by different cross-compilation frameworks. The investigation of frameworks with respect to energy efficiency is discussed in Section 7.1, and the frameworks themselves are discussed in Section 7.2.

1.2 Problem description

The main problem that this thesis aims to solve is to find a relationship between different synchronization techniques, describing which technique is more energy efficient. In the following sections some of the subproblems that arise from the main problem are discussed.

1.2.1 Energy efficiency

The problem of comparing energy efficiency of different synchronization techniques is not a trivial task. Much of today's measuring equipment consists of either expensive power meters [8, 13], or complex statistical models that target a specific phone running a specific operating system [3, 11]. To be able to evaluate whether an application is energy efficient or not, a subproblem was defined by which tools to use when measuring energy efficiency. As a developer, finding energy critical parts of an application can be hard due to the multiple levels of middleware that exists between an application and the hardware. The problem exists both when measuring energy as well as when predicting energy consumption using energy models, as the energy cost from the middleware is unknown [14].

As different phones have different characteristics, there existed a problem to determine which phone to use when investigating the main problem. A phone's underlying operating system, hereby referenced as platform, manages the resources of the phone in a specific way. How well resources are managed depends on both the phone's hardware as well as the platform executing the application code. Thus, an application running on one platform might be considered energy efficient but when it is running on another platform it might not. The code in a mobile application are the instructions the platform executes to determine what hardware to use and how to use it. The content of the instructions depends on the tools used to create the application, which in turn are often provided by a framework. Depending on which framework a developer uses, the code that the mobile application will consist of may look and function differently. Therefore, the energy efficiency of a mobile application does not only depend on the platform that is used to run the app, but also on the framework used to implement it.

The problem of determining which phone to use was solved by investigating a number of platforms and frameworks. The investigation included the creation of prototypes implemented using multiple frameworks for multiple platforms, to determine the suitability of platform and framework when measuring energy consumption. The desired properties for a phone was that it uses a platform with a large user base, while it also supported tools to measure and improve energy performance. Examples of such tools are energy monitors and functions provided by the platform that can measure different values in relevant hardware components, such as battery or CPU.

1.2.2 Challenges in prototype design

The most prominent issue when implementing a prototype was how to connect it to a data provider that streams real time data. When developing the application, a connection needed to be simulated so that functionality could be evaluated without having to be in a car. To solve this issue, a computer was used to emulate the sensors and protocols used in the communication with the app.

Another problem with the prototype design was the consistency of data. If the application lost the connection to the sensor, data may have become obsolete. Data may also have been delayed or re-sent which made it arrive later than expected, thus producing a faulty output. Therefore, an investigation in consistency models was carried out to determine how to handle the case when a prototype lost its input stream.

1.2.3 Functionality and energy efficiency trade-off

A problem when creating energy efficient applications was to find the most energy efficient version of a specific feature. It is possible for a developer to create a working mobile application which seems to work fine, even though it is energy consuming. An example of this is an email application which asks a server if new emails has arrived every second. Since it is highly unlikely that a user receives an email every second, the application can ask the server less frequently. The trade-off with lowering the time between synchronizations is that though saving battery, there may be a delay between the time that the email arrives to the server and the time that the application retrieves it. Thus, to lower energy consumption an application developer needs to find an optimized synchronization delay. To investigate the correlation between functionality and energy efficiency, different features will be implemented in different prototypes.

1.3 Limitations

This thesis only considers synchronization techniques that can be used when retrieving data through WiFi. This rules out other energy consuming components, but within the given time frame, focusing on only WiFi can provide a more detailed study of energy efficiency in connected mobile applications. Since an increasing number of applications build their main functionality around synchronized data [13], the result of this study could prove useful for developers and researchers in this area.

The prototypes will only be tested on a limited number of mobile phones. The desired properties for a phone is described in Chapter 1.2.1, and defines that the phone should support tools to measure and improve energy performance.

The amount of features possible to implement in a prototype will also be limited. Therefore application features will be implemented based on a priority scheme, where the functionality that is the most important will be implemented first. The implementation process will also be divided into different phases to ensure that the creation of each prototype is as efficient as possible.

1.4 Structure of report

The rest of the report will be structured as follows. Chapter 2 describes related work and Chapter 3 outlines the theories used in this project, ranging from different mobile platforms to electrical theory. The methodology as well as the motivations for choices made in the project are described in Chapter 4. Following this, Chapter 5 provides details about the design and implementation of the mobile application prototypes and the energy measurement techniques. The results are presented in Chapter 6 which is then followed by a discussion in Chapter 7. Finally, the conclusion of the thesis is given in Chapter 8.

2

Related work

There are previously published works within the area of energy measurement that are related to this thesis. Hao et al. have published work that uses program analysis to estimate the amount of energy an application consumes during execution [15]. This is done through a CPU profile which contains functions for determining the energy cost on a method level. By applying a set of use cases to the implementation of an application they find which methods that are called, which is used as input to the CPU profile. By summing the cost of each called function the total energy consumption of an application is found.

Hao et al. have also produced three other papers where they build and improve a method for measuring energy consumption. In the first paper [16], they show a method that estimates energy consumption at runtime for each line of code in a mobile application. By applying a number of use cases to a graph-based data structure, which is created from the byte code of an application, the path that is traversed through each method when executing the use cases is created. Using an energy profile consisting of functions for calculating energy consumption for different byte-code instructions, the energy consumption of each instruction in the path can be calculated. This method is further improved in [3], where a high-precision energy platform replaces the energy profile used to determine the energy consumption of specific source lines. In this paper robust linear regression analysis is also added to adjust incorrect values. Finally, the work presented in [11] uses their previous work to evaluate different mobile phone components and their energy consumption in relation to each other.

In [17], Liu et al. discuss energy and performance bugs that developers of mobile applications introduce during development. They identify three types of bugs that affect the energy efficiency and performance of applications: energy leak, GUI lagging, and memory bloat bugs. Finding these types of bugs is not trivial, they may only occur in specific usage scenarios that require complex user interactions. The authors discuss various techniques and tools that can identify these bugs but notes that none of the techniques completely solves the problem, and that there are still questions to answer.

In [2], Rice et al. measure energy consumption in a networking application for the Android operating system. The application downloads a test script from a server, runs it and uploads a log to the server. To measure the energy usage of the application the authors use fine-grained hardware equipment which makes it possible to see the energy consumption of events running for a very short time, such as receiving a data packet. They also discuss the importance of aligning the recorded times when using fine-grained metrics, as for instance transmitting a data packet might be done within a few milliseconds.

Carvalho et al. [4] describe a project in which they study energy efficiency in TCP. The study uses a DC current sensor and an Arduino board to measure energy consumption. A mobile application, implemented with either TCP push or TCP pull, fetches data from a server. The authors run the application for one hour and use different update frequencies when pulling data to determine in which cases either technique is more energy efficient. The paper concludes that pulling data consumes less energy. Burgstahler et al. [13] created a similar project but attained different results than Carvalho et al. According to the authors, a push-based approach consumes less energy than an approach where data is pulled frequently. Despite the differences in the results, both papers state that the application scenario is the most important reason for choosing a specific synchronization mechanism, as differences in energy consumption are negligible.

3

Background

The following sections describe the theoretical background for the mobile platforms and development frameworks that were used for the project. Energy measurement as well as network protocols are described.

3.1 Platforms

This section covers the two, in terms of market share [6], biggest mobile operating systems that are available on the market today. An operating system will hereby be referred to as a platform.

3.1.1 Android

Android is an open source platform originally developed by Android Inc., which was later purchased by Google. It is based on the Linux kernel and is used in personal computers as well as in mobile phones [18]. The platform was released for the first time in the fall of 2008 [19], and has grown ever since. Android is currently the most used mobile operating system in the world [6]. During the launch of Android, Google created the Open Handset Alliance (OHA), a consortium between Google and 65 companies working in the mobile phone sector to ensure that the Android OS is well supported. The companies that are members of the OHA include hardware companies, software companies, and mobile operators. As different phones from different vendors work in different ways, many Android applications need temporary solutions to work equally well on all supported devices. Thus, functionality that is provided in one device might not be provided in another [20].

The main programming language for Android applications is Java, but since the platform is written in the language C, applications can also be partly written in this language. The Android platform provides a framework for developers, called Android SDK, that includes features such as 2D and 3D graphics, multi-touch input, and APIs for accessing networks. The platform also supports multitasking and threading [20].

3.1.2 iOS

iOS is a closed source mobile platform created by Apple Inc, which is used in all of Apple's mobile products such as the iPod, iPad and iPhone. To develop a mobile application for an iOS supported device the iOS SDK is required, which can only be used on a Macintosh computer [21].

iOS applications can be written in Objective-C or in Apple's new programming language Swift [22]. Developers can test their applications using the emulator provided in the Xcode IDE. However, in order to test an application on a real device a membership in Apple's Developer Program is required, at an annual cost of $99. The iOS SDK agreement also states that when publishing an application to the Apple App Store it needs to be validated and approved by Apple before it is publicly released [23]. Examples of what causes an app to be rejected includes bugs, placeholder content, or usage of non-public APIs [24].

3.2 Frameworks

In the following sections the software frameworks used in this thesis will be presented. The first section describes native applications for both Android and iOS. Following that are sections describing different cross-platform application frameworks, where each section is based on the application creation technique that the frameworks use.

3.2.1 Native applications

Applications written using the official framework of a platform are called native applications. Depending on the underlying platform a native application will look and function differently. Native applications are implemented in their platform's native programming language [25].

Writing a native application ensures that users get a native experience on their device. As a platform is mainly optimized for the native language, performance could also be a reason to choose native instead of another application development technique [25]. Another advantage with native applications is that the official development kit together with the native language is optimized for working with internal sensors and hardware. A disadvantage of developing a native application is that it only runs on one platform. To reach more users developers need to have separate code bases that cannot be shared between platforms. Thus, maintaining a native app for several platforms might be more costly and more complex [26].

3.2.2 Frameworks using web technology

There exist a number of frameworks that use different web techniques as their primary source for building cross-platform mobile phone applications. Examples of popular frameworks are Apache Cordova (former PhoneGap), Ionic, Appgyver Supersonic, and Titanium [27–30]. Some of these frameworks provide an integrated development environment (IDE) that can be used to quickly utilize the functionality of the frameworks [12].

An application based on web technology is structured in the same way regardless of which framework is used; the logic of the application is created in a scripted language, and the graphical user interface is written in a markup language. The scripted language is usually JavaScript and the markup language is one that is used on the web, for instance HTML or XML. Using JavaScript with a markup language is one of the main features that the frameworks provide, since this makes it possible to run the application on several platforms using the same code. Furthermore, most frameworks provide plugins or an Application Programming Interface (API) that lets developers interact with the underlying system and its functions such as networking, graphics, and threads [31].

Figure 1 shows an overview of how mobile applications based on web technology are structured and how they can interact with the underlying system. The applications run in a special container called a web view which is a feature provided by the platform that uses the same rendering engine as the platform's web browser. This means that the web view is able to both view web pages as well as execute scripted code written in JavaScript. The application's JavaScript logic can change an HTML view dynamically to add data to the view in real time. In order to run the application on a mobile device, the application is packaged into the native

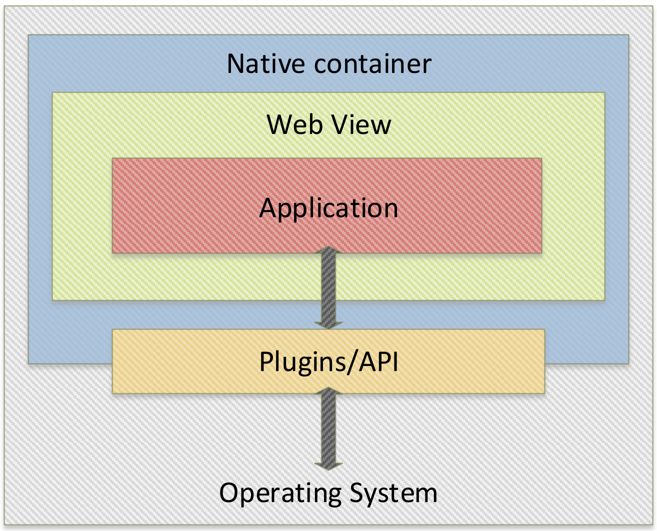


Figure 1: The architecture of mobile applications using web technologies

application container for each specific platform so that it can be installed [32].

3.2.3 Xamarin

Xamarin is a framework that is developed by the company Xamarin. The framework is similar to other cross-platform application frameworks in that it allows developers to create mobile applications that can run on multiple platforms, while only using one programming language. The language of an application implemented with Xamarin is C#, but during compile time the code is translated into platform specific native code [33].

There are two different ways to share code between platforms in Xamarin: Shared Asset Projects (SAP) and Portable Class Libraries (PCL). Shared code must be able to run on all targeted platforms. This means that the platform with the most restrictions set the boundaries for which functionality the shared code can support. An example of such a boundary involves threading, which on some platforms only can be used through a severely restricted API. Native C# threads are therefore not supported in shared code, and to use threads a developer needs to develop own code that provides the same functionality as the thread API [34].

When compiling an application with Xamarin, an intermediate language is used before compiling into native code. For some platforms, like Android, not all code can be translated into pure native code. Thus, a shared runtime called Mono that runs alongside the Java engine must be installed on the Android device [34]. The relation between Mono and Java can be seen in Figure 2, where the framework topology of an application implemented with Xamarin is shown. Xamarin uses the Android bindings on top of Mono to call Android specific functionality, but also uses the .NET APIs that Mono provides for .NET functionality.

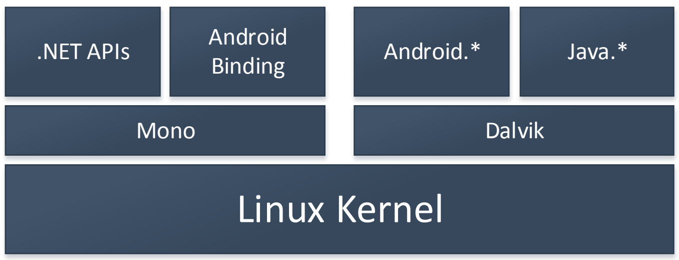


Figure 2: Shared runtime on top of Android's Linux kernel

Being able to bind to Android, a developer has access to all native APIs. This enables the possibility to use third party libraries written in a platform specific language together with C# code. It also ensures that the graphical user interface looks the same as the user interface in a native application [34].

3.2.4 Corona

The Corona framework, or Corona SDK, is a framework developed by Corona Labs. The main feature of the Corona framework is that an application can run on multiple platforms using the same code. This is possible through the use of a virtual machine together with the embedded scripting language Lua. The Corona framework provides libraries which are built to run Lua code on each specific target. When deploying an application, Corona adds the libraries to the application to make sure that it can run on the targeted platform. The libraries also contain native graphical components, so that the app's user interface looks like a native graphical interface on each device. Thus, an application built with the Corona framework can look and function just like a native application [35].

Apart from the already existing functionality of the language, the Corona SDK provides different APIs written in native Lua. The APIs include functionality such as networking, graphics, and multithreading [36].

The Corona framework is mainly built for mobile games. As many games have moving objects, most graphical components are added to the screen using coordinates. Creating an application that supports many different screen resolutions means a lot of code is needed to translate fixed pixels to pixels relative to width and height of the device. Font size is also an issue, as a font may be too small or too large for text to fit on the screen [35].

3.3 Electrical theory

The following sections describe the electrical theory that was used to measure the energy consumption of the prototypes. The first section describes how to measure and calculate power and the second section focuses on energy calculation based on values of instant power.

3.3.1 Measuring power

Power (P) is measured in the unit watt, and depends on both the voltage (V) as well as the current (I). The formula to calculate instant power is:

$$P = V \cdot I \tag{1}$$

Voltage can be measured with a multimeter connected to the positive and negative terminal on the battery. The current can be measured by using resistive current sensing, in which a resistor is connected in series between a battery terminal and the terminal on the phone [37]. This is visualized in Figure 3.

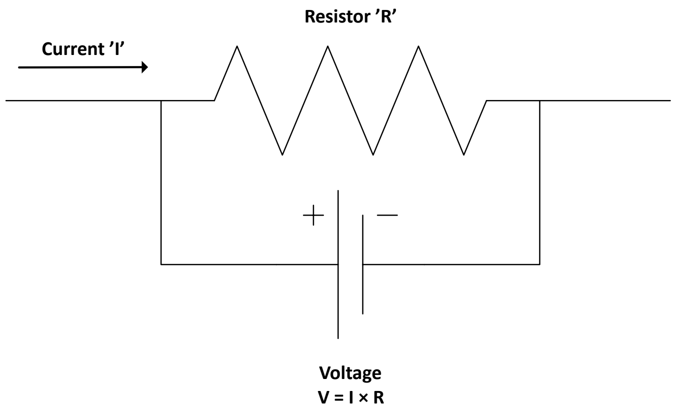


Figure 3: Scheme for measuring current through current resistive sensing

The voltage drop across the resistor is, by Ohm's law, proportional to the current flowing through it. Thus, the formula for calculating the current is:

$$I = \frac{V}{R} \tag{2}$$

3.3.2 Calculating energy consumption

As the energy consumption of a mobile phone constantly changes depending on the workload, the overall consumption is computed over multiple instances of instant power. The unit used when measuring the capacity of a mobile phone battery is therefore not watt but watt-hours, the amount of power consumed during a given time. For instance, one watt-hour is either equal to using one watt for one hour or two watts for half an hour.

In Figure 4, an example of energy consumption for a Samsung GT-S7275R mobile phone is shown. The figure shows 700 milliseconds of data during which approximately 60 measurements were logged.

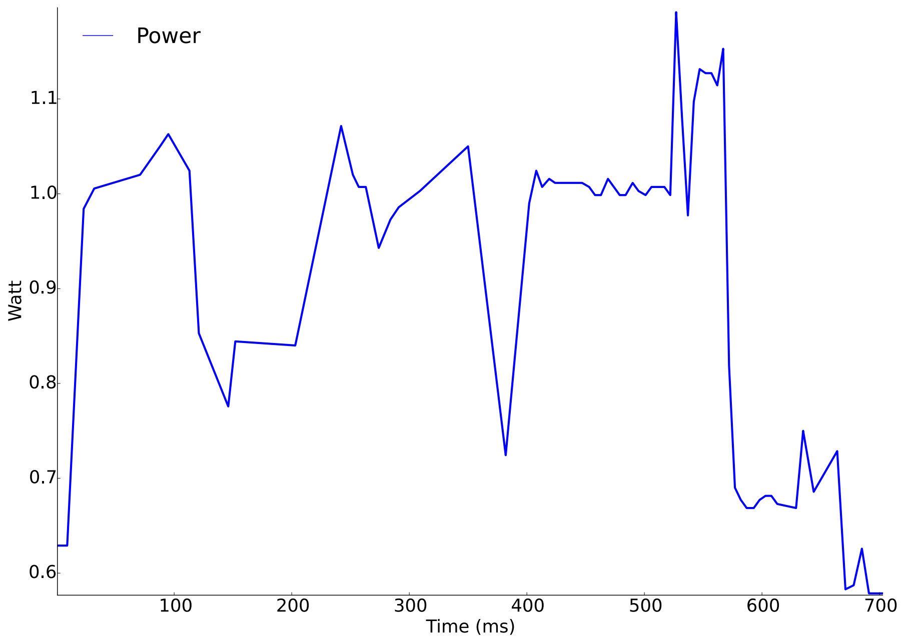


Figure 4: Example of power consumption for a Samsung GT-S7275R

The total energy consumption can be obtained by integrating power over time, as defined in Equation 3. As can be seen in Figure 4, the power that is used by the phone is discontinuous. Thus, a number of points are sampled for some time. These points are then used to approximate the integral numerically.

$$E = \int P(t)dt \tag{3}$$

3.4 Network protocols

This section covers the theoretical background of the network protocols used in this thesis. The data synchronization techniques derived from the protocols are also described.

3.4.1 TCP

The Transmission Control Protocol (TCP), is a network protocol for end-to-end communication that is implemented on top of the IP protocol. When a client wants to connect to another client through TCP, a three-way-handshake is used to establish the connection. TCP offers reliable data transfer which means that packets are delivered to an application in the order that they were sent, and lost or corrupt packets are re-sent until all data is delivered. TCP also implements flow control and congestion control, which are mechanisms that ensure that neither the network nor the end hosts are flooded with data. The protocol should be used in applications that require point-to-point communication where it is important that data is delivered reliably [38].

To further enhance the usage of reliable communication between two end hosts, two different data synchronization techniques can be used which are hereby referred to as TCP push and TCP pull. TCP push and TCP pull are often used in protocols that are built on top of TCP, such as HTTP, but can also be used directly with TCP [39].

With TCP push, a connection is always open and the end hosts can send data to each other instantly. As operating systems can close connections that are not used, so called heart-beat messages are sent periodically to tell the OS that the connection is still in use [40]. By using TCP push, an end host can send data in real-time to another host that is available, and operations that can be energy expensive such as establishing a connection is only done once. With TCP pull, an end host needs to periodically pull data from another end host by creating a new connection within a defined update interval. As a result, the connection is only open during data transfer and is then closed and disposed. Therefore, no data is sent or received between update intervals [39]. However, pulling data frequently might consume more energy due to the fact that many new connections need to be established.

3.4.2 UDP

The User Datagram Protocol (UDP), is a network protocol that is implemented on top of the IP protocol. UDP is connectionless, which means that no connection information is stored. There is no initial establishment and no acknowledgements are sent, thus no guarantees can be made that the data that is sent has arrived. UDP is therefore suitable for time-sensitive applications, as packet loss is better than violating the time constraints. Since there is no handshake and no re-transmission of data, UDP requires less packets than other protocols such as TCP. Less packets also means that data can be delivered faster through UDP [38].

4 Methodology

The project began with a literature study, where papers that covered topics of energy measurement and networking in mobile applications were read and reviewed. A selection of the most relevant papers to this thesis can be found in Chapter 2. From related papers, a table was compiled consisting of key elements used when measuring energy. The table can be found in Appendix A, and consists of platform, application type, and logging equipment.

4.1 Energy measurement approach

To determine a suitable approach the table in Appendix A was reviewed, to find common denominators in the various projects. A common denominator that was found was that all reviewed projects used the Android platform. The papers describe that Android was used since it is open source and thus modifiable, which in many cases suited the authors' needs. With this in mind, a comparison between an Android phone and an iPhone was made, and it was decided to use Android as platform for energy measurements in this project as well. This was due to the fact that the battery of an iPhone is integrated and not so easily removed, in combination with the iOS SDK only providing information about the battery percentage left. Combining the lack of energy measurement features with the restrictions of the platform described in Section 3.1.2, the iOS platform did not match the desired characteristics previously mentioned in Section 1.2.1. The Android phone, a Samsung GT-S7275R running Android 4.2.2, had a battery that could easily be removed and also contained a built-in current sensor. Together with a fine-grained

API for reading voltage, the Android phone did not only fit better for this thesis than the iPhone, but did also match the desired characteristics. Therefore, the results presented in this thesis are energy consumption logged in prototypes implemented with cross-platform frameworks compiled for Android only.

Another common denominator that was found was that logging equipment often consisted of expensive hardware components. No project logged energy through a software-centered approach based on a mobile app, but since both current and voltage values were accessible on the phone used for testing, such an approach would be possible by using the theory described in Section 3.3. Therefore, an energy logging application called PowerLogger was created, which is described in detail in Section 5.2. To assess whether the energy consumption values measured with PowerLogger were consistent with the results of a hardware-based approach, an Intab AAC-2 PC-logger provided by i3tex was used to check the validity of PowerLogger. By connecting the phone to the hardware logger as described in Section 4.1.1, a number of sanity checks were performed which are further described in Section 4.1.2.

4.1.1 Hardware logging

To measure energy through hardware, the setup shown in Figure 5 was used. From the positive terminal on the battery a wire was connected, and the end of the wire was connected to a 0.2 Ohm resistor. On the other side of the resistor, another wire was connected to the positive terminal on the phone. By putting a piece of tape between the positive terminal on the battery and the positive terminal on the phone, the current flowed through the resistor. A red and a black wire, representing positive and negative current, were soldered to the wires that were on either side of the resistor shown in Figure 5.

By connecting the positive and negative wires to the PC logger the voltage across the resistor can be measured. When the voltage across the resistor was known, Equation 2 in Section 3.3.1 describing current resistive sensing could be used to find the relationship between voltage, current, and resistance. As the resistance was a constant of 0.2, the current I was calculated by $I = V/0.2$. To match the logged values from software logging described in Section 5.2.1, a pair consisting of timestamp and current was extracted from the logger. The voltage was read from the internal voltage sensor in the phone, and by using Equation 1 in Section 3.3.1 the power was calculated.

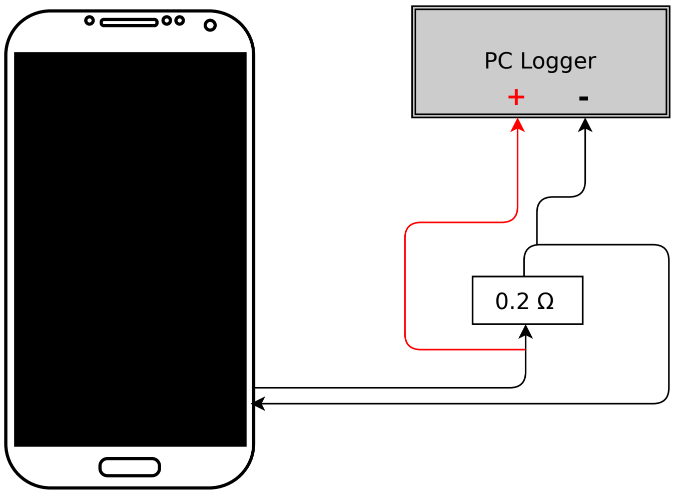


Figure 5: Hardware logging setup

4.1.2 Sanity checks

A number of sanity checks were performed to validate the results from PowerLogger. Each sanity check was performed by connecting the phone to a hardware logger as described in Section 4.1.1, and then starting the PC-logger and PowerLogger simultaneously. The PC-logger produced very fine-grained results but could only log one value per second. PowerLogger was set to log 20 values per second. One log record spanned a maximum of 95 seconds, as this was the maximum time the PC-logger could store values, and during the logging period a user interacted with the screen starting apps such as flashlight, web browser, and a map application. Five different logs were sampled and one of the logs, displayed as a graph, is shown in Figure 6. Each value in the hardware log corresponds to a value in the software log, but due to the hardware logger's lower log frequency the figure shows the hardware log (red line) as an average of the software log (gray line).

The total amount of energy for each software and hardware log was compared, calculated using the methodology described in Section 3.3.2. The results are shown in Table 1. The difference between the total energy consumption logged by the software and hardware-based approach was 0.1 mWh, with a maximum difference of 1.6 mWh. Comparing the difference of 0.1 mWh to the total energy measured by the software approach, the difference was approximately 0.1%. A fraction as small as 0.1% is small enough to be considered noise, which resulted in the sanity checks showing that the software-based approach worked just as well as the hardware-based approach.

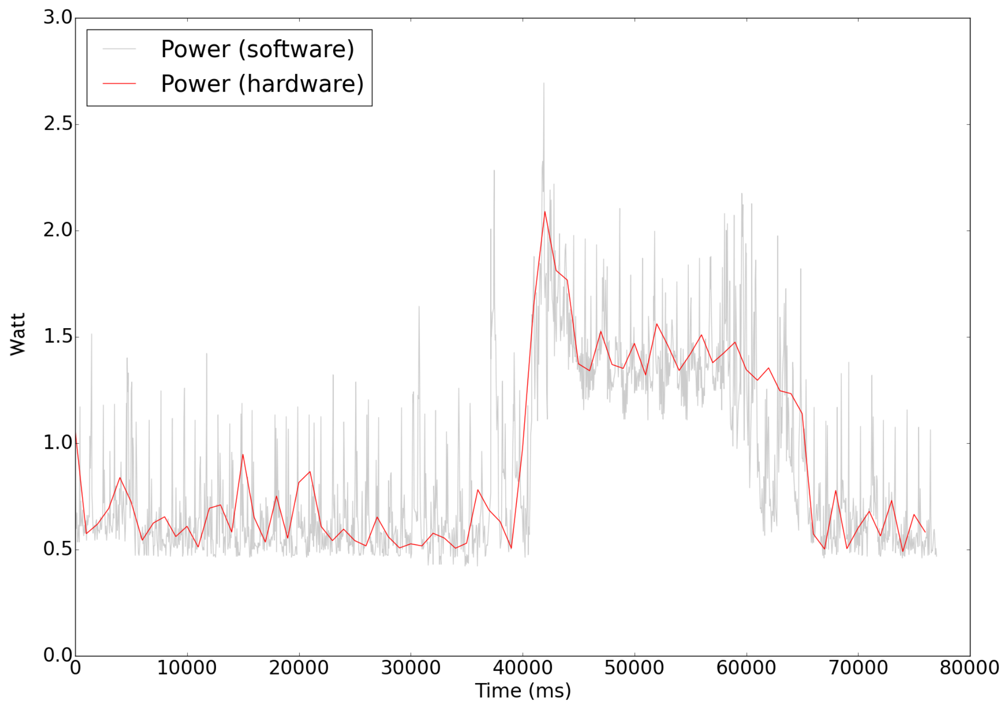


Figure 6: Sample log from sanity check

Table 1: Sanity check results

Log	Run time (s)	Software (Wh)	Hardware (Wh)	Difference (Wh)
1	95	0.0198974	0.0193471	+0.0005
2	92	0.0259503	0.0275873	-0.0016
3	76	0.0187794	0.0192907	-0.0005
4	88	0.0224319	0.0210263	+0.0014
5	90	0.0240841	0.0237465	+0.0003
Total	441	0.1111431	0.1109979	+0.0001

4.2 Implementation approach

The following sections describe the approach used when implementing the prototypes. In the first subsection, the data synchronization techniques UDP, TCP push, and TCP pull are discussed. Following this is a description of the implementation goals together with additional details of the implemented prototypes. In the last subsection, the consistency model that was used in each prototype is explained.

4.2.1 Data synchronization techniques

To be able to evaluate the purpose described in Section 1.1, data synchronization techniques were needed. The chosen techniques were TCP push, TCP pull, and UDP, which are described more in detail in Section 3.4.1 and in Section 3.4.2. These techniques were chosen based on their respective protocol's position in the OSI model as well as the API support that existed in the Android platform. Both TCP and UDP are located in the transport layer of the OSI model [41], which indicates that they are suitable for data transportation. The protocols of the network layer, the layer below the transport layer, did not provide functionality useful to any of the implemented prototypes and were thus not used. Many of the protocols in the layers above the transport layer are based on TCP or UDP, meaning that such protocols would have an added complexity overhead, making them unfit for comparison to TCP and UDP. Those protocols were therefore not used in this thesis.

4.2.2 Prototypes

Each implemented prototype was able to communicate with a data provider through either a TCP or UDP connection. The development was divided into three milestones that were implemented in consecutive order. Each milestone represented a major change in functionality which was used to investigate the correlation between different factors and energy efficiency. The milestones were also implemented to meet specified requirements from Volvo, which are available in Appendix B.

For the first milestone, the goal was to receive real-time data from a provider, without displaying the data. The goal of the second milestone was to display data received from a connection in text form. For the third and last milestone, the goal was to process the received data before showing it in the form of a graph. The reason for splitting the milestones into the specified prototypes was comparability. As each new milestone added an extra feature, differences in energy consumption could be linked to implementation changes.

To evaluate the question concerning whether different frameworks consume different amounts of energy, each prototype was implemented multiple times, each time with a different framework. The frameworks that were used were Ionic, Corona, and Xamarin. These frameworks were chosen mainly because of the fact that they all provide a way to write an application that can run on multiple platforms. Another aspect was based on the fact that, as described in Section 3.2, the frameworks use different techniques to achieve a cross-platform implementation. The ability to test different application creation techniques made it possible to investigate whether one framework was more energy efficient than another. A fourth framework, Appgyver Supersonic, did also match the desired characteristics but no implementation was completed using this framework. The reason for this was the tedious and complicated application build process, as well as problems with using third-party plugins that were required.

4.2.3 Consistency

To handle various network problems such as packet loss, a consistency model was created. The model was developed in collaboration with Volvo Cars and the focus was to provide as high usability to the end user as possible. As the data provider, described in Section 5.3, sent data points each 20 milliseconds the prototypes were classified as real-time streaming applications. As such, some packet loss was deemed to be acceptable as the data provider sent more data than an end user could possibly view. When a connection was lost the application was halted, and if a graphical user interface was implemented a blank screen was shown to indicate that the connection had been closed. As each prototype only received data, a closed connection signaled that the data provider had dropped the connection and needed to be restarted. Thus, no attempts to restart the connection were made from a prototype as the provider would be unreachable.

4.3 Acquiring the results

To evaluate energy efficiency, the energy measurement approach was applied to the implementation. Energy consumption was logged on a Samsung GT-S7275R running the Android OS, version 4.2.2. The key components of the phone are listed in Table 2.

The prototypes were split into as small pieces as possible to enable comparisons between different components, which is described in Section 4.3.1. The energy consumption of each component was logged by using the protocol described in Section 4.3.2. Finally, the factor that affected energy consumption the most was modeled using the previously logged data to gain a deeper understanding of its correlation to energy consumption.

Table 2: Samsung GT-S7275R hardware specification

Component	Specification
Chipset	Qualcomm MSM8930 Snapdragon 400
CPU	Dual-core 1.2 GHz Krait
GPU	Qualcomm Adreno 305
Memory	8 GB
RAM	1 GB
Display	LCD, TFT capacitive 480 $\times$ 800
WiFi	802.11 b/g/n 2.4GHz
GPS	A-GPS, GLONASS
Bluetooth	v4.0, A2DP
Radio	Stereo FM radio with RDS
Battery	1800 mAh (6,84 Wh), 3,8 V Li-ion

4.3.1 Prototypes

To investigate how different elements affect energy consumption, the mobile application was divided into a number of factors, which can be seen in Table 3. First, three different prototypes were developed: the base prototype, Prototype 1, and Prototype 2. Each prototype implemented a specific milestone, and did thus differ in functionality when compared with the other prototypes. The functionality difference depended on the complexity of the graphical user interfaces. Each prototype was split into three parts, where the different parts were defined by which framework was used. Lastly, each framework was implemented with the three data synchronization techniques TCP push, TCP pull, and UDP.

Table 3: Prototypes divided into different factors

Prototype functionality	Network only	Simple GUI	Complex GUI
Framework	Corona	Ionic	Xamarin
Synchronization technique	TCP push	TCP pull	UDP

To enable further comparisons, each data synchronization technique used a set of update intervals, which depended on the prototype. The interval values can be seen in Table 4. TCP pull used update intervals of 100, 500, and 1000 milliseconds regardless of prototype. The different update intervals were used to find a correlation between the frequency of performing a set of tasks and energy consumption. For the base prototype the defined task was to pull data, and for Prototype 1 and Prototype 2 the tasks were to both pull data and to update the graphical user interface.

Neither TCP push nor UDP used an update interval for the base prototype. As a TCP push or UDP connection is always open, data will be received without actively checking the connection. For Prototype 1, both TCP push as well as UDP used update intervals 100, 500, and 1000 milliseconds as delay intervals between updating the screen with received data. This was used to investigate the energy efficiency when adding a graphical user interface, and the different update intervals were used to see if redrawing data more or less frequently made a noticeable impact on energy consumption. For Prototype 2, TCP push and UDP updated the screen using intervals of 100, 500, and 1000 milliseconds, in order to compare the difference in energy consumption between the simple user interface of Prototype 1 with the more complex GUI in Prototype 2.

Table 4: Prototypes with techniques and update intervals

Prototype	Synchronization technique	Update intervals
Base	TCP push	N/A
	TCP pull	100, 500, 1000
	UDP	N/A
Prototype 1	TCP push	100, 500, 1000
	TCP pull	100, 500, 1000
	UDP	100, 500, 1000
Prototype 2	TCP push	100, 500, 1000
	TCP pull	100, 500, 1000
	UDP	100, 500, 1000

4.3.2 Energy measurement protocol

An energy logging protocol, based on the typical use of the prototypes, was created to be able to compare energy consumption. The protocol defined the screen to be on during a log session, and the brightness of the screen to be set to 50%. Each log file spanned 120 seconds with 20 logged power values per second, thus each log file contained 2400 power values.

The different prototypes were bundled as tuples where each tuple consisted of a prototype, a framework, a synchronization technique, and an update interval. The energy consumption for each tuple was logged ten times and was normalized by using the average from these ten different log files. The normalization removed energy spikes and noise, while still preserving the overall energy consumption.

4.3.3 Models

When the energy consumption of all prototypes had been measured, the most influential factor could be found. To gain a deeper understanding of how the energy consumption was affected, a number of models describing the correlation between the factor and the consumption were created.

Each model was based on a function with the influential factor as the key parameter. By plotting the measured energy consumption reported in Chapter 6, the characteristics of the data could be found. This was contrasted with the characteristics of different mathematical functions to find a correlation between the plotted data and a function. When a function was decided, the curve fitting tool available in MATLAB was used to find the coefficients. Finally, by comparing all models with each other it was possible to see if a general conclusion could be drawn about the factor's significance for energy consumption. The result from the model creation is shown in Section 6.3.

5

Design and Implementation

This chapter describes the design and implementation of the different prototypes, together with the chosen frameworks. Furthermore, the energy measurement app that was used is explained in detail. The chapter is concluded with a section describing the data providers sending real-time data.

5.1 Mobile application

The following sections describe the implementation and design of the three mobile application prototypes that were created during the course of this thesis. Each prototype received data tuples consisting of one friction value and one quality value from a data provider, where quality refers to the certainty that the friction value is correct. Both the data and the data providers are described in further detail in Section 5.3.

5.1.1 Architecture of the mobile application

The prototypes that were developed using Xamarin are modeled after the Model-View-Controller (MVC) architecture pattern. This is used to separate business logic and data, which is represented by the model, from the user interface, which is represented by the view. The controller handles input, notifies the model, and sometimes also changes the view. The prototypes that were implemented with

Corona were very simple with no particular architectural pattern utilized for their creation.

The Ionic framework is built around the JavaScript framework AngularJS which is designed using the Model-View-ViewModel (MVVM) pattern, which is derived from MVC. MVVM is focused on separating the UI code from the business logic of an application. This enables UX designers to create the design in a markup language, while other developers can create data bindings to the view-model. The view-model in AngularJS applications is a special object denoted as *$scope*. This was used in the prototypes to handle data by having the logic to receive data in the controller, and then push it to the views. The *$scope* object can also handle inputs from the views so that the model, in this case the controller, can be updated from the views.

5.1.2 Base prototype

The purpose of the base prototype was to implement an application which received real-time data from a data provider without parsing or displaying the data. As soon as data was received it was discarded. This made it possible to measure energy with emphasis on network data, without other energy consuming components such as graphics included. As an application always has a user interface, a black screen was shown to the user. The base prototype was implemented using three different frameworks: Ionic, Corona, and Xamarin. Each framework implemented the three different data synchronization techniques described in Section 4.2.1: UDP, TCP push, and TCP pull. However, given the difference in how the frameworks are designed the implementation differed for each one.

In order to get network socket support with Ionic, a third-party Cordova plugin was required. In Ionic it is not possible to access the sensors and other functions of the underlying platform without plugins. For the Xamarin version the network functionality was implemented using a third-party library that acted as an interface to Xamarin's native network socket support. Using the library makes it possible to write shared code which then calls the correct functions for the different platforms Xamarin supports. Corona provides an API for creating sockets as part of the framework, and thus no external libraries were needed.

5.1.3 Prototype 1

Prototype 1 was an extension of the base prototype, and was also implemented to receive real-time data from a data provider. The functionality of the prototype was fairly limited; the received data was parsed and displayed as text. Adding a graphical user interface made it possible to compare Prototype 1 with the base prototype, to determine the correlation between a user interface and energy consumption. Prototype 1 was implemented with the frameworks Ionic, Corona, and Xamarin, and each framework implemented different versions that used UDP, TCP push, and TCP pull.

Since the Ionic framework is based on web technology, the graphical interface was created using HTML and styled with CSS. By default, the framework provides a CSS file that can be used to further design the layout. The default styles can be overridden by using SASS, a CSS extension language, to customize the look of an app. For this project, SASS was only used to override the default background color to a dark one instead of the default white background. The rest of the interface was created by using Ionic's custom HTML components with their default styles.

The GUI in Xamarin was implemented using Xamarin.Forms, an API for creating native looking apps in C# with shared code. With Xamarin.Forms the entire graphical interface for Prototype 1 was written to be shared between Android and iOS without any platform specific modifications. The graphical interface in Corona was built through the already existing graphics API, which provided functionality to write text to the screen with a certain color at a specific coordinate.

User Interface

The main goal for Prototype 1 was to be able to print out any received data on the screen. Figure 7 shows the text view on a 7 inch tablet running Android 5.1. The values for the friction and the quality are displayed with different colors depending on how high the quality is. If the quality is low (less than 4), then the color of the text is changed to indicate that the estimation may not be accurate.

5.1.4 Prototype 2

In Prototype 2, a graph displaying friction and quality values parsed from the real-time data was added in order to measure energy consumption in a more complex graphical user interface. Both friction and quality values were shown on the y-axis, and each data point was tagged with a timestamp so that the x-axis of the graph showed the elapsed time since the application started. Prototype 2 was implemented with the framework Xamarin, and each version used UDP, TCP push, and TCP pull. The other two frameworks, Corona and Ionic, were deemed to be unfit for further use. As mentioned in Section 3.2.4, Corona is mainly

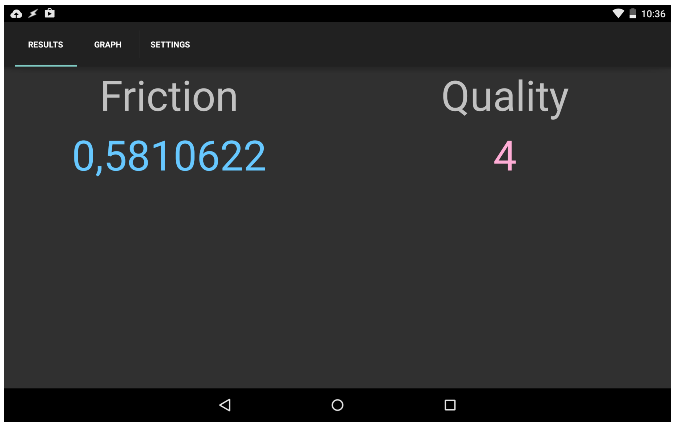


Figure 7: Prototype 1. A received data point from the data provider displayed as text.

designed for creating games, which turned out to be problematic for creating this prototype. The framework lacked many features for normal applications that the other frameworks provided. The reason to stop using Ionic was due to lack of documentation in graph libraries as well as time constraints.

User Interface

The graphical user interface of Prototype 2 was implemented using the cross-platform plotting library OxyPlot. OxyPlot supports a variety of different plotting tools empowering the developer to build complex charts. Some of the features that were used were a plot with different colors depending on which data was added, different axes that were connected to different graph lines, and an x-axis denoting time that removed old data points so that only the latest, most relevant data was shown. Using a cross-platform library made it possible to implement the graph in a shared project and then use it for both Android and iOS, without any platform specific modifications.

The user interface of Prototype 2 is shown in Figure 8, which is a snapshot of the prototype installed on a 7 inch tablet running Android 5.1. The blue graph line together with the right y-axis represents the friction, and the pink graph line together with the left y-axis represents the quality. If the quality decreases below a predefined threshold, both graph lines are shown in gray to indicate that the values are inaccurate.

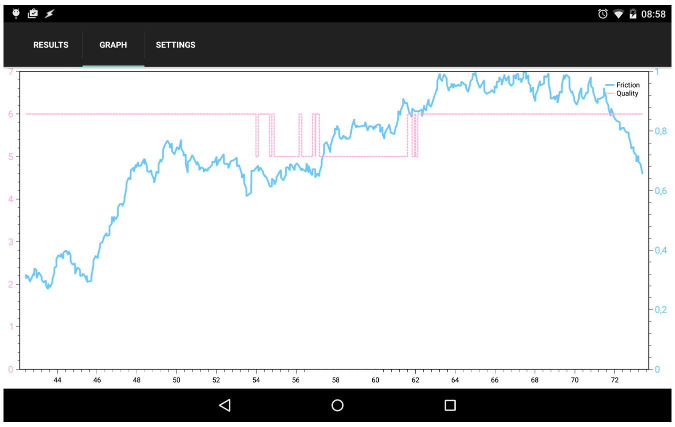


Figure 8: Prototype 2. Received data points from the data provider displayed in a graph.

5.2 PowerLogger

To be able to log energy through software, an application called PowerLogger was created. PowerLogger was divided into two different parts, one mobile application that logged power on a mobile device and one desktop application that processed the logged power information and calculated energy consumption. Splitting the application in two parts was done due to the fact that the mobile application should be as energy efficient as possible to remove as many sources of error as possible. Thus, the computationally heavy part where data is processed and evaluated was instead implemented in a desktop version. The mobile application was implemented in Java in order to run on the Android operating system. The desktop application was implemented in Python 2.7 in order to run on any operating system.

5.2.1 Functionality

The purpose of the mobile application was to log power on a mobile phone. The functionality was implemented by measuring current and voltage, which were used to calculate the power by using Equation 1 in Section 3.3.1. The voltage was retrieved from the Android system's battery manager and the current was retrieved by reading the output of a current sensor located inside the phone. The retrieved current, voltage, and power values, together with the system's timestamp and the battery percentage, were written to a log file. The logging process was automatically started and stopped when entering or exiting the main view.

In order to enrich the functionality and ease the use of the application, a few user-controlled settings were added. The application was configured so that the user could adjust the frequency with which the app retrieved values from the system. Monitoring energy consumption more frequently can yield greater accuracy but may also induce a larger energy overhead, so depending on use case a certain update frequency may be more or less suitable. Such a use case could for instance occur when measuring the energy of an energy efficient app. If PowerLogger's energy consumption accounts for a greater part of the energy consumption it can be difficult to distinguish how much energy that the other application consumes. Furthermore, the user could choose to log the currently running applications. This is useful when visualizing the log file, as an increase or decrease of energy consumption could be directly connected to a specific process.

The purpose of the desktop application was to process and visualize the log file created by the mobile application. The functionality was implemented by parsing the log and splitting the data into lists of the different logged types: date, current, voltage, power, battery percentage and running processes. The list of dates was recalculated into relative values, where each element represented a time difference from the start of the log file, measured in milliseconds. The recalculation enabled the possibility to create a more fine-grained log by aggregating multiple log files. When the log file was processed the desktop application calculated the total energy consumption in watt-hours by using the methodology described in Section 3.3.2 and printed it, together with the total running time, to the standard output. The energy data in the lists was used to create a graph where the x-axis represented the relative time since the start of the log. The corresponding values for current, voltage, and power were plotted in the graph.

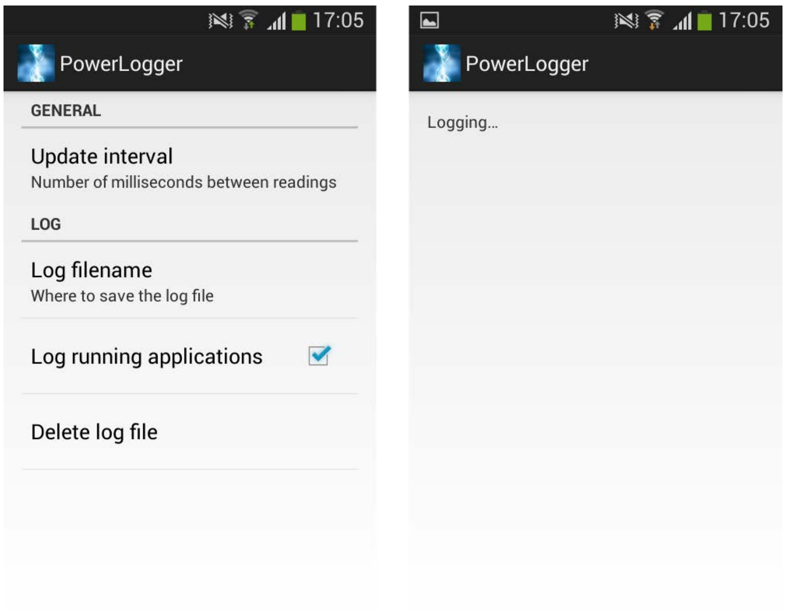
To enhance usability, a number of input parameters were added. The user could choose between plotting current, voltage, and power. A smoothing option, where the plotted values for a certain graph were smoothed using linear regression was also created. The smoothing factor, given as an integer value, determined the number of neighboring data points to use as the base when recalculating each data point. Finally, a choice to show different processes' runtime was implemented. The user was prompted to choose one or more processes that were running during the time the energy was logged. The chosen processes were added to a subplot below the main graph, so that a connection could be drawn between energy consumption and active processes.

5.2.2 User Interface

Both the mobile as well as the desktop part of PowerLogger implemented a graphical user interface. The interfaces, which were used to visualize the output of both application parts, are described in the following sections.

Mobile application

To minimize the complexity of the mobile application, the user interfaces were implemented with as few features as possible. This is shown in Figure 9. To the left in the figure is the settings view where the user can configure the functional settings described in the previous section. To the right in the figure is the main view, shown while the system is logging energy values.


17:05
PowerLogger
GENERAL
Update interval
Number of milliseconds between readings
LOG
Log filename
Where to save the log file
Log running applications ✓
Delete log file

17:05
PowerLogger
Logging...

Figure 9: User interface for the mobile part, with settings and main view

Desktop application

The user interface for the desktop application was provided through a terminal window. Using the parameters described in Section 5.2.1 the user could change the visual output of the program. One of the visual changes that could be made was to smooth a graph. Since a log file can contain many data points or a lot of noise, a smoothed graph can be used to give an overview of the data representation. This is shown in Figure 10, where the right graph is a smoothed version of the left graph. The log used to visualize Figure 10 uses input parameters that smooths the power values just enough to contain the main features such as maximums and minimums of the original graph, while still preserving the overall data distribution. The y-axis of the graph shows watt for the blue graph line, ampere for the green graph line, and voltage for the red graph line.

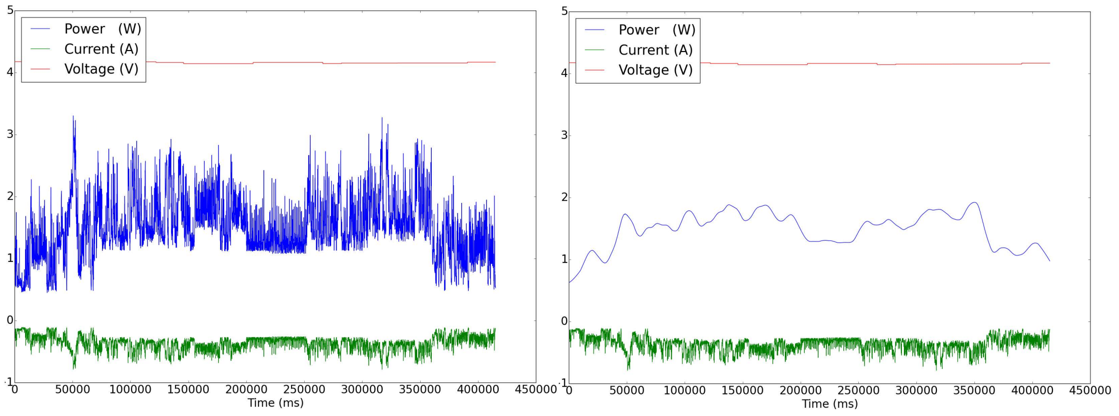


Figure 10: Output for original (left) and smoothed graph (right)

To find the correlation between a specific application and changes in energy consumption, a subplot showing running processes was added. This is shown with an example in Figure 11, where the subplot displays the process *com.devuni.flashlight* together with the power consumption. It can be seen in the same figure that the power consumption increased from approximately 1.6 to 2.2 watt as the same time as the process started. This fact could be used to draw the conclusion that starting the flashlight application momentarily increased the energy consumption with 0.6 watt.

5.3 Data provider

To be able to measure energy efficiency during data synchronization, two data providers were created, each sending data through a different network protocol. Both providers sent a pair of values: one floating point value representing friction data, and one integer value representing the quality of the data. The values were sent in JSON format, which resulted in a payload size of 38 bytes. Quality of data refers to the certainty that the data is correct. For example, a sensing application

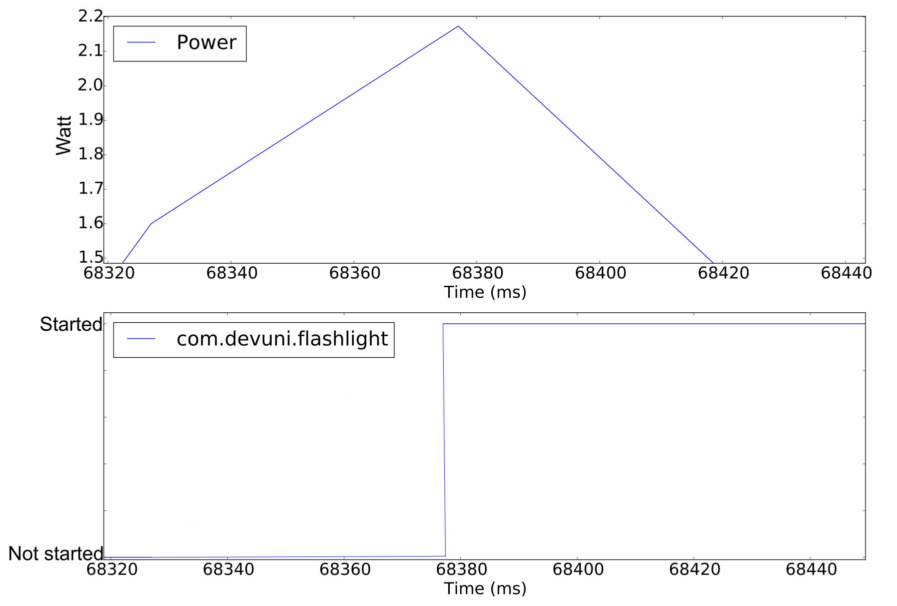


Figure 11: Relation between a running process and energy consumption

in a car could use a quality variable to define the accuracy of data, where good quality would represent data collected and verified by a number of sensors and bad quality would represent data that is unverifiable. The network protocols used by the data providers were TCP and UDP. The delay between each transmission was 20 milliseconds.

The provider that sent values through TCP was implemented in Java in order to run on any operating system. The TCP provider listened to a specific IP address and port, and when an application connected to the provider it started sending data. The initial data values were chosen at random. The quality was represented internally as a floating point value, which was rounded to the closest integer value when sent. At each transmission, a certain offset was added to both the data as well as the quality, to emulate a change in sensor data. The quality offset was chosen so that it would take a minimum of one second to increase or decrease the quality value by one. As the delay between each transmission was 20 milliseconds, the maximum value of the offset could be at most $20/1000 = 0.02$ seconds. Each quality offset was therefore chosen at random from a range between -0.02 and 0.02. For simplicity, the data offset was chosen from the same range.

The UDP provider, which sent data through UDP, was implemented in C++ in order to run in an embedded operating system. The provider was connected to a car to fetch friction and quality values from one or more sensors, but could also be connected to a computer via USB and in this case the data was emulated. Each data pair was sent to a specific port on the network broadcast address so that any device connected to the same network as the provider could read the data by listening to the specified port. The UDP provider was implemented and managed by Volvo.

6

Result

This chapter is structured as follows. The first section presents the energy consumption for different prototypes. In the next section, energy consumption values of elements in a mobile application are compared to find which factors influence energy consumption. This is followed by a section describing energy models of the most influential factor, which are created based on the measured data. Finally, the last section presents additional findings.

6.1 Energy measurement

In this section, the three subsections present and comment on the energy consumption in the different prototypes with focus on data synchronization techniques. The presented results are energy consumption logged in prototypes that are implemented with cross-platform frameworks and compiled for Android only, as described in Section 4.1. All energy consumption values are shown as average power consumption and thus the unit displayed in the tables is watt. The values are, as described in the energy logging protocol in Section 4.3.2, based on log files spanning 120 seconds. All values are normalized by taking the average based on ten different log files.

The reason for displaying energy consumption as average power is based on the fact that the time parameter is removed, and it is therefore easier to compare the values with each other as well as to other projects' values. The presented energy consumption is the energy used during the application's run time only, the consumption when starting and stopping an application where the user interacts with the screen has been excluded. The energy consumption for starting and stopping an application is instead described in Section 6.4.

6.1.1 Base prototype

For the base prototype, the energy consumption of the three different network techniques were logged. As described in Section 5.1.2, the prototype received data through a network connection, but the data was neither processed nor displayed. The techniques that were logged were UDP, TCP push, and TCP pull. TCP pull used three different delay intervals between pulling data: 100, 500, and 1000 milliseconds. As TCP push and UDP had constantly open connections they received one data point every 20 milliseconds, as described in Section 5.3. The prototype was implemented with three different frameworks: Corona, Ionic, and Xamarin. The result of the energy measurements for the base prototype is shown in Table 5.

Table 5: Energy consumption of the base prototype shown as average power (W)

	Corona	**Ionic**	**Xamarin**
Push	0.73481	0.72964	0.69697
Pull 100	0.76818	0.77152	0.74040
Pull 500	0.68540	0.67470	0.67945
Pull 1000	0.63377	0.62584	0.61706
UDP	0.65555	0.60019	0.56196

In Table 5, it can be seen that the most energy efficient synchronization techniques are TCP pull that fetches data every 1000 milliseconds and UDP, depending on which framework is used. The small deviations make it hard to know whether the frameworks implement the various techniques differently or if the difference is a coincidence. Therefore, both TCP pull 1000 and UDP are seen as equally energy efficient.

The reason for the low energy consumption for TCP pull which receives data once per second might be due to that each connection is closed and disposed when data transmission is finished. Thus, the mobile phone is able to sleep between updates and does not need to handle an incoming stream of packets as frequently as for instance TCP push. The low energy consumption in UDP is likely to be based on the fact that UDP is connectionless and has a smaller header to process.

Despite the fact that the frameworks sometimes differ in amount of consumed energy for a specific technique, TCP pull which pulls data every 100 milliseconds is consistently the most energy consuming data synchronization technique. TCP push is the second most energy consuming technique, consuming less energy than TCP pull that updates every 100 milliseconds but more than the other techniques. This might depend on the fact that TCP push maintains one open connection throughout the whole logging period, and as the data provider sends 50 data points each second, TCP push consumes more energy as it needs to process a lot of incoming packets.

6.1.2 Prototype 1

Prototype 1 was implemented with three different frameworks: Corona, Ionic, and Xamarin. For each framework three different synchronization techniques were used: UDP, TCP push, and TCP pull. Each synchronization technique used the update intervals 100, 500, and 1000 milliseconds. Due to the fact that Prototype 1 implemented a graphical user interface, the update intervals of TCP pull was used both for data retrieval and for updating the GUI. The update intervals in TCP push and UDP were solely used for updating the screen.

The result of the energy measurements can be seen in Table 6. Each row in the table represents the energy consumption of a synchronization technique implemented with different frameworks, and the columns of the table represent the energy consumption of a framework implementing different techniques. Lastly, the table is grouped by the delay between updates for different synchronization techniques.

From Table 6, it can be seen that TCP pull and UDP, both with an update interval of 1000 milliseconds, were the synchronization techniques that consumed the least amount of energy. Just like in the results from the base prototype, the most energy efficient technique differs depending on which framework is used. As described in Section 6.1.1, the reason for this difference can depend on the fact that an app created in one framework is implemented differently than an app implemented in another framework.

Independent of framework, the least energy efficient synchronization technique was TCP pull 100, and the second least energy efficient technique was TCP push 100. These results show the same relationship between different techniques as the results in the base prototype in Section 6.1.1. Thus, the results in Table 6 indicates that adding a graphical user interface consumes energy that is equivalent to some constant. The relationship between the different synchronization techniques in Prototype 1 is further visualized in Figure 12.

Table 6: Energy consumption in Prototype 1 shown as average power (W)

	Corona	**Ionic**	**Xamarin**
Push 100	0.83369	0.94376	0.86024
Pull 100	0.90548	0.98874	0.88863
UDP 100	0.75703	0.78720	0.74847
Push 500	0.74460	0.82022	0.74505
Pull 500	0.72988	0.80721	0.73136
UDP 500	0.66190	0.66480	0.67508
Push 1000	0.72524	0.77319	0.72266
Pull 1000	0.67058	0.67089	0.64412
UDP 1000	0.63884	0.63883	0.64756

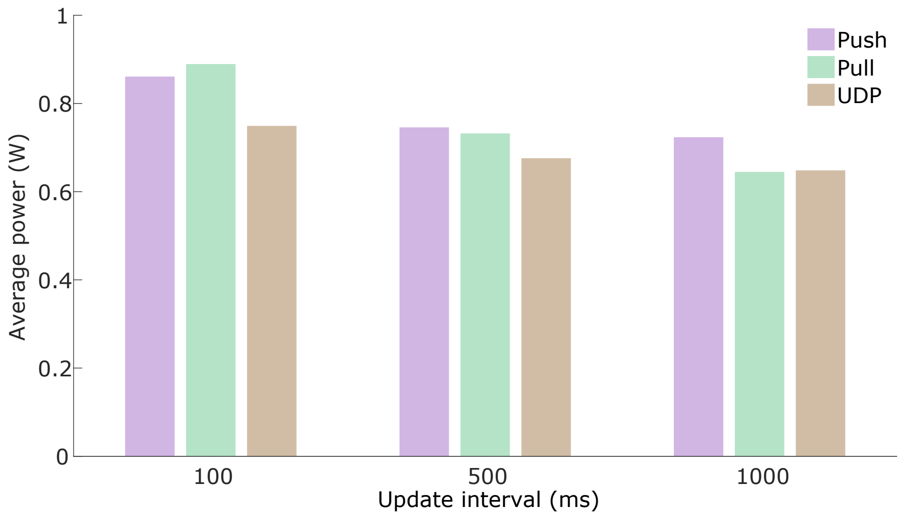


Figure 12: Relationship between synchronization techniques in Prototype 1

In Figure 12 it is shown that when the update interval is equal to 100 milliseconds, TCP push is more energy efficient than TCP pull. However, as the update interval increases TCP pull becomes more energy efficient than TCP push. When updating every 100 milliseconds, TCP pull needs to open and close a lot of connections while TCP push keeps an open connection. The action of opening and closing connections frequently can thus be explaining the difference in energy consumption. In the same manner, when the interval between updates increases TCP pull does not need to open as many connections. As TCP push still keeps an open connection it receives many more packets than TCP pull, which can explain the fact that TCP pull becomes more energy efficient than TCP push.

6.1.3 Prototype 2

The energy measurement was only performed for the Xamarin framework since it was the only framework that was used to implement this prototype, which is described in further detail in Section 5.1.4. Three synchronization techniques were used to retrieve data: UDP, TCP push, and TCP pull. Each synchronization technique used update intervals of 100, 500, and 1000 milliseconds. For TCP pull, the update intervals were used for both data synchronization and screen updates. For TCP push and UDP, the update intervals were used only to update the screen. The total energy consumption for Prototype 2 is shown in Table 7.

Table 7: Energy consumption in Prototype 2 shown as average power (W)

	Xamarin
Push 100	1.07767
Pull 100	1.13540
UDP 100	1.02482
Push 500	0.82110
Pull 500	0.80229
UDP 500	0.69578
Push 1000	0.72013
Pull 1000	0.65804
UDP 1000	0.62966

The results in Table 7 shows that UDP that updates the screen every 1000 milliseconds is the most energy efficient data synchronization technique. The least energy efficient synchronization technique is TCP pull which pulls data and updates the screen every 100 milliseconds. As the relationships between all techniques are consistent with the relations presented for the base prototype in Section 6.1.1 and for Prototype 1 in Section 6.1.2, the results indicate that the functionality added in Prototype 2 adds some constant energy. The results in Table 7 are further visualized in Figure 13. The colored bars in the chart represents the different synchronization techniques and the bars are grouped by the interval in milliseconds used to update the screen.

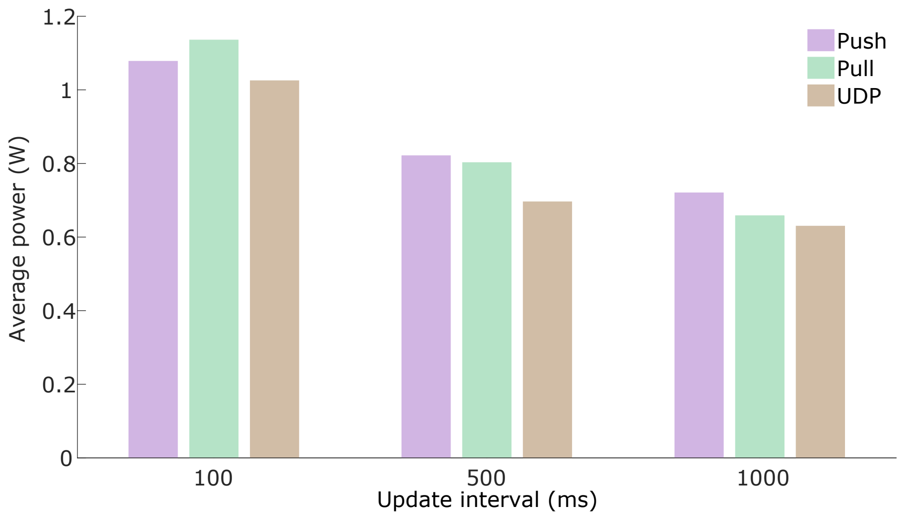


Figure 13: Relationship between synchronization techniques in Prototype 2

As can be seen in Figure 13, the main difference between the synchronization techniques is that when using an update interval of 100 milliseconds TCP push consumes less energy than TCP pull. When the interval between updates increases, TCP pull becomes more energy efficient than TCP push. This relation, which was also shown for Prototype 1 by using Figure 12, suggests that when retrieving data very frequently it is more energy efficient to keep a connection continuously open than to keep opening and closing it. It also indicates that when receiving data less frequently, it is more energy efficient to close each connection. Differences between TCP and UDP are possibly explained by the smaller overhead in UDP packets, which would mean that less energy is needed to process incoming data.

6.2 Factors affecting energy consumption

To further investigate the result from Section 6.1, different factors that might affect the energy consumption were compared. The factors, which were based on the general characteristics of a mobile networking application, were: synchronization technique, framework, and functionality. Functionality was implemented by graphical user interfaces of varying complexity. As both Prototype 1 and Prototype 2 implemented a graphical user interface on top of the network functionality, the energy consumption of the base prototype was subtracted from these prototypes to be able to compare the energy consumption of network techniques with the energy consumption of GUIs of varying complexity. Furthermore, the energy consumed when running only PowerLogger was determined to be equal to 0.51033 watt, and this constant value was subtracted from each data point to remove the energy overhead of PowerLogger.

An additional factor, update interval, was discovered during the course of the thesis. To provide a clearer picture, the update interval was converted into update frequency to see the number of updates per second instead of number of milliseconds between each update. Both the update frequencies for retrieving data as well as the update frequencies for updating the screen are investigated in Section 6.2.1. The comparison of the factors was done based on the results given in Section 6.1, and therefore each factor was contrasted against the network synchronization techniques. The results showed that the order of the factors, sorted from most influential energy consumption factor to the least, was: update frequency of graphical user interface, graphical user interface, data synchronization technique, and framework.

6.2.1 Update frequency

Figure 14 shows the relation in energy consumption between different factors and frequency. The blue line represents the network synchronization technique TCP pull, the red line the simple graphical user interface, and the yellow line the complex graphical user interface. TCP pull is shown in the figure since it is the only data synchronization technique that uses a frequency to retrieve data.

From Figure 14, several different relationships can be seen. The energy consumption of the network technique increases between values of 1 and 10 updates per second, but for frequencies higher than 10 the consumption levels off. As the difference in energy consumption between the lowest and highest frequency is small, it can be said that the frequency with which data is pulled does not influence the energy consumption very much.

This can be contrasted with the energy consumption of the GUIs of varying complexity. Adding a graphical user interface with a low frequency consumes very little energy but, as Figure 14 shows, an increase in update frequency largely impacts the energy consumption. This relation applies to both the simple as well as the complex GUI, with the difference that the complex GUI is more affected by the frequency than the simple GUI is. It can thus be concluded that the frequency with which a graphical user interface is updated influences the energy consumption more than just the interface itself.

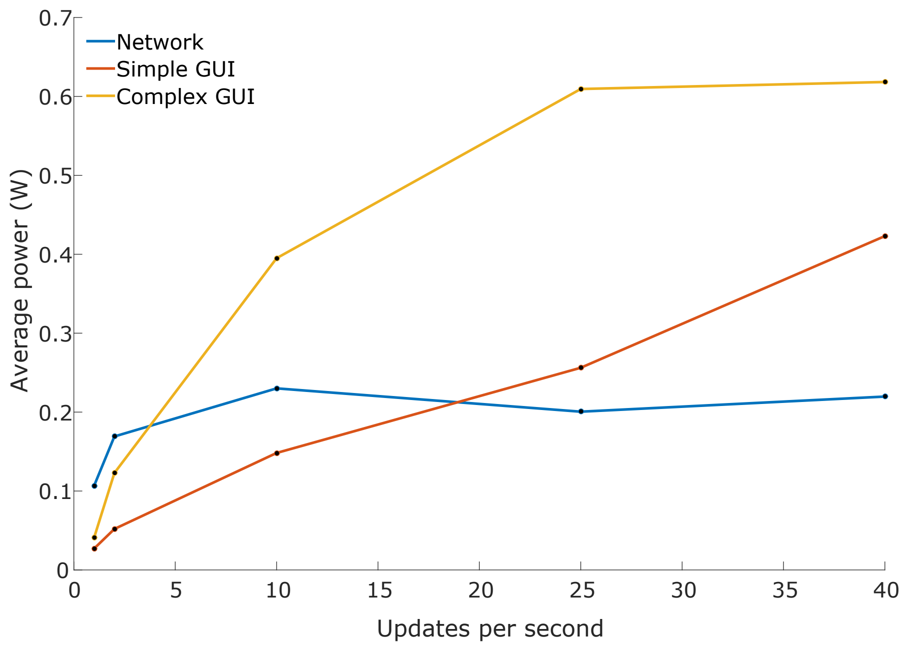


Figure 14: Difference in energy consumption between factors

6.2.2 Graphical user interface

As described in the previous section, the graphical user interface is a smaller factor than the GUI update frequency when considering energy consumption. Thus, the GUI update frequency has to be included when comparing the energy consumption of graphical user interfaces to the energy consumption of network synchronization techniques. However, as shown in Section 6.2.1, the update frequency of the synchronization techniques does not impact energy consumption very much. Therefore, the network frequency can be replaced with a constant indicating the energy consumption. This constant is the maximum energy consumption of all network techniques and all frequencies to consider the worst case. The energy consumption for graphical user interfaces and network synchronization techniques is visualized in Figure 15.

Figure 15 shows that determining which of graphical user interface and network synchronization technique that affects energy consumption the most depends on the frequency of the graphical user interface, as the frequency and GUI cannot be separated. Based on Figure 15, the complex graphical user interface consumes more energy than the network synchronization techniques when updating 5 or more times per second, and the simple GUI consumes more energy than the network techniques when updating more than 20 times per second. Since the energy consumption increases when considering varying complexity of graphical user in-

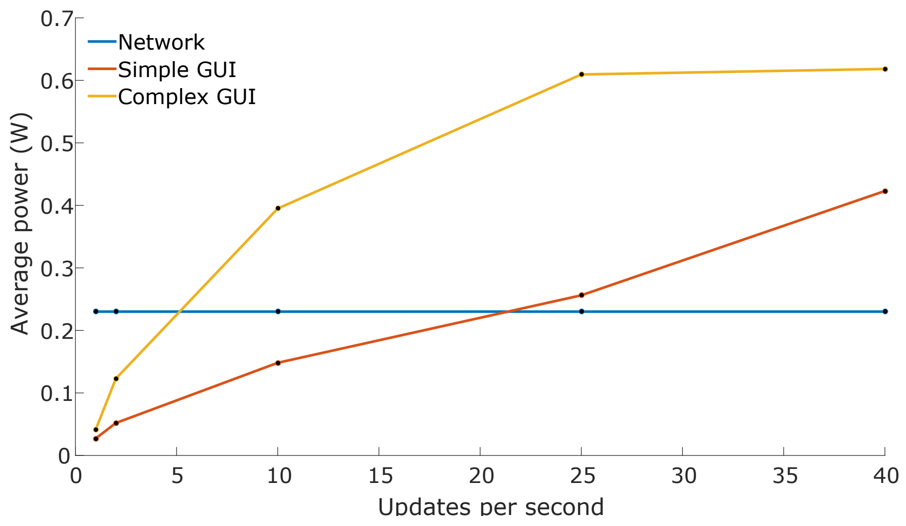


Figure 15: Energy consumption of graphical user interfaces and network techniques

terfaces and the energy consumption of the network techniques can be seen as a constant, the graphical user interface is a bigger factor of energy consumption than the synchronization technique is.

6.2.3 Frameworks

From the results in Section 6.1, it can be concluded that the differences between the frameworks are very small. This is further shown in Figure 16, where the data from the base prototype is plotted in a bar chart. The values on each bar represents the average power for a certain framework and a certain synchronization technique. The figure shows that there exists a difference in energy consumption between the frameworks. However, the numbers denoting the consumption indicate that the difference is insignificant.

Considering the worst case, the maximum difference between two frameworks in Figure 16 is 0.13 watt. This can be compared with the maximum difference for synchronization techniques for the same prototype. The data in Table 5 in Section 6.1.1 shows that the maximum difference ranges between 0.13 watt and 0.18 watt depending on which framework is considered. Thus, a conclusion can be drawn that the framework used to implement a mobile application is a smaller or equally big energy factor as the synchronization technique that is used to retrieve data.

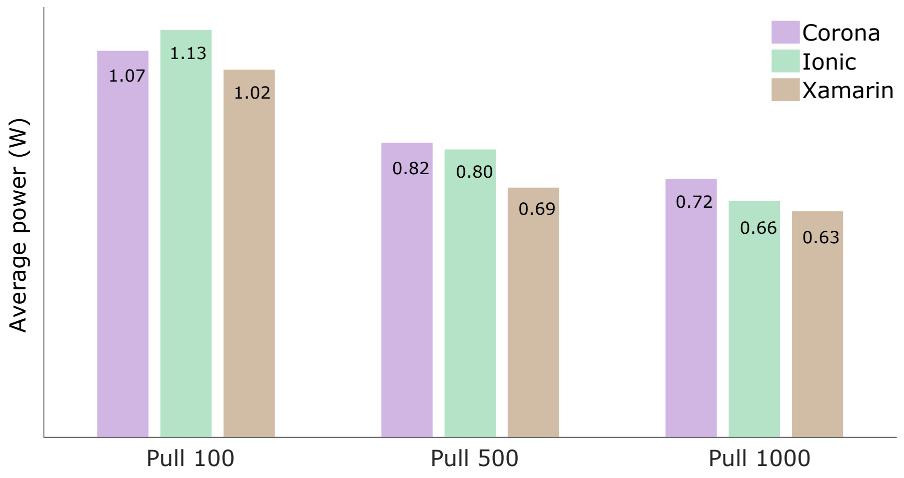


Figure 16: Energy consumption in different frameworks shown as average power.

6.3 Energy models

From the comparison between different influential factors in Section 6.2, it could be seen that the update frequency with which the graphical user interface was updated was the factor that affected energy consumption the most. To understand how update frequency influences energy consumption, different models were created. Each model was based on a mathematical function. The range of each model was given by the set of natural numbers larger than 0 but smaller or equal to 50, since it was not possible to measure 0 updates per second and 50 updates per second was the maximum update frequency due to 50 data points being sent each second.

By plotting average power consumption of a prototype based on different frequencies a pattern emerged which is visualized in Figure 17. In the figure it can be seen that the average power increases up until a frequency of 10 updates per second. When the frequency exceeds 10 updates per second, the average power stagnates. The figure shows data from Prototype 1 implemented with the Xamarin framework, retrieving data through TCP push. Each prototype that included an update frequency could be modeled in the same way.

$$P_{avg}(freq) = \begin{cases} a_1 * freq + b_1 & \text{if } freq \text{ is in the range of (0-10]} \\ a_2 * freq + b_2 & \text{if } freq \text{ is in the range of [10-50]} \end{cases} \tag{4}$$

The function describing the average power for a specific frequency is shown in Equation 4. The equation is split into two parts dependent on the frequency: one part for frequency values between close to 0 and 10, and a second part for frequency

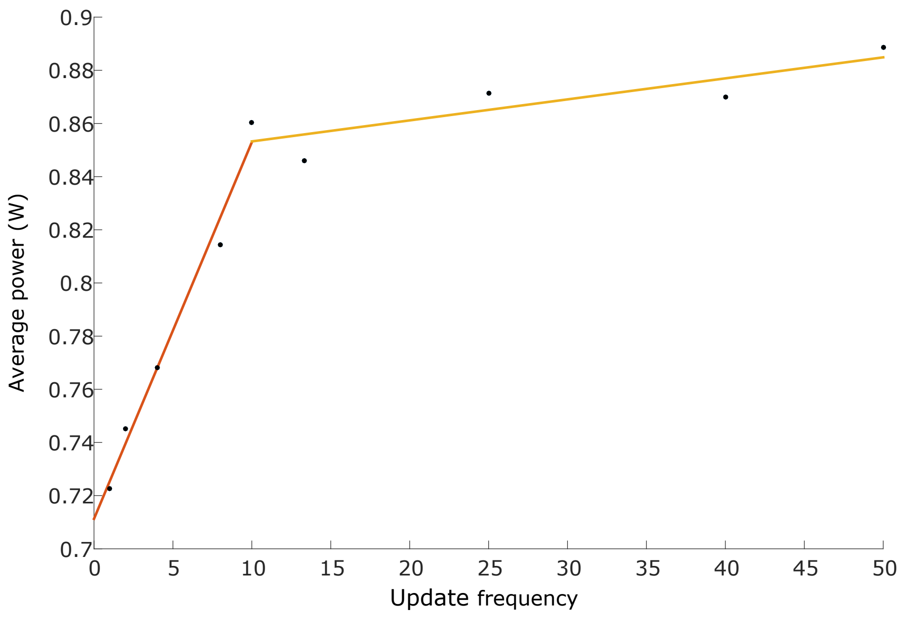


Figure 17: Average power for different frequencies

values between 10 and 50. From Figure 17, it can be seen that the values for a_1 is the slope of the first part and b_1 denotes the start of the first part. Similarly, as the second line is almost a constant, a_2 will approach 0 and b_2 will be a constant. This is further demonstrated by the values used to produce the figure. Here, a_1 is equal to 0.014, the slope of the first part, and b_1 is equal to 0.711, the starting value. Furthermore, a_2 is equal to 0.0007 which is a value approaching 0, and b_2 is equal to 0.845 which is approximately the value of all points of the second part. The a and b values for each model were found using the curve fitting tool in MATLAB.

Other functions were considered but did not model the system as well as Equation 4. One of the tested functions was a simple linear equation for all points. However, as the second part resembles a constant, a linear equation did not fit well to most of the data points. Instead, the linear equation gave a very brief overview of the model as a whole.

6.4 Additional findings

When the energy consumption of all prototypes was logged, different log files were plotted to find additional connections between prototypes and energy consumption. The connection that was found is shown in Figure 18, where the energy consumption data for the base prototype implemented with the Xamarin platform is plotted. The graph lines represent different techniques with different update intervals.

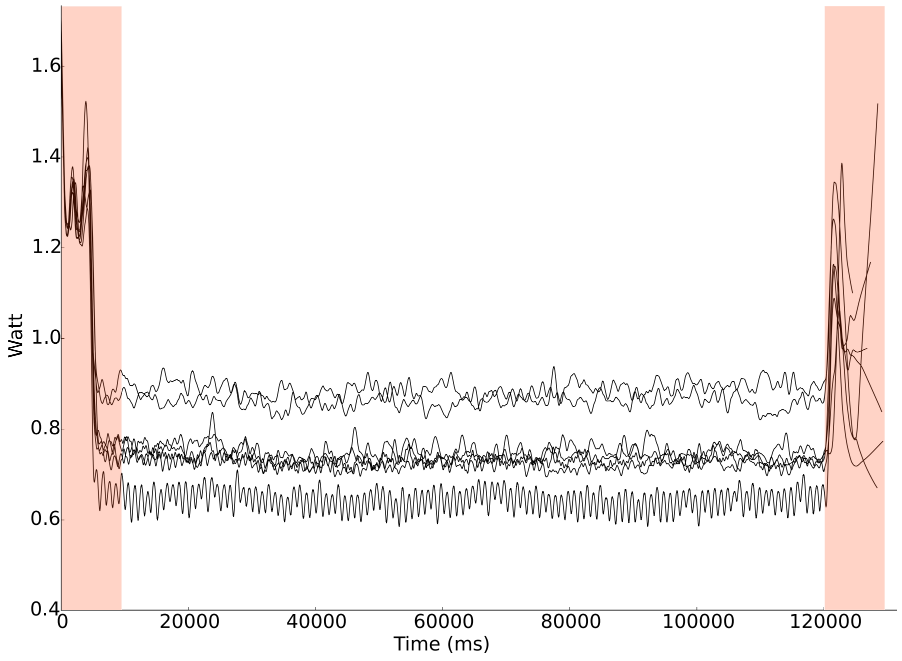


Figure 18: Start and tail energy

From the figure it can be seen that there are two clearly defined areas, the start and tail, which consume nearly the same amount of energy regardless of which technique or update frequency is used. The same pattern also appeared when plotting every prototype implemented with each framework. The reason for the start energy starting at a high value and then dropping might be due to the fact that the phone needs to load the application into memory, and then start it. This operation can be energy consuming, hence when the phone is finished starting the application the energy consumption decreases. The same reason might explain the tail energy; when the application is exited, the phone needs to stop the process by disposing old objects and stop running threads, and therefore needs to consume more energy. The only difference that was found was that different frameworks consumed different amounts of start and tail energy. Thus, these areas could be expressed as a constant value dependent on which framework was used. Table 8 shows the constant energy in watt-hours for starting an application in each framework, and also the total constant calculated by adding the start and the tail energy.

As Table 8 shows, the difference of the total constant between each framework is very small, making the constant negligible in normal use. This is further visualized

Table 8: Constant values for start and tail energy (Wh)

	Start	**Tail**	**Total constant**
Corona	0.00060	0.00172	0.00232
Ionic	0.00148	0.00144	0.00292
Xamarin	0.00195	0.00173	0.00368

in Figure 19, where the start and tail energy are stacked on top of run time energy. The run time energy is calculated by taking the average power of the base prototype synchronizing data through TCP push implemented with three different frameworks, and multiplying it with a specified time to convert watt into watt-hours. In Figure 19, the time that has been used is 120 seconds, which is equal to the time of a log file. As can be seen in the chart, the start and tail energy represented in the stacked bars accounts for a very small part of the total energy consumption, which strengthens the earlier statement that these values can be disregarded. As a side note, if the application would run for a very short time, the start and tail energy would account for a larger percentage of total energy consumption. For instance, if using the data in Figure 19, the application can run for a maximum of 18 seconds for the start and tail energy to be equivalent to the run time energy. However, as a run time as short as 18 seconds is not the normal case, it is still motivated that the start and tail energy are insignificant.

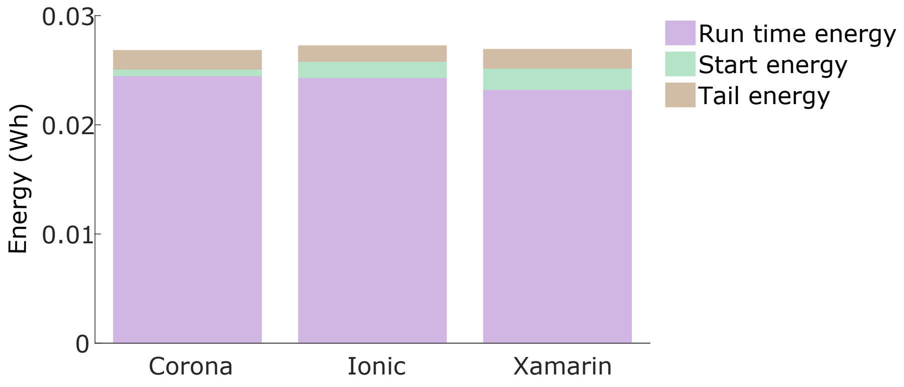


Figure 19: Start and tail energy stacked on run time energy

7

Discussion

In this chapter the results, mobile application, energy measurement approach, and the chosen frameworks are discussed and evaluated. The chapter is concluded with a section about possible future work.

7.1 Results

From the data presented in Chapter 6, two main results could be found; the relationships between different synchronization techniques and their energy consumption, and which of the four affecting factors influenced energy consumption the most.

The results showed two different relationships between synchronization techniques: UDP consumes less energy than TCP, and the difference between TCP push and TCP pull depends on the frequency with which data is pulled. Since UDP is a lighter protocol than TCP, with no connections, handshakes, or retransmissions, the energy consumption can be reduced since it does not need to process as much data as TCP does. As the only thing that is required from UDP is to process data received to a certain port, the results showing that UDP consumes less energy than TCP are well backed by theory.

The influential factors that were investigated, sorted from largest energy consumption factor to the smallest, are: update frequency of graphical interface, graphical user interface, data synchronization technique, and framework. These results are also well backed by theory.

The two largest impacting factors, the update frequency of a graphical user interface and the GUI itself, suggests that it is energy consuming to draw objects on the screen. The more often the screen is redrawn, the more energy is consumed, thus making the update frequency the larger of the two factors. The reason that the energy consumption depends more on the update frequency might also depend on the fact that when the mobile phone needs to operate more frequently it consumes more energy. A lower update frequency can thus reduce the energy consumption since the phone can be idle for longer periods of time. The results also showed that it was hard to separate the graphical user interface from the update frequency, as there was no use case when the GUI was static. Therefore, the comparison with the data synchronization techniques depended partly on the frequency. As the energy consumption of the data synchronization techniques could be seen as a constant and the energy consumption of the GUI was fluctuating depending on complexity, the graphical user interface was considered the bigger factor. Even so, there are simple graphical user interfaces updated with a low frequency that consume less energy than some network synchronization techniques do.

The third factor of energy consumption, data synchronization techniques, did not vary much. The only small difference that could be found was that the energy consumption increased when pulling between 1 and 10 times per second. However, as the differences were so small, the energy consumed when retrieving data is negligible compared with the graphical user interface.

Lastly, the factor that mattered the least was which framework that was used. The differences in energy consumption between the three used frameworks were minimal, and the only observation that was made was that the Ionic framework used a bit more energy than Corona and Xamarin when a graphical user interface was added. This could indicate that frameworks targeting web technologies use more energy when displaying data, or it could indicate that since the Ionic framework was a beta release during the development period it consumed more energy. Both the Corona and Xamarin versions were major releases and could thus have more optimized processes for handling energy consuming GUI components. It could also be interesting to compare the frameworks used in this thesis with a native framework, and see if there is a difference in energy consumption.

The most influential factor was used in Section 6.3 to create simple models, in order to further investigate how energy consumption was affected by the factor. The models showed that the largest factor, the update frequency of the graphical user interface, affected the consumption the most when updating the screen between 1 and 10 times per second. With frequencies bigger than 10 times per second, the energy consumption was close to a constant. The characteristics of these models are reasonable and indicates that when updating 10 or more times per second the phone needs to operate so often that it never goes to sleep. Thus, the energy consumption is fairly the same for all such frequencies. However, when the update frequency is bigger than 0 but smaller than 10, the phone is able to sleep between updates and does thus not consume as much energy. Therefore, a change in frequency in the range between 0 and 10 impacts energy consumption more than it does for frequencies where the phone never sleeps.

As the results presented in this thesis were collected using a software-based approach, it might be hard to compare the obtained values with the energy consumption shown in other papers. All other reviewed projects have used a hardware-based approach and do therefore not contain the energy overhead induced by an energy logging app. However, by removing the static energy of the PowerLogger like in Section 6.2.1, the energy consumption differences can be discussed.

Though the projects differ in payload size and data delivery, the results presented by Burgstahler et al. [13] are of the same magnitude as the ones presented in this thesis. Both this thesis and their paper note that using a pull-based approach is slightly more energy efficient than a push-based approach when using a delay interval of approximately 1 second between data retrieval. The work performed by Carvalho et al. [4] uses a longer logging period, encrypted packets, and sends data less frequently, and therefore the energy consumption values are not directly comparable to those presented in this thesis. However, the results of this thesis and their paper show that the difference between TCP push and TCP pull decreases as the delay between data pulls increases. Both projects conclude that if new data needs to be synchronized often, TCP push is the most energy efficient technique, otherwise TCP pull is the better choice. Differences in energy consumption between the projects also indicate that there might be a correlation between the amount of data that is sent and the threshold in time for when TCP push or TCP pull is more energy efficient. This could explain the fact that the project by Carvalho et al., which uses less data synchronized less frequently, has a higher threshold value.

7.2 Cross-platform frameworks

As previously mentioned in Chapter 5, three frameworks were initially reviewed and tested. As this project required socket support and the ability to show a graph, some other frameworks were deemed to be unable to fulfill the requirements. They lacked either required functionality or were not documented enough which made it difficult to know what could be accomplished by using them. Many of the cross-platform frameworks rely on web technology such as HTML, CSS, and JavaScript and for this thesis, Ionic was selected since it seemed to have a better community support and documentation than other frameworks.

Despite being in beta during the time of this thesis, Ionic seems to be one of the most mature frameworks that uses web technology to create mobile apps. It has one of the more active development teams and the utilization of plugins is easier than in other frameworks. There is no built-in support for creating graphs so in order to implement such a feature, developers are required to use third-party graph libraries or build their own. For this project's application, several different graph libraries were tested. However, due to lack of documentation of the libraries it was difficult to create the type of graph needed for the mobile application. With this being said, for developers that do not need specialized plugins or libraries Ionic could be a viable alternative to native development for creating apps.

Xamarin was the framework that best suited the requirements for the application. It is well documented and has a large user community. As described in Section 3.2.3, Xamarin can bind directly to a platform like Android which means that native features can be used. Thus, Xamarin applications can use features that applications written with other frameworks need third-party plugins for. The downside is that while many other cross-platform frameworks are free, Xamarin requires users to subscribe to one of their paid plans if they want to create relatively big apps. There is a free plan but it is limited in the size of the application that can be compiled and no external libraries can be added. Furthermore, users of the free plan cannot use Xamarin Forms, a GUI framework that simplifies implementation of graphical components for Xamarin's supported platforms. The company offers a free student license which was used in this project, but for companies and other developers the license cost is something that needs to be taken into account when deciding if Xamarin should be used for a project.

The Corona framework was very convenient to work with as examples and APIs were well described. The framework provides many built-in features such as networking and graphics which together with Lua, the programming language used in the framework, makes retrieving and displaying data from a data provider easy. However, as mentioned in Section 3.2.4, all graphical elements in an application needs to be positioned at a specific coordinate on the screen which makes programming the graphical interface a tedious task. Adjusting the graphical interface to fit devices with different screen sizes is also an issue that made app implementation complicated.

To conclude this subsection, all frameworks that have been tested have their own strengths and weaknesses. As this thesis considered a real-time streaming app that required a graph, neither Ionic nor Corona were the best choice of framework.

However, for other types of applications, such as an application showing web-based content or a game, Ionic or Corona would fit well. The Xamarin framework was the best framework for this type of project but with another type of project it might also lack important features, thus making it unfit for use. A suggestion to other developers is therefore to choose a framework that supports sought-after features, since all frameworks have their pros and cons.

7.3 Measuring energy

When measuring energy in a complex system, no matter which approach is used, there will always be a problem of determining which process uses what part of the total energy consumption. As described in Section 3.1.1, the Android operating system supports multitasking and background threads which makes it hard to pinpoint the exact amount of energy a specific app consumes. In some of the research papers described as related work in Chapter 2, as well as in this thesis, differences in energy have been used to draw the conclusion that one application uses more or less energy than another. An issue with this approach is that the Android system itself can start processes such as other applications or alarms, which induce extra energy.

There are two ways of removing the energy overhead of extra processes. The first approach solves the issue by letting the energy logger be aware of which processes are running in order to detect changes. Whenever a new processes is started during a logging session, the logged values are considered invalid as it is impossible to know which process consumed which amount of energy. The second approach is to shut off as many applications as possible, and cancel alarms and background processes to minimize the risk that a process is started. By logging several times, the data in the logs can be normalized by either taking the median or average value, which removes extra energy induced by other processes that might have been started. The second approach, which was used in this thesis, produces results that can be easily verified. By looking at which processes are running while logging energy as well as plotting each individual log file, it can be seen that the values are of the same magnitude and shape in all log files. The downside is that to be able to normalize the log files, a lot of energy measurement logs are required. Thus, the first approach might be faster in terms of hours spent on logging, but the verifiability of the second approach made it more fitting for this thesis.

As stated in Section 3.1.1, not all Android phones provide the same functionality. Measuring energy through software might therefore be possible on one device but not on another, since not all devices have a built-in current sensor. As of Android

5.0 the SDK provides an API for retrieving current, enabling easy access for developers to create an energy logging application. However, without a current sensor the value provided by the API will always be 0, yielding the logging application useless.

Another issue when logging energy is how fast the used measurement technique is. In this thesis an update frequency of 20 times per second has been used for the energy logger. As the maximum update frequency of each prototype is 10 times per second, the energy logger could log energy at least twice as fast as the prototypes received data. However, since TCP push and UDP have constantly open connections these techniques receive data 50 times per second, which means that the energy logger does not capture all packet receiving events. When deciding upon the update frequency for the energy logger, a few tests were run to see if it would make a difference logging energy 20 or 50 times per second. The outcome of the tests was that the energy consumption is the same regardless of frequency. As there was no difference between the frequencies plus the fact that, as stated in Section 5.2.1, a higher update frequency might consume more energy, the lower frequency was chosen. An additional detail is that since it was only the relationship in energy consumption between different prototypes and not the exact values that were compared, a higher update frequency would be required if it had yielded a different relationship.

Related to this is the issue described in Section 4.1.2, where the PC-logger was used to show that the software-based energy logger works just as well as a hardware-based approach that updates once per second. A hardware logger that is able to log more frequently would have been preferred, but since fast, fine-grained energy loggers are often very expensive, the only machine that was available was the one used in this thesis. A fast hardware logger would have enabled the possibility to create energy consumption models for smaller parts of a mobile application. For instance, as mentioned in Chapter 2, Rice et al. [2] were able to log the energy when receiving a single packet. It would also have been possible to log tail energy of the WiFi module or the energy consumption when the screen was off. Such models, which are not possible to create with PowerLogger, could have provided more reference values that could have been compared with the energy consumption of the prototypes with. Moreover, as previously mentioned in Section 5.2.1, extra applications that are running, such as the PowerLogger, induce an energy overhead. Thus, the only conclusion that can be drawn is if one prototype uses more energy than another. With a hardware logger capable of logging more frequently, the energy consumption of the prototypes could have been measured with more certainty and it would have been easier to precisely define the amount of energy each prototype used.

7.4 Mobile application

The mobile application prototypes in this thesis were developed with the sole purpose of measuring energy consumption. Thus, the code was split into parts that were easily divided into different prototypes. However, when developing an application without energy efficiency in mind, it can be hard to extract and compare different prototypes the way it was done in this thesis. Without the extraction it would therefore be hard to know which elements of an application should be improved to decrease energy consumption.

Another issue with the mobile application that can be improved is the consistency model. As this model was developed together with Volvo, improvements could only be made when both parties had time. The advantage of the simple consistency model that is used is that the application does not need to keep track of consistency variables. One of the desired features in the model was UDP data with timestamps so that the data can be presented in the order that the packets are sent. However, without this feature it was easier to log energy, as there was no need to take the consistency variables into account when investigating the energy consumption.

7.5 Future work

The work in this thesis provides a good foundation for further work. As previously mentioned in Section 7.3, it would be interesting to log energy with a fast and fine-grained hardware logger to retrieve the energy consumption of more parts of a mobile application. The energy consumption of such components could then be used to find which components account for a larger amount of consumed energy. Both Rice et al. [2] and Xu et al. [8] are able to log energy in WiFi with a logger precise enough to see the start and tail energy of the WiFi module. Being able to log the different states of the WiFi could provide a detailed insight about the energy consumption in different network protocols.

Other components of the phone could also be monitored to find more factors of energy consumption. One such component is the CPU, which depending on clock frequency might consume more or less energy. By logging energy consumption of the prototypes with different clock frequencies, such a relationship could be investigated. Another parameter that could be changed is the screen brightness, to determine if this is a significant factor for energy consumption. Lastly, there are variables in the prototypes that could be changed to further model energy consumption. For instance, the size of the data or the frequency with which data is sent could be changed to find correlations between different parameters.

8

Conclusion

This thesis investigated energy consumption of different synchronization techniques when retrieving data from a network through WiFi. The different techniques were compared with each other to find which technique was the most energy efficient. The techniques were also compared with other factors, to assess whether network techniques consume more or less energy compared with other elements in a mobile networking application.

The energy consumption was logged through a software-based approach, where a mobile application measured energy used by other applications. The difference between using a software-based approach compared with a hardware-based approach was also discussed, and the difference revealed that logging through software works just as well as through hardware. To measure energy consumption, three different mobile application prototypes were developed. All prototypes were implemented with different functionality, allowing for a comparison of energy consumption between different elements in a mobile application. The base prototype retrieved data without displaying it, Prototype 1 both retrieved and displayed data, and Prototype 2 retrieved, processed, and displayed data.

In the result, it was shown that using a network protocol with a smaller header consumes less energy, as lesser data might mean lesser processing time and thus more time for the components in the phone to sleep. It was also shown that when synchronizing data frequently, using a constantly open connection is the most energy efficient choice of technique compared with techniques that open new connections. However, when the frequency of creating new connections decreases it is more energy efficient to close the connections. A possible reason could be that the energy consumption when receiving and discarding packets is larger than the consumption when starting and closing a connection.

Furthermore, it was concluded that the update frequency with which an application displays data is the biggest energy consumption factor. The less frequently an application updates the screen, the more energy is saved. Drawing a graphical user interface on the screen is the second biggest factor, followed by network techniques. The least important factor is which cross-platform framework is used. It was also discussed that other parameters such as packet size or the delay with which packets are sent might affect the energy consumption, and that future work will have to conclude if this is the case.

References

[1] J. F. Dimarzio, *Android: a programmer's guide.* Blacklick, OH, USA: McGraw-Hill Osborne Media, 2008.

[2] A. Rice and S. Hay, "Measuring mobile phone energy consumption for 802.11 wireless networking," *Pervasive and Mobile Computing*, 2010.

[3] D. Li, S. Hao, W. Halfond, and R. Govindan, "Calculating source line level energy information for Android applications," in *The 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*, 2013, pp. 78–89.

[4] S. A. L. Carvalho, R. Lima, and A. Silva-Filho, "A Pushing Approach for Data Synchronization in Cloud to Reduce Energy Consumption in Mobile Devices," in *Brazilian Symposium on Computing System Engineering (SBESC)*, 2014.

[5] T. Zhang, S. Madhani, P. Gurung, and E. van den Berg, "Reducing energy consumption on mobile devices with WiFi interfaces," in *Global Telecommunications Conference*, nov 2005, pp. 561–565.

[6] International Data Corporation. (2015) IDC: Smartphone OS Market Share 2014, 2013, 2012, and 2011. [Online]. Available: http://www.idc.com/prodserv/smartphone-os-market-share.jsp [Accessed: 2015-01-29]

[7] A. Rodrigues Tonini, L. M. Fischer, J. Balzano de Mattos, and L. Brisolara de Brisolara, "Analysis and Evaluation of the Android Best Practices Impact on the Efficiency of Mobile Applications," in *Brazilian Symposium on Computing Systems Engineering (SBESC)*, dec 2013, pp. 157–158.

[8] F. Xu, Y. Liu, T. Moscibroda, R. Chandra, L. Jin, Y. Zhang, and Q. Li, "Optimizing background email sync on smartphones," in *11th annual international conference on Mobile systems, applications, and services*, Taipei, Taiwan, 2013, pp. 55–68.

[9] Y. M. Kwete, "Power consumption for iOS," Master's thesis, North Dakota State University, Fargo, North Dakota, USA, oct 2013.

[10] Apple Inc. (2015) Analyzing CPU Usage in Your App. [Online]. Available: https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/AnalysingCPUUsageinYourOSXApp/AnalysingCPUUsageinYourOSXApp.html [Accessed: 2015-01-29]

[11] D. Li, S. Hao, J. Gui, and W. Halfond, "An Empirical Study of the Energy Consumption of Android Applications," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, oct 2014, pp. 121–130.

[12] J. McWherter and S. Gowell, *Professional Mobile Application Development.* Somerset, NJ, USA: John Wiley & Sons, 2009.

[13] D. Burgstahler, N. Richerzhagen, F. Englert, R. Hans, and R. Steinmetz, "Switching Push and Pull: An Energy Efficient Notification Approach," in *IEEE International Conference on Mobile Services (MS)*, june 2014, pp. 68–75.

[14] C. Thompson, D. Schmidt, H. Turner, and J. White, "Analyzing mobile application software power consumption via model-driven engineering," Master's thesis, Vanderbilt University and Virginia Polytechnic Institute and State University, Nashville, TN and Blacksburg, VA, USA.

[15] S. Hao, D. Li, W. Halfond, and R. Govindan, "Estimating Android applications' CPU energy usage via bytecode profiling," in *First International Workshop on Green and Sustainable Software (GREENS)*, june 2012, pp. 1–7.

[16] S. Hao, D. Li, W. Halfond, and R. Govindan, "Estimating Mobile Application Energy Consumption Using Program Analysis," in *The 2013 International Conference on Software Engineering*, 2013, pp. 92–101.

[17] Y. Liu, C. Xu, and S.-C. Cheung, "Diagnosing Energy Efficiency and Performance for Mobile Internetware Applications," *Software, IEEE*, vol. 32, no. 1, pp. 67–75, Jan 2015.

[18] Encyclopædia Britannica Online. (2015) Android 2015. [Online]. Available: http://academic.eb.com.proxy.lib.chalmers.se/EBchecked/topic/1483582/Android [Accessed: 2015-02-12]

[19] D. Morill. (2008, sept) Announcing the Android 1.0 SDK, release 1. [Online]. Available: http://android-developers.blogspot.in/2008/09/announcing-android-10-sdk-release-1.html [Accessed: 2015-02-26]

[20] S. Allen, V. Graupera, and L. Lundrigan, *Pro smartphone cross-platform development: iPhone, BlackBerry, Windows Mobile, and Android development and distribution.* New York, N.Y., USA: Apress, 2010.

[21] J. Nutting, F. Olsson, D. Mark, and J. LaMarche, *Beginning iOS 7 Development: Exploring the iOS SDK.* Berkeley, CA, USA: Apress, 2014.

[22] J. Bucanek, *Learn iOS 8 App Development.* New York, N.Y., USA: Apress, 2014.

[23] J. A. Brannan, S. Weber, and B. Ward, *iOS SDK Programming: A Beginners Guide.* Blacklick, OH, USA: McGraw-Hill Osborne Media, 2011.

[24] Apple. (2015) Common App Rejections. [Online]. Available: https://developer.apple.com/app-store/review/rejections/ [Accessed: 2015-04-22]

[25] S. Dhillon and Q. H. Mahmoud, "An evaluation framework for cross-platform mobile application development tools," *Software: Practice and Experience*, 2014.

[26] O. Cinar, *Android native development using eclipse.* Berkeley, CA, USA: Apress, 2012.

[27] Apache. (2015) Apache Cordova. [Online]. Available: http://cordova.apache.org/ [Accessed: 2015-02-22]

[28] Drifty. (2015) Ionic: Advanced HTML5 Hybrid Mobile App Framework. [Online]. Available: http://www.ionicframework.com/ [Accessed: 2015-03-30]

[29] Appgyver. (2015) Supersonic. [Online]. Available: http://www.appgyver.com/supersonic [Accessed: 2015-02-22]

[30] Appcelerator Inc. (2015) Titanium Mobile Application Development | Appcelerator Inc. [Online]. Available: http://www.appcelerator.com/titanium/ [Accessed: 2015-02-22]

[31] A. Charland and B. Leroux, "Mobile Application Development: Web vs. Native," *Commun. ACM*, vol. 54, no. 5, pp. 49–53, may 2011.

[32] J. Bristowe. (2015, March) What is a Hybrid Mobile App? Telerik. [Online]. Available: http://developer.telerik.com/featured/what-is-a-hybrid-mobile-app/ [Accessed: 2015-04-20]

[33] P. F. Johnson, *Xamarin Mobile Application Development for iOS.* Olton, Birmingham, GBR: Packt Publishing, 2013.

[34] C. Petzold, *Creating mobile apps with Xamarin.Forms.* Redmond, Washington, USA: Microsoft Press, 2014.

[35] F. W. Zammetti, *Learn Corona SDK Game Development.* Berkeley, CA, USA: Apress, 2013.

[36] M. M. Fernandez, *Corona SDK Mobile Game Development : Beginner's Guide.* Olton, Birmingham, GBR: Packt Publishing, 2012.

[37] E. Ramsden, *Hall-Effect Sensors.* Burlington, MA, USA: Newnes, 2006.

[38] M. Tatipamula, E. Oki, and R. Rojas-Cessa, *Advanced Internet Protocols, Services, and Applications.* Hoboken, NJ, USA: John Wiley & Sons, 2012.

[39] S. Gundavaram, *CGI Programming on the World Wide Web.* USA: O'Reilly Media, mar 1996.

[40] R. Stewart and C. Metz, "SCTP: new transport protocol for TCP/IP," *Internet Computing, IEEE*, vol. 5, no. 6, pp. 64–69, nov 2001.

[41] G. Bora, S. Bora, S. Singh, and S. M. Arsalan, *OSI reference model: An overview.* Elsevier Inc, 2014.

[42] R. Mittal, A. Kansal, and R. Chandra, "Empowering Developers to Estimate App Energy Consumption," in *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, 2012, pp. 317–328.

[43] X. Li, X. Zhang, K. Chen, and S. Feng, "Measurement and analysis of energy consumption on Android smartphones," in *IEEE International Conference on Information Science and Technology (ICIST)*, apr 2014, pp. 242–245.

[44] A. Singhai, J. Bose, and N. Yendeti, "Reducing Power Consumption in Android Applications," in *IEEE International Advance Computing Conference (IACC)*, feb 2014, pp. 668–673.

A

Related projects

A Pushing Approach for Data Synchronization in Cloud to Reduce Energy Consumption in Mobile Devices	
Authors	S. A. L. Carvalho, R.N. Lima and A.G. Silva-Filho [4]
Platform	Android
Logging equipment	Adafruit INA219
Application type	Networking application

An empirical study of the energy consumption of android applications	
Authors	D. Li, S. Hao, J. Gui and W.G.J. Halfond [11]
Platform	Android
Logging equipment	Monsoon power meter
Application type	Networking application

Calculating source line level energy information for android applications	
Authors	D. Li, S. Hao, W.G.J. Halfond and R. Govindan [3]
Platform	Android
Logging equipment	LEAP
Application type	Networking application

Empowering Developers to Estimate App Energy Consumption	
Authors	R. Mittal, A. Kansal and R. Chandra [42]
Platform	Android
Logging equipment	Monsoon power meter
Application type	Networking application

Estimating Mobile Application Energy Consumption using Program Analysis	
Authors	S. Hao, D. Li, W.G.J. Halfond and R. Govindan [16]
Platform	Android
Logging equipment	Atom LEAP
Application type	Various mobile applications

Measurement and Analysis of Energy Consumption on Android Smartphones	
Authors	X. Li, X. Zhang, K. Chen and S. Feng [43]
Platform	Android
Logging equipment	Standard power meter
Application type	Various Android components

Measuring mobile phone energy consumption for 802.11 wireless networking	
Authors	A. Rice and S. Hay [2]
Platform	Android
Logging equipment	National Instruments PCI-MIO-16E-4
Application type	Networking application

Optimizing Background Email Sync on Smartphones	
Authors	F. Xu, Y. Liu, T. Moscibroda, R. Chandra, L. Jin, Y. Zhang and Q. Li [8]
Platform	Android
Logging equipment	Monsoon power meter
Application type	Networking application

Reducing Power Consumption in Android Applications	
Authors	A. Singhai, J. Bose and N. Yendeti [44]
Platform	Android
Logging equipment	Monsoon power meter
Application type	Graphics application

Switching Push and Pull: An Energy Efficient Notification Approach	
Authors	D. Burgstahler, N. Richerzhagen, F. Englert, R. Hans and R. Steinmetz [13]
Platform	Android
Logging equipment	Hitex Power Scale with ACM Probe
Application type	Networking application

B

Requirements

B.1 Functional requirements

System

- The system shall be able to connect to a module through WiFi
- The system shall reconnect to a module if connection is lost
- The system shall close a connection when done
- The system shall be able to receive data
- The system shall be able to display received data in a graphical user interface
- The system shall be able to display a set of filtered data
- The system shall be able to save a configuration
- The system shall be able to load a configuration on start

User

- The user shall be able to filter data based on conditions and values
- The user shall be able to switch between different graphical user interfaces

Graphical user interface

- It shall be possible to change screen orientation

- The GUI shall display data given by the system
- It shall be possible to start and stop a connection using screen interaction

B.2 Nonfunctional requirements

System

- The system shall respond to user interaction within 1 second

User

- A user shall be able to use the application without getting problems with the phone, such as screen freeze

Reliability

- The system shall not crash due to lost network, bad data or similar exceptions

Usability

- It shall be possible to internationalize the application to at least two different languages (e.g. Swedish and English)
- It shall be possible to keep the screen alive