Title: Understanding Data Analysis in an End-to-End IoT System

Student: Sindre Schei

Problem description:

Internet of Things (IoT) is known as the concept of connecting everyday physical devices to the Internet. It is natural to assume that the popularity and development within this field will increase in the following years. This means that more and more things will be able to communicate over the Internet. In the process of developing IoT, an important part is to build reliable and scalable networks, and understanding where data should be processed concerning power consumptions and costs of transferring data in different parts of the network.

The task of the thesis will be to access data in a complete prototype of an IoT network, and both collect and analyze the data. The goal is to study different alternatives for a typical IoT system, and provide an overview of current state-of-the-art technologies, products and standards that can be used in such a setting. Data can be generated by using and comparing different sensors connected to end nodes in the network.

To achieve these goals, a central part will be to understand the benefits of processing data in the end nodes, concerning power, costs and time. This means much less data needs to be sent through the network. If the calculations needed are too complex, the measured data needs to be transferred to a central node with higher processing power and easier access of energy. Another part is testing devices and sensors needed, and write programming code associated with these.

Responsible professor: Frank Alexander Kraemer, ITEM

Supervisor: David Palma, ITEM

Abstract

The Internet of Things (IoT) is known as the concept of connecting everyday physical devices to the Internet. It is natural to assume that the popularity and development within this field will increase in the following years. This means that more and more things will be able to communicate over the Internet. In the process of developing IoT, an important part is to build reliable and scalable networks, and understanding where data should be processed concerning power consumption and costs of transferring data in different parts of the network.

The task of the thesis will be to access data in a complete prototype of an IoT network, and both collect and analyse the data. The goal is to study different alternatives for a typical IoT system, and provide an overview of current state-of-the-art technologies, products and standards that can be used in such a setting. Data can be generated by using and comparing different sensors connected to end nodes in the network. A complete network of both microcontrollers and single-board computers will be built and explained in this thesis. The network will from now on be referred to as *testbed*.

Microcontrollers as end nodes in an IoT network will be the central element tested in this thesis. The main focus is to establish a connection between two devices, A and B, and form a network between these that can transport data efficiently. A central point of discussion will be to find transfer protocols and technologies that are in such a network. It will be discussed the advantage and disadvantage of sending data rather than doing computation in end nodes, with a main focus on optimal throughput in the network. To do this, a deep understanding of the benefits of processing data in the end nodes, concerning power, costs and time is needed.

Results from this work include graphs and discussions explaining in which case the different transport protocols suggested are preferred, from tests done in the testbed. These show that different protocols are suited for different usage, and that one of the tested possibilities more stable than the other in the tests presented. Both registered their highest measured goodput at approximately 600 bytes/second. Being a quite slow transfer rate, this opened up for another discussion about the possible use cases for future Bluetooth Low Energy (BLE)-based IoT applications.

Keywords: Optimizing payload sizes, fragmentation, maximizing throughput, power usage.

Sammendrag

Tingenes Internet, mer kjent under det engelske navnet Internet of Things (Iot), er konseptet der hverdagslige fysiske gjenstander kobles til Internet. Det er naturlig å anta at populariteten og utviklingen rundt dette vil være økende de kommende årene. Dette betyr at flere og flere ting vil kunne kommunisere over Internet. I prosessen der Tingenes Internet utvikles, er en viktig del å bygge pålitelige og skalerbare nettverk, samt å forstå hvor i nettverket data bør prosesseres med tanke på energibruk og kostnader ved å overføre data mellom deler av nettverket.

Hovedoppgaven presentet i denne avhandlingen vil være å jobbe med data i en komplett prototype av et Tingenes Internet-nettverk, og både samle og analysere dataene. Målet er å studere de forskjellige alternativene for et typisk slik nettverk, og lage en oversikt over teknologiene og produktene som kan bli brukt i denne sammenhengen. Data som trengs til dette kan bli hentet ved å sammenligne forskjellige sensorer koblet til løvnodene i nettverket.

Med dette som utgangspunkt vil oppgaven bli kul.

Preface

This thesis was issued by the Department of Telematics (ITEM) at the Norwegian University of Science and Technology (NTNU) the spring of 2016 as a Master Thesis in Telematics. The responsible professor has been Frank Alexander Kraemer, ITEM, who has given helpful advices on how to build up and write such a large project, as well as answering questions and giving support the whole period. David Palma has been the supervisor, giving impressively close monitoring of the project to fill in ideas, thoughts and good advices to help me finish the thesis. I would like to thank both these ITEM representatives for the work they have put down to make this project as good as possible.

Secondly I would like to thank fellow students for the many discussions, good advices and other more social activities the last five years. I would like to point out the guys at A-179 for their support, Jon Anders for helping me with code specific problems during the programming period, and Anders for the many hours we spent together setting up central parts of the network described in this thesis. Special thanks to you!

Last, but not least, I would like to thank my family for their support both during the period of this thesis, but also during my entire period as a student. Special thanks to my father, Svenn Arne, for helping me review the thesis and to get a discussion point with someone without a technical background.

Contents

List of Figures

List of Tables

List of Acronyms

6LoWPAN IPv6 over Low Power Wireless Personal Area Networks.

ACK Acknowledgement.

ACL Asynchronous Connection-Less.

AWS Amazon Web Services.

BLE Bluetooth Low Energy.

CoAP Constrained Application Protocol.

CON Confirmable CoAP message.

CPU Central Processing Unit.

DNS Domain Name System.

GUI Graphical User Interface.

HCI Host Controller Interface.

HTTP Hyper Text Transport Protocol.

I2C Inter-Integrated Circuit.

ICMP Internet Control Message Protocol.

ICMPv6 Internet Control Message Protocol version 6.

IDE Integrated Development Environment.

IoT Internet of Things.

IPv4 Internet Protocol version 4.

IPv6 Internet Protocol version 6.

ISM Industrial Scientific Medical.

ITEM The Department of Telematics.

L2CAP Logical-link Control and Adaption Protocol.

LAN Local Area Network.

M2M Machine-to-Machine.

MQTT Message Queueing Telemetry Transport.

MTU Maximum Transmission Unit.

NDP Neighbor Discovery Protocol.

NON Non-Confirmable CoAP message.

NTNU Norwegian University of Science and Technology.

OS Operating System.

PAN Personal Area Network.

QoS Quality of Service.

radvd Router Advertisement Daemon.

RAM Random Access Memory.

RTT Round Trip Time.

SoC System on chip.

SPI Serial Peripheral Interface.

TCP Transmission Control Protocol.

TDMA Time Division Multiple Access.

TFTP Trivial File Transfer Protocol.

UDP User Datagram Protocol.

UI User Interface.

USB Universal Serial Bus.

Glossary

ADXL345 accelerometer	Small accelerometer mounted on a chip delivered by Adafruit. Connected to the nRF52 in the system presented.
byte	Used as a synonym for *octet*, meaning 8 bits put together as one unit.
goodput	The number of bytes in a packet that contains the intended message, sent one way through the network per time unit.
message code	Is a code for a message in CoAP, for instance PUT, SET or GET.
message type	Is a type of message in CoAP, for instance CON, NON or ACK.
microcontroller	Small computer that contains processor, memory and programmable parts in one integrated circuit.
nRF52	Nordic Semiconductor nRF52 DK , Development Kit for the nRF52 microcontroller used as end nodes in this system. From now on noted as "nRF52".
packet	A network packet is a chunk of data transported through the network as one piece. The packet size may wary from protocol to protocol.
payload	The part of the transmitted data that is the intended message.

Raspberry Pi	Mini computer in the size of a credit card. Runs on electric power from a cord in this system. From now on noted as "Raspberry Pi" or shortened to "Pi".
single-board computer	Small computer that contains processor, memory and programmable parts and I/O features like ports and antennas on a single circuit board.
throughput	The total amount of data sent one way through the network per time unit. The sum of payload and additional header files needed to transport the packet.

Chapter 1 Introduction

1.1 Motivation

Internet of Things (IoT) is a general term describing a network of small devices connected to the Internet with either a direct connection, or using a forwarding device as a central point of connection. The term includes all sort of devices, from small sensors and microcontrollers to everyday smart objects, from phones and glasses to cars and buildings. A common factor for all of these is Machine-to-Machine (M2M) communication, where machines can communicate with each other without Human-computer interaction. The term IoT was first used by Kevin Ashton in 1999 [10], describing a global network of objects. He later explained how he predicted that most of the data contained on the Internet today will be overtaken by the amount of sensor collected data with M2M communication in the future. Both with this as an argument, and the high interest for smart devices and sensors in the general population, it may be said with a high certainty that this will be a central part of the coming years of the Internet.

A develop from a network mostly based on human made material to a network based more on data from sensors using M2M communication, require that several factors needs to be considered. It is natural to believe that most end nodes must be battery powered for practical reasons. If a complete system contains hundreds of sensors and microcontrollers it would be impractical to set up a power cable to all of these. From a users perspective, it would also be very annoying to have to change these batteries very often. Because of this, the available computational power in the end nodes is very limited, and should be limited further as much as possible to increase the battery life.

A very central point of discussion in any IoT system will be how to transport data as efficient as possible. Raw data from sensors are seldom useful to an end user. Therefore, the data needs to be analysed and often represented in another form before it can be useful to the user. A device in the network needs to analyse the data, find out what is important or unimportant, search for patterns, draw graphs or figures and forward the results to a monitor, a web page, or to be stored on a server to be used later. Arguments will be presented to discuss if the process of analysing data should take place in the end nodes, or if it is more preferable to forward raw data to a central component of the network.

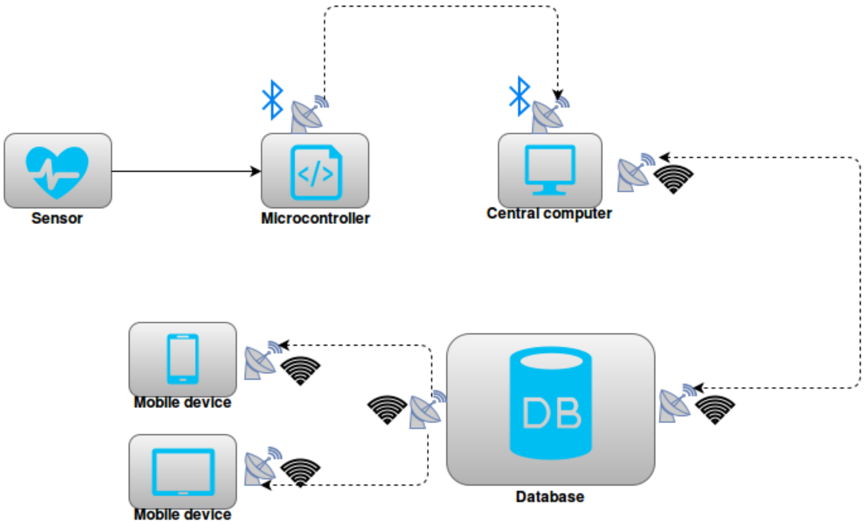


Figure 1.1: Example of IoT architecture

Figure 1.1 shows an example architecture of IoT. Here a sensor, (or in real life scenarios most likely several sensors), is connected to a microcontroller by a standard interface using cables.

A central company in developing technologies to be used is Nordic Semiconductor. Having one of the world leading companies developing microcontroller based in Trondheim, is also a motivation. To use locally developed devices seemed like a perfect opportunity for a project on Norwegian University of Science and Technology (NTNU).

1.2 Scope and objectives

1.2.1 Scope

This thesis will mainly focus on the best way of optimizing transportation and analysing data in an IoT network. The goal is to find the optimal solution on how to treat data. Central points of discussion will be:

- How to gather data from sensors efficiently, both concerning time and power consumption
- How to transport data efficiently, considering power consumption and optimal throughput, both concerning time spent, and amount of useful data that gets through
- To find where in the network it is preferable to analyse the raw data, concerning energy consumption and time spent in total

To achieve these central points, some explanation of background protocols, used devices and network topology will be addressed as well, in addition to low-level details needed to set up the system architecture, in order to maintain a stable and reliable network.

1.2.2 Objectives

O.1: Build a star network of microcontrollers

This is the most elementary objective, to build a network that can be tested. All the other objectives are dependent on this.

O.2: Connect sensors to the end-nodes to collect data

Objective two involves gathering real data. To do this, some kind of sensor is needed, and needs to be correctly configured for the end node to be sure that the read data can be trusted and reliable. Objective three and four can still be successful without this, since simulated data can be a replacement.

O.3: Gather information of the data sent through the network

Objective three is to find tools or write programming code to gather and analyse the data sent through the network, and present these in a way that makes it easy to spot the advantages or disadvantages of the different protocols and technologies.

O.4: Analyse and discuss the gathered information

Objective four involves discussing the given results, and use these to discuss and draw conclusions on how to optimize the network and propose solutions, improvements or further work.

1.2.3 Research Questions

R.1: Which transport protocols are suitable in such a network?

To answer this question, the network must be built and tested, to see if there are any noticeable differences in the tested protocols.

R.2: What are the main limitations concerning transporting data?

This question must be answered by measuring time spent in the different parts of the network during routing of packets, to determine the bottleneck of the network or system.

R.3: Are the microcontrollers powerful enough to gather data this frequently?

This is not specified in the documentation of the microcontrollers, since this depends on the network, the type of sensor and the type of data. To answer this question, the sensors must therefore gather data at an even higher rate to see if it is possible to reach an acceptable rate of sampling.

R.4: Could data analysis be done in the end nodes in this network?

This is dependent on the result from R.3. This might be possible if the results reveal that the microcontrollers can easily handle the gathering of data and still have power to do calculations. The alternative is to forward raw data to a central node.

1.3 Structure

Chapter 2 describes the technical background of technologies, protocols and devices needed to understand the rest of this thesis, and explains why some solutions were chosen over others in this particular network. This chapter answers objective O.1 in detail, and discusses research questions R.1 and R.2.

Chapter 3 describes in detail how the different components of the network are connected and set up to communicate with each other. This chapter answers objective O.2, and discusses research question R.2 further.

Chapter 4 describes, explains and discusses the performed network measurements using tables and graphs of gathered data as a central point of discussion. This chapter answers objective O.3 and discusses the research questions R.3 and R.4. The chapter concludes that both Constrained Application Protocol (CoAP) Confirmable CoAP message (CON) and Non-Confirmable CoAP message (NON), have their advantages in different scenarios, which is summarized in chapter 4.6. CON has still been without doubt the most reliable when tested in this network.

Chapter 5 discusses the results found in chapter 4 further, by going through the central points of the objectives. It discusses what was most successful, what could have been better and what should be considered for future works. At the end the chapter contains an overall evaluation of the used devices and technologies, and how the experience gained in this project can be used in the future.

Chapter 6 summarizes the entire work conducted in this project and presents the final conclusion. In the end, possible future works are discussed.

Chapter 2
Background

This thesis describes the setup and usage of an end-to-end IoT system. In order for the testbed to be set up and reproduced by others later on, a detailed description of components, sensors and protocols used, is provided. This chapter will undergo the background information of the devices, technologies and protocols used, and why these where chosen over other alternatives.

2.1 Hardware

The hardware section will undergo the physical devices used to build the IoT network, which is central to solve objective O.1.

2.1.1 Raspberry Pi

Developed by Newark Element 14, the Raspberry Pi has become a central tool for many people wanting to get started using small computers [5]. The device has been known as a single-board computer specially designed for small network projects. It can be used as an educational tool used all the way from elementary schools to higher-education research environments, such as here at NTNU. This was therefore a natural device to use as a starting point in the testbed.

The Raspberry Pi is the size of a credit card. Model 3 of this was released in February 2016, just in time to become a part of the system set up in this project. This includes a Central Processing Unit (CPU) speed of 1,2 GHz and 1 GB of Random Access Memory (RAM). This makes it approximately 12 times faster than the first Raspberry Pi. Both Bluetooth and WiFi are included, and it was quite easy to set up, given that the right Linux kernel has been used in the Operating System (OS) of the Pi. Along with the Raspberry Pi, a good and stable operating system with a kernel that supported the IPv6 over Low Power Wireless Personal Area Networks (6LoWPAN) architecture were needed. For this, Ubuntu Mate version 15.10 with kernel version

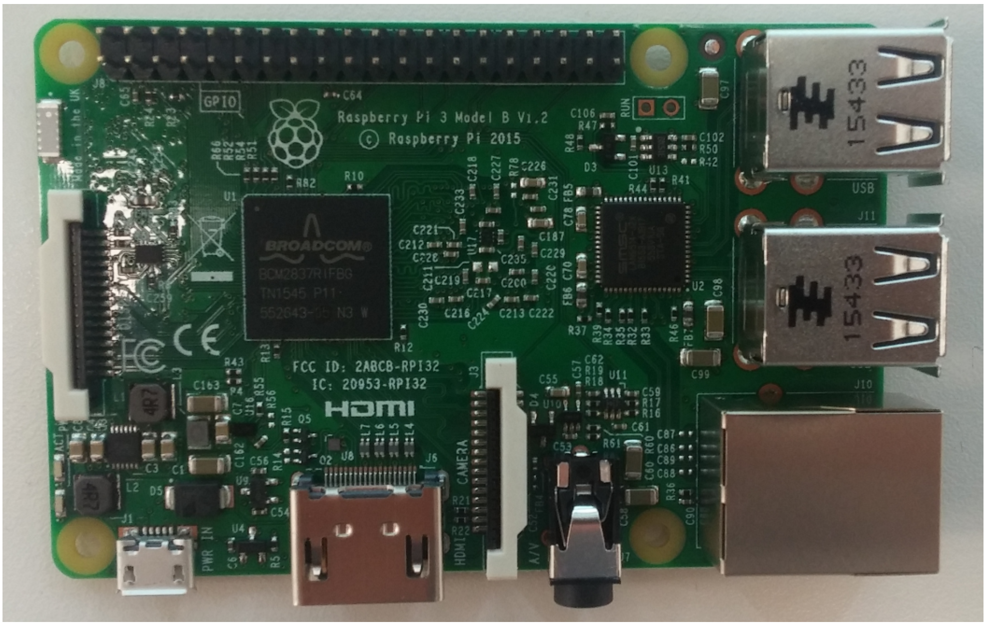


Figure 2.1: Raspberry Pi 3

4.15 was chosen, and used on the Raspberry Pi. As other versions of Ubuntu, this is Linux based, and has a complete Graphical User Interface (GUI) of a full OS.

2.1.2 nRF52

The most central device of this network is the microcontroller used as end-node, the nRF52 developed by Nordic Semiconductor with the IoT development kit. It is presented as a family of highly flexible, multi-protocol system-on-chip devices.

This device has been advertised as a powerful multiprotocol single chip solution, with both a 32-bit ARM Cortex processor, a 512kB flash, and 64kB of flash memory. The key features mentioned by Nordic Semiconductor [7] that will be relevant in this network are:

- Multi-protocol 2.4GHz radio
- Application development independent from protocol stack
- Full set of digital interfaces including Serial Peripheral Interface (SPI) and Inter-Integrated Circuit (I2C)

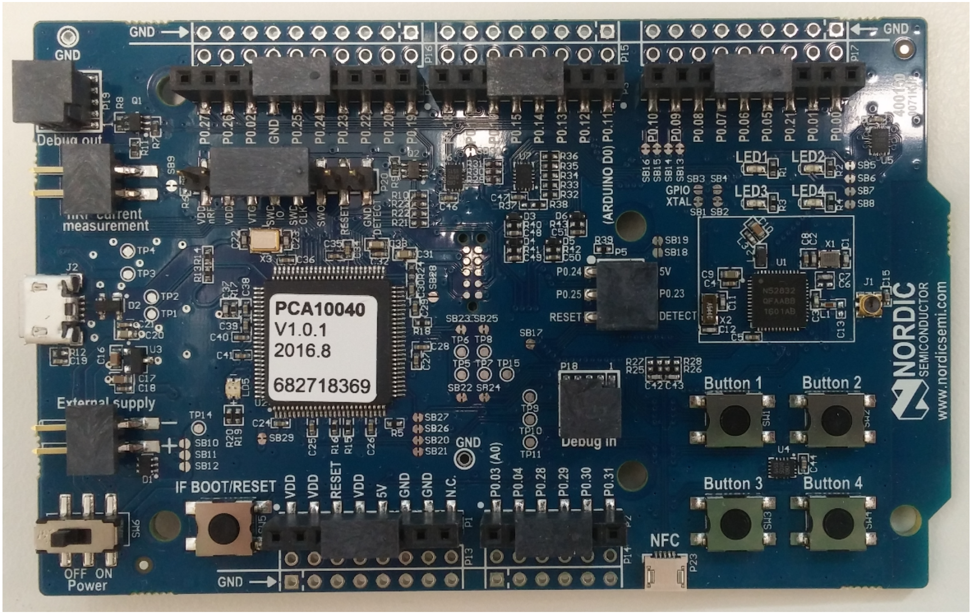


Figure 2.2: Nordic Semiconductor nRF52

– Low cost external crystal 32MHz ± 40ppm for Bluetooth, ± 50ppm for ANT

– Wide supply voltage range (1.7 V to 3.6 V)

The most interesting points here are the processing power, the flash storage and RAM, the I2C and SPI buses, and the bluetooth antenna. In this project, three different versions of the nRF52, named (from oldest to newest) *PCA10036 V1.0.0, PCA10040 V0.9.0* and *PCA10040 V1.0.1* were used. All three shows similar results when tested in this system, and Nordic Semiconductor reported that the only significant change is that newer versions should be more stable. Almost all tests in this thesis have been done by using PCA10040 V0.9.0.

A SoftDevice is a precompiled binary software that implements BLE on the nRF52. This means that the user can start to work directly in a standard C language interface, which is independent from the Soft Device implementation [8]. This makes it possible for users to write standard programming code instead of requiring a deep knowledge of device specific configurations. There are several versions of SoftDevices to the nRF52 that can be downloaded from Nordic Semiconductors website[1].

[1] http://www.nordicsemi.com

2.1.3 Adafruit ADXL345 Accelerometer

As seen in the previous section, the nRF52 have several possibilities when it comes to radio communication. In addition to this, an external sensor was needed to collect data. Supporting both the I2C and SPI, the nRF52 has got most of the standard interfaces needed. Objectives presented in the introduction to this thesis says that it would be preferable both to collect, transport and analyse data in this network. The sensor chosen to do this was the ADXL345 accelerometer from Adafriut. This was selected for the following main reasons:

- It can measure acceleration in all three axes, X, Y and Z.
- It sends digital data immediately, therefore it is no need to use computational power to calculate digital values as needed if the data was captured by an analog accelerometer.
- It supports both I2C and SPI, which makes it possible to connect to the nRF52.
- It supports voltage of 3.3V-5.0V, which fits within the range of output from the nRF52.

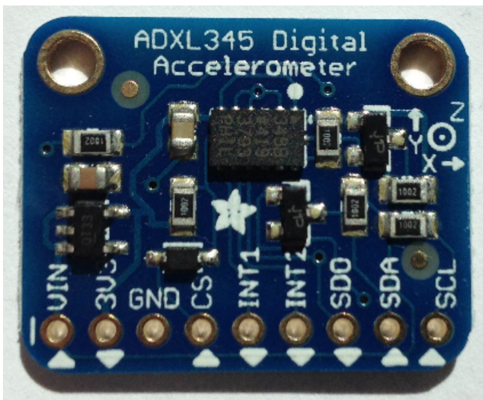


Figure 2.3: ADXL345 Accelerometer

When connecting to the nRF52, using the I2C interface was chosen because it is simple with few cables, supports an acceptable bit rate, and several sensors in the same link.

2.1.4 Additional computational power

The devices presented so far are small network devices, reaching from limited computational power to a more powerful central device. These devices can be used as end nodes or more central nodes in an IoT network. The Raspberry Pi has already network connectivity, and can therefore be used as the final node before the results are presented on a screen, a web page, or to be stored on a server. In many cases however, it will be an advantage to include another node with considerably more computational power before the results are being published. This both limits the computations needed to be done at the Raspberry Pi, and means that the systems are able to do more deep analyses of the gathered data, without the fear of a system overload. A central stationary computer, a supercomputer or computational power from a web service like Amazon Web Services (AWS) are possible solutions. In this network a standard stationary computer running a Linux Ubuntu based system was used as this node. Because of the limited time provided for this thesis, the scope focuses mostly on data analysis and transportation between small nodes in an end-to-end IoT system. The results were mostly obtained and calculated on the Raspberry Pi, meaning this last central node was not extensively used in this solution, but could be a central topic for future projects aiming for a more complex data analysis.

2.1.5 Alternative devices

An alternative for the Raspberry Pi was never considered, since this is a well known device with a good reputation. This should be easy to use, easy to find advice and help when needed, and easy get hold on devices when needed. There are of course other alternatives available[2] that could have been considered if there where any problems with the Raspberry Pi. The main contestant to replace the nRF52 was the Zolertia Z1[3] microcontroller. This was a good alternative to use since it already has got an accelerometer fitted on the board, but does not have the same computational power as the nRF52.

2.2 Communication technologies

After the devices to use had been selected, the next step was to find relevant communication technologies that could be used to establish a reliable, fast and low power connection between the nRF52 and the Raspberry Pi.

2.2.1 Bluetooth Low Energy

BLE, also known as *Bluetooth Smart*, is a wireless technology for short range communication developed by the Bluetooth Special Interest Group. The idea was to create a low energy single-hop network solution for Personal Area Networks (PANs). A major advantage of this solution is that Bluetooth 4.0 is already a well established

[2] For instance the Ardurino Uno, Banana Pi or the BeagleBone Black
[3] http://zolertia.io/product/hardware/z1-platform

technology in cell phones, laptops and several other devices. This means that few changes needs to be made to these devices, in order to be able to work with bluetooth smart. However, to this date, a device that only implements BLE is not able to communicate with a device that only implements classic Bluetooth [13]. The 6LoWPAN Working Group has recognized the importance of BLE in IoT [14], as one of the most central technologies in the further development.

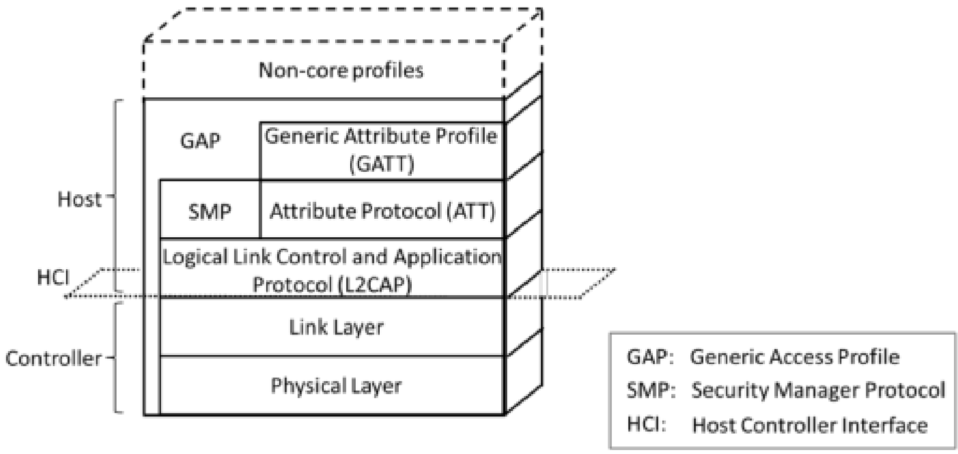


Figure 2.4: BLE protocol stack [13]

The protocol stack of BLE has two main parts, the controller and the host, as shown in 2.4 [13]. In the testbit, the Raspberry Pi represents the controller (master), nRF52 the host (slave). The communication between these components are done through the standard Host Controller Interface (HCI), a bluetooth protocol. All slaves are in sleep mode by default, and are woken up by the master when these components are needed. Links are being identified by a randomly generated 32-bit code, and the Industrial Scientific Medical (ISM) band used is 2,4 GHz [13]. Other protocols include Logical-link Control and Adaption Protocol (L2CAP) used to multiplex data between higher protocol layers, and the segmentation and reassembly of packets. From here packets are being passed to the HCI, which is the interface used to communicate between the two BLE devices. This is being used in conjunction with Asynchronous Connection-Less (ACL), which is used to create the Time Division Multiple Access (TDMA) scheme used to transfer packets over the network link, as well as controlling uptime of the end nodes, as this link is set to disconnect automatically after a given time period if there is no activity on the link. Concrete examples of these protocols will be shown later in the thesis.

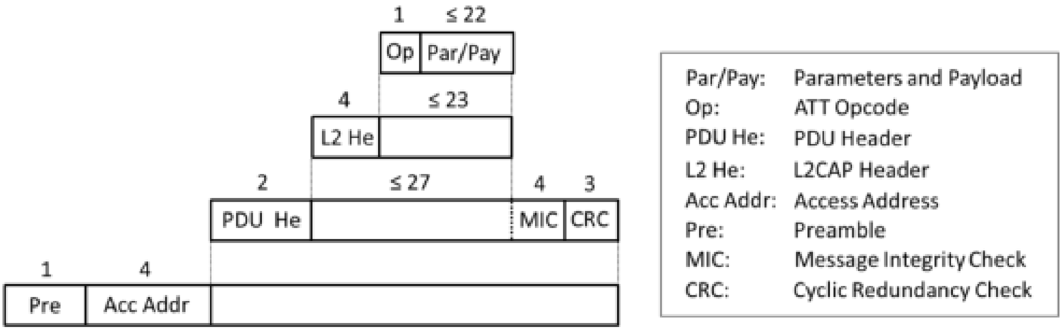


Figure 2.5: BLE Data Unit Structure [13]

Figure 2.5 shows the data unit structure in BLE, meaning the different fields that can be used in a packet [13]. The header fields of 4 byte of access addresses and L2CAP will be central topics of discussion later in this thesis. In the case of the network presented here, when the BLE slave has been connected to a master, it stops searching for other connectable points. This means that it is not possible to connect to several masters, and it will only be possible to create a *star network*, not a *mesh network*. A mesh network would in many cases be preferable, since BLE is considered a PAN with a very limited range. In a mesh network end nodes can communicate with each other, meaning they can span a larger area without the need of a central and common point of connection. Otherwise, BLE seems like a very good alternative in this project.

2.2.2 6LoWPAN

6LoWPAN is a defined protocol for using Internet Protocol version 6 (IPv6) in low energy networks, to identify sensors and devices over IEEE 802.15.4, as defined in RFC 4944 [19]. To use The Internet Protocol in low energy networks in addition to standard networks was proposed by Geoff Mulligan and the 6LoWPAN Working Group[20]. It was chosen because it seemed like a simple and smart protocol definition. Since packets in this network can end up being forwarded all the way from a microcontroller to a central computer through several nodes without being changed, it makes sense to use the same base protocol for all links. In [20], the advantage of 6LoWPAN is explained as not too big to be used in small networks with a small header field, and more flexible to network sized compared to *Zigbee*[4] and *Zensys*[5].

[4] http://www.zigbee.org/
[5] http://www.zensys.com/main.html

Utilizing IP in these networks and pushing it to the very edge of the network devices flattens the naming and addressing hierarchy and thereby simplifies the connectivity model. This obviates the need for complex gateways that, in the past, were necessary to translate between proprietary protocols and standard Internet Protocols and instead can be replaced with much simpler bridges and routers, both of which are well understood, well developed and widely available technologies [20].

6LoWPAN was developed to be used in small sensor networks, and implementations can fit into 32Kb flash memory parts. The Maximum Transmission Unit (MTU) is given to be 1280 byte, and it also uses a complex header comparison mechanism that allows the transmission of IPv6 packets in 4 bytes, much less than the standard IPv6 40 bytes. This is done by using stacked headers, same as in the IPv6 model, rather than defining a specific header as for Internet Protocol version 4 (IPv4). The device can send only the required part of the stack header, and does not need to include header fields for networking and fragmentation [14]. The maximum packet size of the physical layer is set to be 127 bytes, far below the limit of 1280 byte [17]. It is expected that other layers will produce packets of the desired size to fit the system. In the example code on the nRF52 in the testbit this is set to 270 bytes for every packet This will be shown in practical examples and tests later in the thesis.

2.2.3 Other alternatives

ANT was the other main alternative to BLE when network protocols where chosen. It also uses the 2,4 GHz ISM band, and is made to be used in sensor based networks [2]. It is supported by the nRF52, and could be used with a Raspberry Pi if an ANT Universal Serial Bus (USB) dongle is fitted. This would have solved the BLE problem not being able to connect several devices together in a mesh network, since ANT supports this. Other than this the difference is small. BLE is, on the other hand, supported by other mobile devices, meaning it is possible to use a mobile application developed by Nordic Semiconductor to test the connection. One of the main intentions from The Department of Telematics (ITEM) when the problem description was written was to test BLE in such a setting, therefore this was chosen.

Two other contestants, other than 6LoWPAN, were *Zigbee* and *Zensys*, being compared directly in [20]. The major factors here are that 6LoWPAN has a network size bigger than the others. It supports Internet connectivity using routers, the use of User Datagram Protocol (UDP) and Transmission Control Protocol (TCP), low amounts of RAM and small headers [20].

2.3 Transport protocols

In order to transfer data from the end nodes to the central points of the network, either for analysing or already analysed data, a fast, efficient and stable transport protocol has to be used. This is a central aspect in this network, because the limitations of the sending rate is thought to be one of the main constraints for network throughput, either in the form of limits of data at once or number of transmissions per second. The protocol needs to be stable and energy efficient and work with both BLE and 6LoWPAN. Nordic Semiconductor provides example code and examples on how to get started with this, having been used as the basis for this work.

2.3.1 CoAP

CoAP is a transport protocol designed to be used in constrained networks for M2M communication. It is UDP based, and works well in low-power and lossy networks. It can be used with microcontrollers, and with IPv6 and 6LoWPAN. Both GET and PUSH functionalities can be used, as well as *observable* GET. Other commands used in CoAP are GET, PUT, POST and DELETE, to get or change data. This means that a server can "subscribe" to end nodes in the network, and get updates either after a given time span or when changes have been made. Therefore, this seemed like a promising protocol to use, and was chosen as the main transport protocol in the network test[22]. The main technical features described in CoAP include fulfilling M2M requirements, support of asynchronous messages, UDP based communication and stateless mapping to Hyper Text Transport Protocol (HTTP).

CoAP has several similarities with HTTP, using the same client and server roles. The client sends a request, and the server sends a response back. Many of the response codes are also very similar, with *404: Not found* as the best known. When M2M communication is used, both participants sometimes need to be both client and server, and the CoAP protocol handles this with a two-layer approach. There are four different main messages defined in CoAP [22].

- A Confirmable message, CON, requires one Acknowledgement (ACK) for every packet received. If a message is not received correctly, the receiver will ask for exactly return message of the type Acknowledgement.

- A Non-confirmable message, NON, does not require an ACK. This may result in a higher possibility of a packet getting lost, especially in lossy networks, but it requires less capacity from the network and should in general be faster.

- An Acknowledgement acknowledges that a specific CON packet has reached its destination.

– A Reset message tells the sender that a specific message was received, but some content is missing in order to to be able to fully understand it. For instance if the receiver has had a reboot during the transmission. An empty Reset message represents a ping test of Round Trip Time (RTT), which will be used when testing a connection.

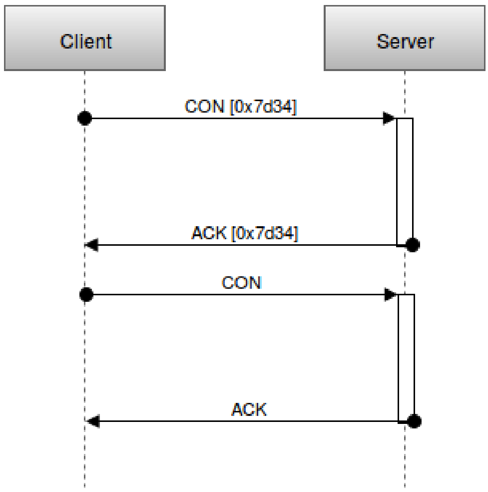


Figure 2.6: CON CoAP set up sequence diagram [22]

Figure 2.7 shows the basic message sequence between the client and the server in a CoAP CON network. Every request CON message needs to be given an ACK back.

Figure 2.9 shows the same for NON, where no ACKs are needed. The same initial set up with CON and ACK messages are still needed, to establish a connection between the client and server before a stream of NON messages can be sent. This means a much more unreliable connection than using CON, since messages can be dropped without either the client or the server gets notified. Systems where a few packets can be lost without difficulties, for instance in a sensor based network like the network presented in this thesis, can use this as an advantage. A message ID is still provided to every message to remove duplicated messages, but dropped messages are lost data. If the sent packages contained data that could not be dropped, for instance containing crucial patient information from sensors on a patients body, this is not a good solution. The more reliable solution CON is a much better alternative in this case. These differences are the same as being experienced in the IoT system described in this thesis, which can be seen in figure 2.8. A CON message is sent several times, to set up the initial connection. When an ACK is received as a response, the continuous transportation of NON packets without ACKs can begin.

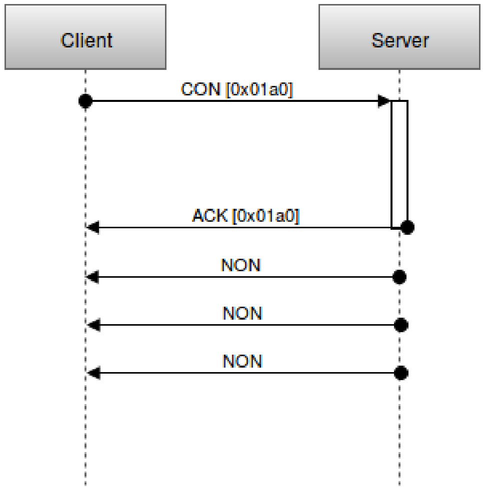
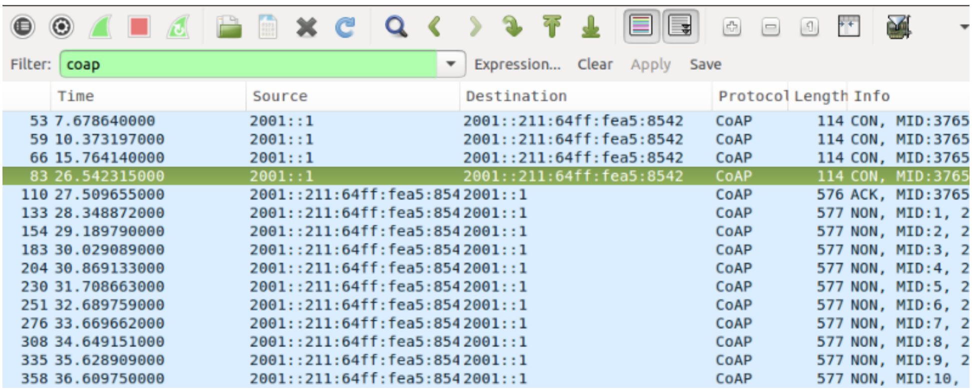


Figure 2.7: NON CoAP set up sequence diagram [22]



	Time	Source	Destination	Protocol	Length	Info
53	7.678640000	2001::1	2001::211:64ff:fea5:8542	CoAP	114	CON, MID:3765
59	10.373197000	2001::1	2001::211:64ff:fea5:8542	CoAP	114	CON, MID:3765
66	15.764140000	2001::1	2001::211:64ff:fea5:8542	CoAP	114	CON, MID:3765
83	26.542315000	2001::1	2001::211:64ff:fea5:8542	CoAP	114	CON, MID:3765
110	27.509655000	2001::211:64ff:fea5:854	2001::1	CoAP	576	ACK, MID:3765
133	28.348872000	2001::211:64ff:fea5:854	2001::1	CoAP	577	NON, MID:1, 2
154	29.189790000	2001::211:64ff:fea5:854	2001::1	CoAP	577	NON, MID:2, 2
183	30.029089000	2001::211:64ff:fea5:854	2001::1	CoAP	577	NON, MID:3, 2
204	30.869133000	2001::211:64ff:fea5:854	2001::1	CoAP	577	NON, MID:4, 2
230	31.708663000	2001::211:64ff:fea5:854	2001::1	CoAP	577	NON, MID:5, 2
251	32.689759000	2001::211:64ff:fea5:854	2001::1	CoAP	577	NON, MID:6, 2
276	33.669662000	2001::211:64ff:fea5:854	2001::1	CoAP	577	NON, MID:7, 2
308	34.649151000	2001::211:64ff:fea5:854	2001::1	CoAP	577	NON, MID:8, 2
335	35.628909000	2001::211:64ff:fea5:854	2001::1	CoAP	577	NON, MID:9, 2
358	36.609750000	2001::211:64ff:fea5:854	2001::1	CoAP	577	NON, MID:10,

Figure 2.8: CoAP NON, set up sequence, Wireshark capture

The message format used in CoAP is very simple, and can be seen in figure 2.9. The 4 first byte are header files, followed by optional tokens and options. When these are not being used, like in this network, the minimal header size will be either 4 or 5 bytes. This is a huge advantage in IoT networks.

2.3.2 MQTT

An alternative transport protocol in a system such as this is Message Queueing Telemetry Transport (MQTT). This is known as a publish-subscribe messaging system based on TCP for M2M communication. A client will in this case *subscribe* to

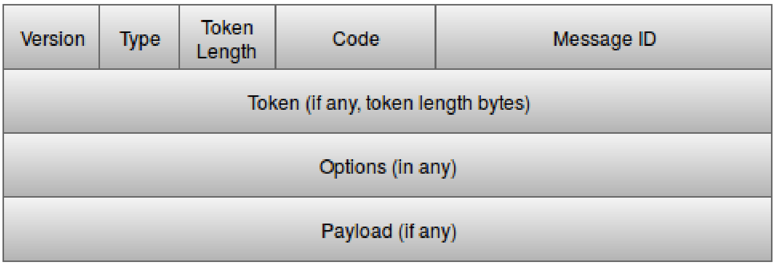


Figure 2.9: CoAP message format [22]

a *publisher* in the network [15]. When a publisher updates a field of interest for the subscriber, the subscriber will get notified. Subscriptions are being coordinated by a *broker*, as seen in figure 2.10. Messages sent in such a network are either *sub(topic)* to subscribe to a topic, or *pub(topic, data)* to publish data [4].

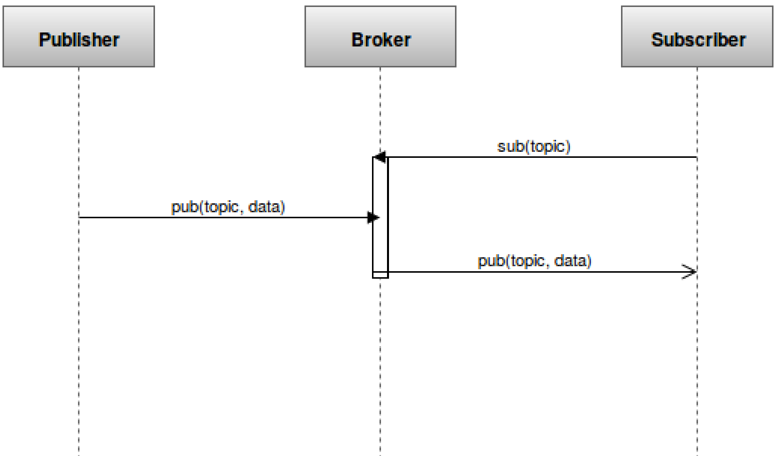


Figure 2.10: MQTT subscription sequence diagram [15]

MQTT supports end-to-end Quality of Service (QoS), and has a simple and effective message architecture. This protocol would also be possible to use in the testbit. Because of the limited time frame of this thesis however, it was decided to study CoAP in depth first, and leave the testing of MQTT to future work.

2.4 Software tools

As an Integrated Development Environment (IDE), the *KEIL Vision* was used, as recommended by Nordic Semiconductor in [1], for writing C programming. For other programming languages, (for instance Python 3.4), Sublime Text 2 for Windows and Linux was used, as well as *Pluma* for Ubuntu Mate on the Raspberry Pi.

Wireshark is a software tool used to analyse networks and capture packets sent with different technologies [18]. Later the data can be filtered and analysed, for instance by filtering out all packets except CoAP and BLE, which was used in this case. Wireshark has been one of the most important tools to be able to analyse data to such an extent as done in this thesis. An example of use is shown in figure 2.8.

Copper[6] is a generic browser which can be used in *Firefox* browser [3]. It is made to be used in IoT networks based on CoAP, just like this network. Using Copper it was easy to use GET and PUT messages, as well as observing a server by using a simple GUI. By removing the need of using terminal commands and programming scripts this makes the system easier to use with less development effort outside the scope of this thesis. An example of the GUI can be seen in figure 2.11 [16].

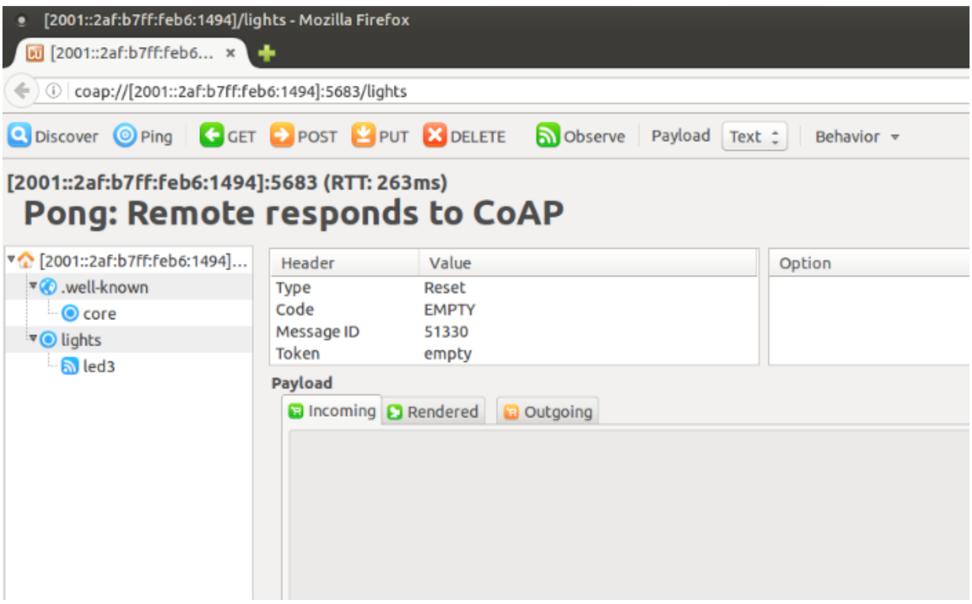


Figure 2.11: Copper example

[6] http://people.inf.ethz.ch/~mkovatsc/copper.php

Router Advertisement Daemon (radvd) is a software tool that can be used to advertise IPv6 addresses in a local network, using Neighbor Discovery Protocol (NDP) [11]. It is being used to multicast and forward packets in this network. When a packet is sent from an end node to another, the communication needs to go through the central point in the star network, the Raspberry Pi in this case. Here, radvd ensures that the packages are being routed to the right end point, or the right nRF52 in this system. To make the most basic figures in this thesis the web based tool *draw.io* was used, unless no other is specified. *polt.ly* was used to draw the graphs used. All the images of another devices in the network have been taken by the author.

Chapter 3
System Architecture

The purpose of this thesis is to build an end-to-end system, that will be able to transfer data all the way from a microcontroller to a server. This chapter will describe in detail how the different components of the testbit are connected, and how the different protocols have been configured to read, process and transfer data efficiently.

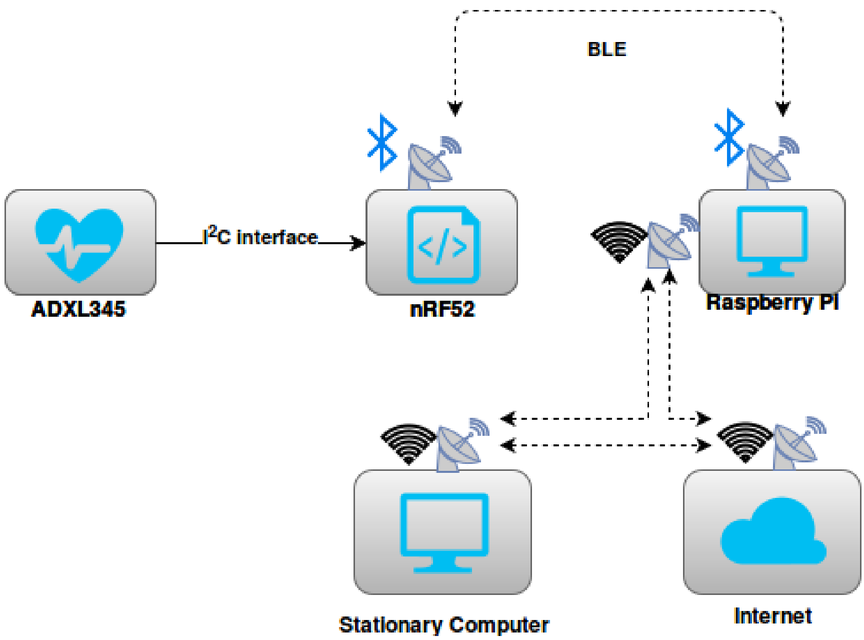


Figure 3.1: End-to-End architecture in the presented system

Figure 3.1 shows how the complete end-to-end system of this thesis is set up. In short terms, the ADXL345 accelerometer is connected to the nRF52, using the I2C interface. The nRF52 is connected to a Raspberry Pi using 6LoWPAN and BLE. The Raspberry Pi is connected to the Local Area Network (LAN), and can therefore forward data to a central stationary computer if additional computational power is needed. Otherwise the results can be sent to the web, to a screen or anywhere else. Several microcontrollers can possibly be connected to a Pi at the time, forming a star network. Up to eight connections have been tested successfully in this network.

There are in general three main limitations in a network such as this:

- Computational power in the different nodes
- Battery capacity of the end nodes
- Network limitations between the nodes

A central part of the testing in this thesis will be to test the different limitations, and to understand the advantage and disadvantages of doing computations in end nodes, compared to transferring information to a server with higher computational power. Power usage is very often closely related to computational power, and will also be a central factor. The next section will contain a walk-through of the system, and discuss these three main limitations in each node and the links between them.

3.1 Connecting Raspberry Pi and nRF52

Since its not possible to connect a screen to the nRF52, it makes sense to connect this to the Pi first, before measuring values. To set up the communication between a Raspberry Pi and the nRF52, the two code examples TWI and Observable server from Nordic Semiconductor were used as a starting point for coding on the nRF52. It was however not straight forward to connect these two together the first time. Following is a listing of Linux terminal commands that was run on the Raspberry Pi to get the system up and running [6].

Install an OS on the Raspberry Pi that has a Linux kernel version later than 3.18. On *Raspbian* version 3.18 is the only stable version, (Note: Jan. 2016), but *Ubuntu Mate* is stable in version 4.15. Ubuntu Mate was therefore chosen as the best and most stable OS, and was installed on the memory card from another computer [9]. When this is done, a resizing of the file system is needed to use all the capacity of the memory card. This is not crucial to get the OS up and running, but recommended to be able to use more than 4GB of the memory card. Recommended size of the memory card is 16GB. To resize, after the initial boot of the OS on the Raspberry Pi, run the following commands:

Remove commands to Appendix

```
sudo fdisk /dev/mmcblk0
```

Delete partition (d,2), and run the following after a reboot

```
sudo resize2fs /dev/mmcblk0p2
```

All the following commands require admin rights on the system. It is therefore easier to type in the following command to temporarily become a *super user*. Alternatively type in *sudo* before every command in the rest of the recipe.

```
sudo su
```

It should now be possible to exploit the whole memory card, and start downloading and activating services needed in the system. To use BLE, install Bluez and radvd using *apt-get*:

```
apt-get install radvd
apt-get install bluez
apt-get upgrade
apt-get update
```

IPv6 forwarding is needed to let the end nodes discover each other through the central node in the star network. To activate this, uncomment the following line (remove "#") in the file */etc/sysctl.conf*

```
net.ipv6.conf.all.forwarding=1
```

To find the IPv6 prefix in the network, run the command *ifconfig*. Find a field named *inet6 addr*, and write down the first and last number on this line (For instance 2001 and /64). The communication will in this case go through a custom designed interface. This will be named bt0. Start by creating the *radvd.conf*-file, and open it for editing.

```
touch /etc/radvd.conf
pico /etc/radvd.conf
```

Write in the following bt0 interface. Replace the number 2001 and /64 with the numbers found in the previous step.

```
interface bt0
{
    AdvSendAdvert on;
    prefix 2001::/64
    {
        AdvOnLink off;
        AdvAutonomous on;
        AdvRouterAddr on;
    };
};
```

To mount the modules *bluetooth_6lowpan, 6lowpan and radvd*, add the following to */etc/modules*. If the file does not exist, create it by entering *touch /etc/modules* first.

```
bluetooth_6lowpan
6lowpan
radvd
```

When the system is booted, these modules will be automatically loaded. The *hcitool* command should now be available. This is a tool designed to connect and keep track of connected devices, both through standard bluetooth and BLE.

```
hcitool lescan
```

lescan will scan for BLE devices nearby, and find the bluetooth address, for instance *00:AA:11:BB:22:CC*. The normal procedure in this case would be to run the following command:

```
echo 1 > /sys/kernel/debug/bluetooth/6lowpan_enable
\color{red}{hcitool lecc 00:AA:11:BB:22:CC}
service radvd restart
```

These commands never established a stable connection in this system. It was not possible to test the connection, and each connected device became automatically disconnected after about 15 seconds. The reason for this was never found. Instead it was possible to not use *hcitool* for this part. Instead, the following commands worked fine:

```
cd /sys/kernel/debug/bluetooth
echo 1 > 6lowpan_enable
echo "connect 00:AA:11:BB:22:CC 1" > 6lowpan_control
service radvd restart
```

The command *hcitool con* shows the connected BLE devices. If the device is connected, the connection can be tested by typing:

```
ping6 2001::02AA:11FF:FEBB:22CC
```

Note that *2001::02AA:11FF:FEBB:22CC* is the full IPv6 address of the device when the bluetooth address is *00:AA:11:BB:22:CC* in this system. The IPv6 address can be used to route packets using 6LoWPAN. Using the basic examples provided by Nordic Semiconductor described in chapter 3.3, it was now possible to send messages both using CoAP CON and NON.

Computational power is closely related to power usage. The nRF52 microcontroller is battery powered using a small *3V Lithium CR 2032* battery. Given this limitation the computational power will be limited as well. The optimal solutions therefore seems to handle as little data as possible here.

3.2 Raspberry Pi to Network Computer or Server

When running the Linux based OS Ubuntu Mate, the raspberry Pi can be used more or less like a regular computer. This OS has a pre-installed version of the most basic programs needed, for instance *Mozilla Firefox Browser*, *Pluma text editor* and *Linux terminal*. In this system it has been connected to the LAN using either a wireless or a wired connection. This makes the link from the Pi to another computer very stable and quick, capable of much higher transfer rates than the other links discussed in the system. Since neither the Pi nor the central computer is battery powered in the testbit, this is not an option. Using a forwarding script it is simple to forward data either to the computer or directly to a location on the web, as shown in 3.1. All these factors implicates that these links will not be a bottle neck in the system. Since this is higher level computer programming and not as limited concerning computational power or battery usage, it will be more interesting to look at the other links in more detail in this thesis. To test these links with higher capacity in more detail will therefore be left to future works, discussed in chapter 6.

3.3 Connecting nRF52 and ADXL345

The used ADXL345 accelerometer was connected using I2C, which is supported by the nRF52. Connection scheme is as follows (nRF52 $\rightarrow$ ADXL345 accelerometer):

Table 3.1: Connection scheme nRF52 to ADXL345

Pin connection	**Explanation**
5V - VIN	Power source, **green cable** in Figure 3.2
GND - GND 0	Ground, **red cable** in Figure 3.2
P0.27 - SDA	I2C Serial Data Line, **orange cable** in Figure 3.2
P0.26 - SCL	I2C Serial Clock Line, **brown cable** in Figure 3.2

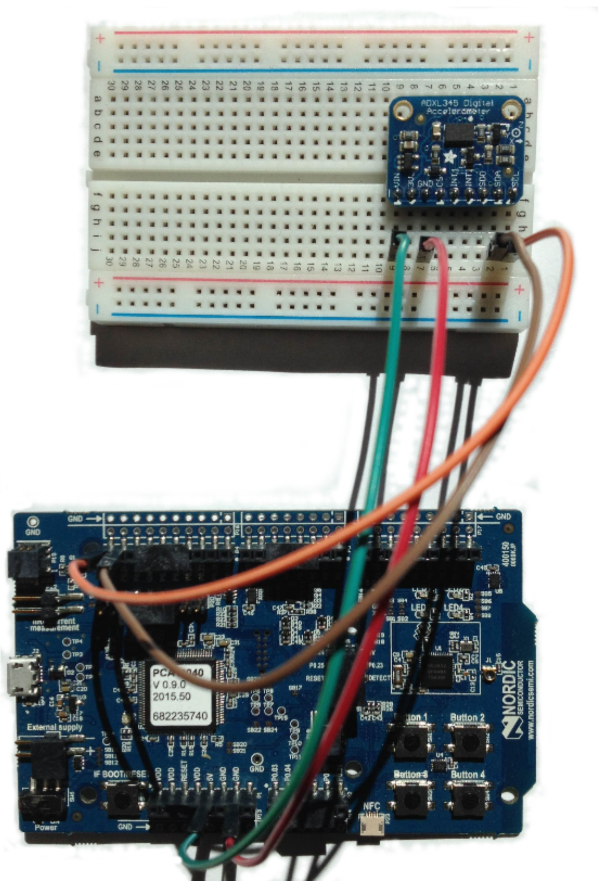


Figure 3.2: Connected nRF52 – ADXL345

nRF52 supports both I2C and SPI serial computer buses. I2C was chosen in this case, because it is fast enough, flexible and simple to set up with use of few cables. As seen in table 3.1 and in figure 3.2, this interface only requires four cables, for power, ground, data and clock. This gives a bandwidth of 1 bit and a maximum bitrate of 5 Mbit/s [21]. The is approximately the same as the assumed max rate capable from the nRF52.

After the physical connection was complete, it was possible to start the process of initializing the ADXL345 accelerometer. Acceleration values can only be read from this sensor if this has been correctly initialized at compile time. In order to do so, code to write to and read from the registers had to be added. To establish this communication, another example from Nordic Semiconductor was used as a starting point, named *TWI master with TWI slave*. By using specific methods from this example and writing to the right accelerometer registers in the right order [12], it was possible to configure the accelerometer as wanted. The detailed description of the programming code used to do this can be seen in appendix C.

It turned out to be difficult and very time consuming to configure the ADXL345 accelerometer to work as expected with the nRF52. In short, registers for *data format control, initial power saving, interrupt enable control* and *the offset of each axis* has to be written to in that order. After this, the acceleration value from the different axes can be read. It was then possible to read from the registers containing current acceleration values using the following code:

```
static uint16_t read_reg(uint8_t register_address, uint8_t data_returnValue)
{
  uint16_t rd;
  ret_code_t ret;
  uint8_t buff[2];
   uint8_t addr8 = (uint8_t)register_address;

   ret = nrf_drv_twi_tx(&m_twi_master, ADXL345_SLAVE_ADDRESS, &addr8, 1,
       true);
   if(NRF_SUCCESS != ret)
   {
       break;
   }
   ret = nrf_drv_twi_rx(&m_twi_master, ADXL345_SLAVE_ADDRESS, buff, 2,
       false);
  rd = (uint16_t)(buff[0] | (buff[1] << 8));

   return rd;
}
```

In the solution proposed in this thesis the acceleration values are being read as often as possible, limited by the processing power of the nRF52 and the I2C connection. Furthermore, the read value is being stored in a simple dynamic char array in the nRF52 before being sent and reset when the BLE channel is ready. The highest obtained measurement frequency in this system was 11 times per main loop, and 150 within this, resulting in 1650 measurements every second. The ADXL345 accelerometer updates its acceleration value when instructed by the master, and the default setting is to follow the oscillator *tick* of the nRF52. This gives an update approximately every second. The result was that even though the register was being read as often as possible, the same value was read up to 1650 times before it was updated.

To solve this problem the default setting of updating the register when told by the oscillator needed to be changed. This turned out to be very time consuming and hard to solve in a proper way, both because of problems with initializing the accelerometer correctly and making the nRF52 read and store the values fast enough to get proper data. The ideal solution would be to read at least 1000 values every second, to get a good starting point before analysing values. At this point it was not possible to get enough real data to be used in data analysis at another point in the system. In order not to loose too much time on hardware problems, it was decided to focus more on analysing the data sent over the network communication with random generated data.

The next chapter will describe the data analysis of the data sent through this network in detail, and how to optimize the percentage of usable data being transported.

3.4 Discussion

Now the full system shown in figure 3.1 has been connected. Due to problems explained in the previous section, the rest of the thesis will focus mainly on the link between the nRF52 and the Raspberry Pi, with the option of using extra computational power from the stationary computer or a web service if necessary. The central point of discussion at this point is how to process and analyse data in the system. The main options to consider in all the different devices concerning how much computation to do in this link are the following:

- No computation: All data arrives as useful data, and can be posted directly to a web page or a server for storage
- No computation: Forward all data directly to a computer with more computational power
- Some computation: Analyse the data to find data that is not relevant to filter out
- Full computation: Do a full analysis of the data. The results can then be posted directly to a server or displayed on a web page.

The most relevant option of these four depends on the data, and on how much computational power is needed. It is possible to run the Raspberry Pi from a power bank, but this has not been tested in this project. When set up without a battery as power source, the Pi is the first node that could possibly do computations without having to take power limitations as a major concern. The main limitation is therefore computational power, while the main limitation may be battery power on the nRF52. It therefore makes sense to do some easy computation on this device. For instance, if this network is being used to measure vibrations, it is reasonable to assume that measurements more frequent than once every 100 *ms* is needed. Any less frequent than this and vibrations could be missed, especially if it is periodical. It would then be perfectly reasonable to assume that the Pi could go through these values, and calculate whether or not the current acceleration value has breached a given threshold. This result can then be displayed directly on a web page or a connected monitor from the Pi. If however the system is to calculate *patterns* in the acceleration values, several values needs to be compared together. The need of complex algorithms to find these patterns is expected, before the results can be displayed. In this case it is reasonable to assume that the Pi would need additional computational power. The Pi can then be set up as a forwarding device, that forward data directly to a computer with more computational power.

Chapter 4
Network Measurements

This chapter will display the experiments carried out concerning data sending rate from the nRF52 to the Raspberry Pi in the form of graphs, tables and figures. The goal is to determine the most efficient combination when it comes to amount of data to send in one transmission, sending frequency and protocols to use in the different scenarios.

4.1 Description of measurements

The following sections will discuss if *fragmentation* is a major issue when sending data through low energy networks, and which of the protocols described in chapter 2 is the most efficient to use when the goal is to get as high amount of payload as possible compared to the total throughput. Data will be sent through the network, containing a payload of constant length. Wireshark will be a central part of the measurements presented, and the packet size will be increased to see the changes. These problems are essential in the discussion of objective O.4.

Before these experiments started, the expected results were that sending a small amount of data at a time would not be preferable, because of the needed bytes to set up the connection, header files and so on. It was not known the packet size needed before it would be considered profitable to send them regarding the cost of energy and network capacity. This is dependent on the situation where the network will be used. A system with sensors to analyse real-time patient data to see if a patient is in a stable state needs to be reliable, and the data will be sent no matter if it is profitable for the network or not. In this case, timing is the most important. A system used in a company to monitor how many cups of coffee are being drunk during a day can easily store data in the end node and send larger amounts of data less frequently, if this is profitable for the network.

When sending BLE packets over the network, observations from the system show that the maximum packet size over one BLE frame is 31 bytes in this system. Each of these packages needs a header field of 4 bytes, meaning 27 bytes left for useful data. However, to start the connection at all, 76 bytes are needed, meaning three BLE packets. The ratio between *useful* and *needed* data transferred start out very poorly if the payload sent is very small. The best possible percentage of useful data we can hope to achieve will also be limited by this. 27 bytes payload and 4 bytes header field is 87,1 %, calculated in the following equation:

$$lim_{x\to\infty}\frac{x+27}{x+31+C}=\frac{27\,byte}{31\,byte}*100\ \approx 87,1\,\% \tag{4.1}$$

Tjaa, ikke helt enig? Forklar bedre at det er header?

During measurements in the physical system, the actual result will probably be considerably lower than this, because 6LoWPAN and CoAP packets could require some additional fields for each packet. There are also other additional protocols in BLE that will require some packets, like the occasional ACKs from ACL. It is also logical to expect some other disturbing factors in a real world wireless network.

4.2 Possible limitations in the network

4.2.1 Stable transfer rate

As soon as the end nodes of the network could communicate with the Raspberry Pi using CoAP, the next step was to test the transfer rate of the connection. To measure the network transfer rate, *ping6* was used. This is a software tool used to test networks using IPv6 in a network. Results from these measurements are being shown as *ms* used for every Round Trip Time.

```
ping6 2001::2e6:6aff:fe64:54dd
ping6 google.com
```

To receive messages using CON, a CoAP GET-message is sent from a requesting client. As soon as the response has been received at the client-side, another GET message is being sent, which means there is always either a GET-message or a message containing a payload in the network, given that the end node has a sensor connected that provides streaming data.

In NON, there are two different options. A new message can be sent when the set field *Max-Age* has expired, which tells the number of seconds the system should wait before sending a new packet [22]. The other choice is to send a new message every time a given field has been updated. This would have been the best solution in the testbit, but it was never possible to update this field using accelerometer data, as expected in this case. The only choice was to set the Max-Age to 1 second, which was the lowest setting leading to a stable . This gives a stable and reliable transfer frequency at 1 second, even though this is a limitation compared to the sending frequency in CON. After a test period it was therefore decided that the best solution for NON would be to gather data from sensors at a higher rate, and store them in the nRF52 temporarily. Every second all the measured values are being transferred to the Raspberry Pi, the temporarily values are deleted and the measurement continues. This has proven to be a very stable solution, with successful tests over several days. CON can handle more frequent transportations than this in this system, on average 4 times per second. See the test shown in chapter 4.6. Figure 2.8 shows the initial set up of a NON connection, where the stable transfer rate of one second can be seen at the timestamps at the bottom of the figure.

Results from these first tests gave an approximate average of 250 ms RTT. The standard deviation in these cases was on average $\sigma = 25\,ms$, which is a variation of 10 %. In total 10 different measurements at different time were performed, an example can be seen in figure 4.1. These measurements were performed using different versions of nRF52s, and a different amount of these devices connected to the central Raspberry Pi at once. This is considered quite slow in such a system, and way beneath the transfer limitations of both BLE, 6LoWPAN and CoAP. Another factor could be limited power supply and computational power, but it is not clear what is the main cause at this point. This will regardless be a major limitation in the network.

Tja, staar det noe i background egentlig?

```
64 bytes from 2001::2af:b7ff:feb6:1494: icmp_seq=156 ttl=64 time=242 ms
64 bytes from 2001::2af:b7ff:feb6:1494: icmp_seq=157 ttl=64 time=228 ms
64 bytes from 2001::2af:b7ff:feb6:1494: icmp_seq=158 ttl=64 time=270 ms
64 bytes from 2001::2af:b7ff:feb6:1494: icmp_seq=159 ttl=64 time=251 ms
64 bytes from 2001::2af:b7ff:feb6:1494: icmp_seq=160 ttl=64 time=230 ms
64 bytes from 2001::2af:b7ff:feb6:1494: icmp_seq=161 ttl=64 time=280 ms
64 bytes from 2001::2af:b7ff:feb6:1494: icmp_seq=162 ttl=64 time=260 ms
64 bytes from 2001::2af:b7ff:feb6:1494: icmp_seq=163 ttl=64 time=240 ms
64 bytes from 2001::2af:b7ff:feb6:1494: icmp_seq=164 ttl=64 time=289 ms
64 bytes from 2001::2af:b7ff:feb6:1494: icmp_seq=165 ttl=64 time=270 ms
64 bytes from 2001::2af:b7ff:feb6:1494: icmp_seq=166 ttl=64 time=326 ms
64 bytes from 2001::2af:b7ff:feb6:1494: icmp_seq=167 ttl=64 time=228 ms
64 bytes from 2001::2af:b7ff:feb6:1494: icmp_seq=168 ttl=64 time=285 ms
64 bytes from 2001::2af:b7ff:feb6:1494: icmp_seq=169 ttl=64 time=266 ms
64 bytes from 2001::2af:b7ff:feb6:1494: icmp_seq=170 ttl=64 time=237 ms
^C
--- 2001::2af:b7ff:feb6:1494 ping statistics ---
170 packets transmitted, 170 received, 0% packet loss, time 169172ms
rtt min/avg/max/mdev = 214.144/253.094/347.238/26.557 ms
sindre@PiMATE:~$
```

Figure 4.1: Ping nRF52 from Raspberry Pi

The conclusion from these initial tests are therefore that CoAP CON can be used at at lower transfer interval than CoAP NON in the testbit. CON can in best case send a message every 250 *ms*, which is as expected compared to the high RTT measured in this network. An example of one of the RTT tests can be seen in figure **??**. NON has shown the same results regarding RTT, but the connection has in general been more unstable in the initial tests, even though Max-Age means data can only be sent every second. From this starting point it is expected that CON can send a larger payload per second for small payloads, but NON require less of the network to send each message. In addition it is expected that NON will get a higher percentage of payload compared through the total throughput for larger payloads, since no ACKs are required in this solution.

4.2.2 Packet fragmentation

In Internet Routing, fragmentation is known as the action of splitting data into smaller packets, to satisfy the maximal limits of the different technologies or protocols used (e.g. BLE and 6LoWPAN in the testbit). Each of these packets needs header fields of a certain size, or other requirements. In a network of microcontrollers, fragmentation can be a factor that needs to be taken into account to optimize the payload sent through the system. To better understand fragmentation, imagine a train with carriages as shown in Figure 4.2. To be able to operate at all, the train needs a locomotive with an engine driver, a conductor and a cafe carriage. As soon as these things are already there, the company owning the train gets better and better off for every passenger buying a ticket. Lets assume that every carriage can carry 4 employees and 27 passengers, to make it directly comparable to the BLE packets in the network. Eventually all the carriages will be full, and a decision has to be made if it will be profitable to fit another carriage. It will in general be most profitable to use as many carriages as the locomotive can handle, and to fill up every carriage as much as possible. It will however not be a good idea to connect another carriage if there will only be one additional passenger sitting there, since the extra weight of the carriage adds unnecessary additional weight to the train set compared to the income.

In this example, the locomotive and employees are the 6LoWPAN packet, that are needed no matter what to get the train working. Each additional carriage is a BLE packet. The goal is therefore to find the maximal number of passengers compared to the cost of adding additional carriages, in other words the maximum number of bytes compared to the number of sent packets. This is known as fragmentation, fragmenting data into smaller pieces to satisfy the maximal limitations of packet sizes in the different protocols. This can be exploited by a system developer to maximize the percentage of payload compared to throughput in the network.

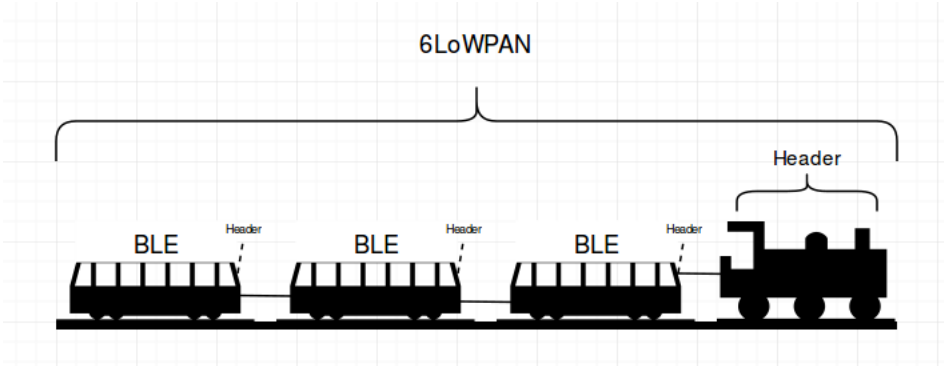


Figure 4.2: Packet fragmentation - train comparison

4.3 Measurements

4.3.1 CoAP CON

As previously explained, CoAP can be split into two different main sections. CON messages can in the testbit be sent quite frequently, but every message needs to get an ACK before the next message can be sent. This means that it has the possibility of being quite fast, but several extra packets needs to be transported through to get usable data at the other end of the link. The other alternative is NON, where each message does not need an ACK.

In *Wireshark* it is possible to display all captured packets, or filter packets regarding what protocol they are using (e.g. TCP or CoAP). The following examples will only focus on packets sent using CoAP in addition to capturing all bluetooth packets. This will show how the fragmentation of CoAP packets needs to be done in order to fit into the size of BLE packets. When measuring packet sizes concerning fragmentation of the network, all measurements of the same constant payload gave the same result, since fragmentation is being handled the same way every time. These examples are therefore taken from one of the experiments, even though several where done[1].

Table 4.1 shows the most basic example of a capture of packets in Wireshark using CON. The full capture can be seen in the Appendix B. In this case an empty *char* array was sent, meaning a payload equal to 0 byte. As a consequence, all the captured bytes correspond solely to data sent by the network protocols. The 31 bytes per BLE packet, the maximum for BLE in the system, have been exceeded twice. Therefore, three packets were needed. The first packet is labeled *[Reassembled in #40]*, the second *[Continuation to #38]* and the last *Connection oriented channel*. Then the

[1]All the measured data can be found on https://www.github.com/sische/MasterThesis

Table 4.1: Wireshark CoAP CON 0 bytes payload

Number	Time	Protocol	Length	Info
36	3.7471	CoAP	72	ACK, MID:57083, 2.05 Content
37	3.7759	CoAP	113	CON, MID:57084, GET
38	3.9571	HCI_ACL	31	Rcvd [Reassembled in #40]
39	3.9584	HCI_ACL	31	Rcvd [Continuation to #38]
40	4.0274	L2CAP	5	Rcvd Connection oriented channel
41	4.0367	L2CAP	58	Sent Connection oriented channel
42	4.0368	L2CAP	50	Sent Connection oriented channel
43	4.0975	L2CAP	16	Rcvd LE Flow Control
44	4.0977	HCI_EVT	7	Rcvd Number of Completed Packets
45	4.1678	HCI_EVT	7	Rcvd Number of Completed Packets
46	4.0275	CoAP	72	ACK, MID:57084, 2.05 Content
47	4.0366	CoAP	113	CON, MID:57085, GET

ACK packages follows, two packages of 58 and 50 bytes, respectively. A final pair of packets tells how many packages were completed, as a built in feature in BLE HCI and ACL. All of these packages can fit into one 6LoWPAN packet, since the total number of bytes are less than 270 bytes.

Say 0% goodput somewhere in here?

Table 4.2: Wireshark CoAP CON 100 bytes payload

Number	Time	Protocol	Length	Info
29	2.4514	HCI_ACL	31	Rcvd [Reassembled in #36]
30	2.4516	HCI_ACL	31	Rcvd [Continuation to #29]
31	2.2425	CoAP	173	ACK, MID:16354, 2.05 Content
32	2.2538	CoAP	113	CON, MID:16355, GET
33	2.5217	HCI_ACL	31	Rcvd [Continuation to #29]
34	2.5218	HCI_ACL	31	Rcvd [Continuation to #29]
35	2.5907	HCI_ACL	31	Rcvd [Continuation to #29]
36	2.5921	L2CAP	25	Rcvd Connection oriented channel
37	2.6099	L2CAP	58	Sent Connection oriented channel
38	2.6100	L2CAP	50	Sent Connection oriented channel
39	2.6610	L2CAP	16	Rcvd LE Flow Control
40	2.6621	HCI_EVT	7	Rcvd Number of Completed Packets
41	2.5922	CoAP	173	ACK, MID:16355, 2.05 Content
42	2.3097	CoAP	113	CON, MID:16356, GET
43	2.7311	HCI_EVT	7	Rcvd Number of Completed Packets

In table 4.2, 100 bytes of payload is being sent through the network using CON. The same basic packages are still needed there, but in addition 100 bytes of data is added. This means adding more BLE packets, but also that the percentage of useful data sent through is higher, approximately 34 % in this case, as seen in equation 4.2. By doing several experiments like this, it was possible to create the graph in Figure 4.3. This shows the correlation between payload and throughput compared to the number of packets sent, measured every 10th byte from 0 byte to 200 bytes large packets.

$$\frac{100\,byte\,goodput}{180\,byte\,throughput +\ 113\,byte\,ack} * 100\ \approx 34\,\% \tag{4.2}$$

In this particular case shown in figure 4.3, it makes no sense to send less than 50 bytes of useful data at once, since more than 50 % of the bytes sent will be header files. This is comparable to having a locomotive and full crew at disposal, but only a few or none paying passengers. The best possible result is to have every carriage full, with 27 passengers and 4 employees. Since at least 4 bytes out of every 31 sent needs to be used to header information, the best possible result will be 87,1 %. In mathematics, this is described as a *horizontal asymptote* since the distance between the graph and y = *87,1* will approach zero after an infinite number of bytes

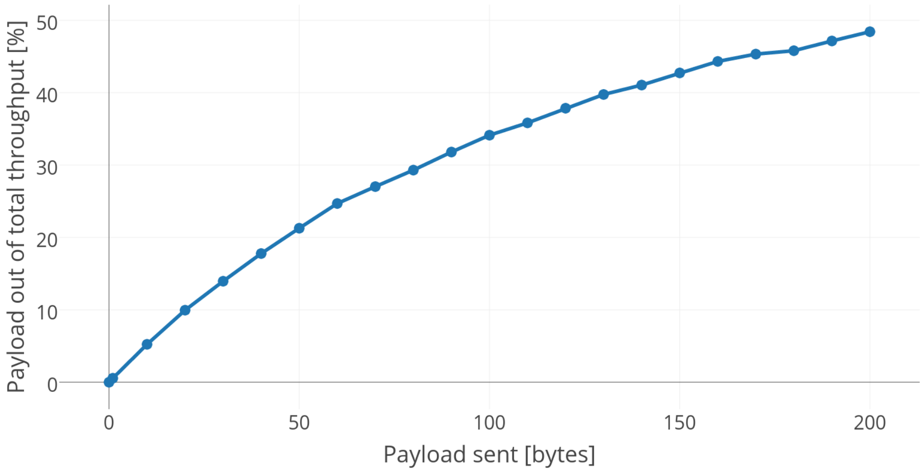


Figure 4.3: CoAP CON with ACKs, 0-200 bytes sent

has been transferred, if the only limitation was BLE packets. In this network other limitations like 6LoWPAN header files needs to be considered as well. The graph will still approach 87,1 %, which was calculated in equation 4.1, just as the values climbing in figure 4.3, even before the payload has reached 200 bytes.

$$\frac{100\, byte\, goodput}{180\, byte\, throughput +\ 113\, byte\, ack} * 100 \approx 34\,\% \tag{4.3}$$

4.3.2 CoAP NON

A CoAP NON request does not require a response in form of an ACK to each CoAP packet being sent. This means that the 108 bytes sent to and handled by the end node can be skipped. In the end nodes this leads to less computational power, and less network capacity is needed to transfer data both ways. This solution makes sense to use in networks where the system will still work as needed even if some packets are being dropped, since packets can be lost without the use of ACKs. As explained in chapter 4.1.1, the transfer frequency is limited using NON in this system, due to the use of Max-Age instead of GET-requests. The transfer rate is therefore set to one per second for this protocol.

A basic example of the CoAP NON connection is shown in table 4.3[2]. This is directly

[2]The entire Wireshark capture can be seen in Appendix B

Table 4.3: Wireshark CoAP NON 0 bytes payload

Number	Time	Protocol	Length	Info
90	23.0405	HCI_ACL	31	Rcvd [Reassembled in #92]
91	23.0411	HCI_ACL	31	Rcvd [Continuation to #90]
92	23.1107	L2CAP	9	Rcvd Connection oriented channel
93	23.1109	CoAP	76	NON, MID:14, 2.05 Content

comparable to table 4.1, that shows a Wireshark capture of CoAP CON packages. It is easy to see that a lot fewer packages needs to be sent using BLE, without the use of ACKs. The total amount of bytes sent is 31+31+9=71 bytes, meaning three BLE packages sent in one 6LoWPAN packet. CON has a packet size of 76 bytes in this transmission, which means a total of 5 header bytes and additional fields are needed. This is less than half of what was needed using CoAP CON, where the 108 ACK packets were needed in addition. The packets are still recognizable the same way as before when captured in Wireshark. The first packet is labelled *[Reassembled in #40]*, the second *[Continuation to #38]* and the last *Connection oriented channel*.

Fewer packets sent means less energy used in end nodes, less network capacity needed and less computational power in the end node. This approach will hopefully lead to a higher percentage of payload compared to throughput. Therefore, tests were set up to measure this with payload sizes between 0 and 200 bytes, measuring with an increasing interval of every 10 bytes. This will compare how the correlation between payload and throughput developed as the payload size increased.

Table 4.4: Wireshark CoAP NON 100 bytes payload

Number	Time	Protocol	Length	Info
39	11.0363	CoAP	177	NON, MID:2, 2.05 Content
40	11.9452	HCI_ACL	31	Rcvd [Reassembled in #45]
41	11.9465	HCI_ACL	31	Rcvd [Continuation to #40]
42	12.0154	HCI_ACL	31	Rcvd [Continuation to #40]
43	12.0168	HCI_ACL	31	Rcvd [Continuation to #40]
44	12.0857	HCI_ACL	31	Rcvd [Continuation to #40]
45	12.0858	L2CAP	29	Rcvd Connection oriented channel
46	12.0860	CoAP	177	NON, MID:3, 2.05 Content

Table 4.4 shows the case where 100 bytes of data are sent using CoAP NON. This is directly comparable to the test shown in table 4.2, where the same amount of data is sent using CON. The overall structure is the same as before. Since it is only one packet noted *Reassembled in #n*, which marks the beginning of a new 6LoWPAN

packet. The total amount sent must therefore be under 270 bytes, which makes sense. The NON packet size is here 177 bytes, compared to 173 bytes in CON, meaning that a NON packet require an additional header field of 4 bytes. Overall a small difference in the packets containing data, but as expected a lot more packets needs to be sent in total in CON. Using these measurements the following plot could be drawn.

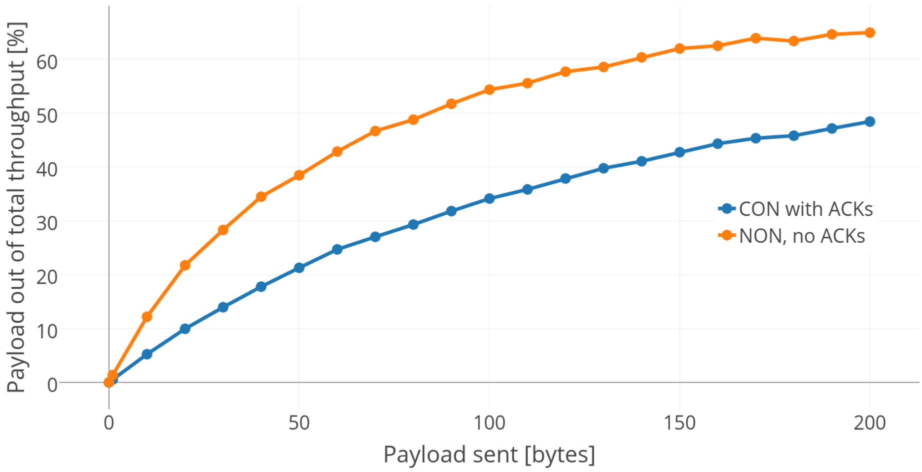


Figure 4.4: CON with ACKs vs NON 0-200 bytes

Figure 4.4 shows the comparison between sending between 0 and 200 bytes of useful data through the network. As expected from the Wireshark captures in the four previous tables, there are no major differences between these graphs when it comes to payload out of total throughput. The tables show that NON requires a few more bytes for each packet, which gives the difference between the two in this plot.

4.3.3 Discussion

Even though these first tests were carried out with a very limited amount of data transferred, it is easy to see that the curves clearly flattens out and forms the shape of a *parabola* with a vertical *directrix* at $x = 0$. Several BLE packets have been fragmented and sent during these test, without the graph showing a special payload size where this gives a noticeable result. This means that the fragmentation of BLE packets does not have a major impact of the % of payload compared to throughput in this system. More data needs to be sent at once to check if the same can be said for fragmentation of 6LoWPAN packets. The next step will be to transfer a larger amount of data, to verify that the assumptions that the graph will converge to the asymptotic value $y = 87{,}1$ when $lim_{x \to \infty}$. The limitations of transfer rate is not nearly yet met by neither BLE or 6LoWPAN. These tests will therefore be explained in the next section.

To see what happens when the limit of a 6LoWPAN packet was breached in the system, tests will be set up to send larger amounts of data than 200 bytes. The following test and examples will therefore send a fixed number of bytes at once from 0 to 1000 bytes (1 kB), with a 50 byte interval. This will also be a good way to test if the percentage of payload compared to throughput will converge to 87,1 %, only considering BLE packets. As the previous tests, CON will transfer data using GET-messages, and therefore request a new message as soon as the ACK of the previous message has been received. NON has a fixed Max-Age value at 1 second, that determines the frequency of sending NON packets.

4.3.4 Tests with more data

Table 4.5 shows a bigger and more complex case, where a payload of 700 byte is being sent at once using CoAP CON. In total 889 bytes are being sent in this process, with a CON packet size of 774 bytes. This gives a percentage of payload compared to throughput at 78.74 %. The maximum packet length of a 6LoWPAN packet at 127 bytes and the max in the testbit of 270 bytes are therefore exceeded. This can be seen in the table, after 8 BLE packages of 31 bytes each, have been sent, there is only room for 22 bytes in the last packet before the 6LoWPAN packet reaches its maximal capacity. This is repeated several times, once for each 6LoWPAN packet, until the last BLE packets at the size of 17 bytes. After this the standard packages for ACK and *Number of completed packages* follows. This is a good example of how fragmentation of packets works in this system.

Table 4.5: Wireshark CoAP CON 700 bytes

Number	Time	Protocol	Length	Info
53	3.2570	CoAP	774	ACK, MID:35081, 2.05 Content
54	3.2727	CoAP	113	CON, MID:35082, GET
55	3.4671	HCI_ACL	31	Rcvd [Reassembled in #63]
56	3.4747	HCI_ACL	31	Rcvd [Continuation to #55]
57	3.5374	HCI_ACL	31	Rcvd [Continuation to #55]
58	3.5375	HCI_ACL	31	Rcvd [Continuation to #55]
59	3.6077	HCI_ACL	31	Rcvd [Continuation to #55]
60	3.6078	HCI_ACL	31	Rcvd [Continuation to #55]
61	3.6767	HCI_ACL	31	Rcvd [Continuation to #55]
62	3.6782	HCI_ACL	31	Rcvd [Continuation to #55]
63	3.7533	L2CAP	22	Rcvd Connection oriented channel
64	3.7534	L2CAP	16	Sent LE Flow Control
65	3.8172	HCI_EVT	7	Rcvd Number of Completed Packets
66	3.8172	HCI_ACL	31	Rcvd [Reassembled in #74]
67	3.8185	HCI_ACL	31	Rcvd [Continuation to #66]
68	3.9575	HCI_ACL	31	Rcvd [Continuation to #66]
69	3.9577	HCI_ACL	31	Rcvd [Continuation to #66]
70	4.0266	HCI_ACL	31	Rcvd [Continuation to #66]
71	4.0342	HCI_ACL	31	Rcvd [Continuation to #66]
72	4.0979	HCI_ACL	31	Rcvd [Continuation to #66]
73	4.0982	HCI_ACL	31	Rcvd [Continuation to #66]
74	4.1671	L2CAP	22	Rcvd Connection oriented channel
75	4.2372	HCI_ACL	31	Rcvd [Reassembled in #83]
76	4.2373	HCI_ACL	31	Rcvd [Continuation to #75]
77	4.3075	HCI_ACL	31	Rcvd [Continuation to #75]
78	4.3076	HCI_ACL	31	Rcvd [Continuation to #75]
79	4.3777	HCI_ACL	31	Rcvd [Continuation to #75]
80	4.4466	HCI_ACL	31	Rcvd [Continuation to #75]
81	4.4480	HCI_ACL	31	Rcvd [Continuation to #75]
82	4.4481	HCI_ACL	31	Rcvd [Continuation to #75]
83	4.5170	L2CAP	22	Rcvd Connection oriented channel
84	4.5871	HCI_ACK	31	Rcvd [Reassembled in #86]
85	4.5875	HCI_ACL	31	Rcvd [Continuation to #84]
86	4.6574	L2CAP	17	Rcvd Connection oriented channel
87	4.6731	L2CAP	58	Rcvd Connection oriented channel
88	4.6732	L2CAP	50	Rcvd Connection oriented channel
89	4.6577	CoAP	774	ACK, MID:35082, 2.05 Content
90	4.6729	CoAP	113	CON, MID:35083, GET

Table 4.6: Wireshark CoAP NON 700 bytes

Number	Time	Protocol	Length	Info
264	57.1476	CoAP	777	NON, MID:3, 2.05 Content
265	57.2177	HCI_ACL	31	Rcvd [Reassembled in #273]
266	57.2178	HCI_ACL	31	Rcvd [Continuation to #265]
267	57.2879	HCI_ACL	31	Rcvd [Continuation to #265]
268	57.2943	HCI_ACL	31	Rcvd [Continuation to #265]
269	57.3570	HCI_ACL	31	Rcvd [Continuation to #265]
270	57.3583	HCI_ACL	31	Rcvd [Continuation to #265]
271	57.4272	HCI_ACL	31	Rcvd [Continuation to #265]
272	57.4286	HCI_ACL	31	Rcvd [Continuation to #265]
273	57.4975	L2CAP	22	Rcvd Connection oriented channel
274	57.4979	L2CAP	16	Sent LE Flow Control
275	57.5676	HCI_ACL	31	Rcvd [Reassembled in #284]
276	57.5685	HCI_EVT	7	Rcvd Number of Completed Packets
277	57.5753	HCI_ACL	31	Rcvd [Continuation to #275]
278	57.6378	HCI_ACL	31	Rcvd [Continuation to #275]
279	57.6379	HCI_ACL	31	Rcvd [Continuation to #275]
279	57.7080	HCI_ACL	31	Rcvd [Continuation to #275]
280	57.7082	HCI_ACL	31	Rcvd [Continuation to #275]
281	57.7771	HCI_ACL	31	Rcvd [Continuation to #275]
282	57.7785	HCI_ACL	31	Rcvd [Continuation to #275]
283	57.7785	HCI_ACL	31	Rcvd [Continuation to #275]
284	57.8474	L2CAP	22	Rcvd Connection oriented channel
285	57.9176	HCI_ACL	31	Rcvd [Reassembled in #293]
286	57.9176	HCI_ACL	31	Rcvd [Continuation to #285]
287	57.9878	HCI_ACL	31	Rcvd [Continuation to #285]
288	57.9879	HCI_ACL	31	Rcvd [Continuation to #285]
289	58.0580	HCI_ACL	31	Rcvd [Continuation to #285]
290	58.0581	HCI_ACL	31	Rcvd [Continuation to #285]
291	58.1270	HCI_ACL	31	Rcvd [Continuation to #285]
292	58.1284	HCI_ACL	31	Rcvd [Continuation to #285]
293	58.1972	L2CAP	22	Rcvd Connection oriented channel
294	58.2673	HCI_ACK	31	Rcvd [Reassembled in #296]
295	58.2688	HCI_ACL	31	Rcvd [Continuation to #294]
296	58.3378	L2CAP	20	Rcvd Connection oriented channel
297	58.3379	CoAP	777	NON, MID:4, 2.05 Content

Table 4.5 shows a bigger and more complex case, where a payload of 700 byte is being sent at once using CoAP CON. In total 889 bytes are being sent in this process, which consists of four 6LoWPAN packets where three of them is the full 270 bytes. This can be seen in the table, after 8 BLE packages of 31 bytes each has been sent, there is only room for 22 bytes in the last packet before the 6LoWPAN packet has reached its maximal capacity. The last packet contains $31 + 31 + 17 = 79\,bytes$, which leaves 199 unexploited bytes in the packet. Concerning fragmentation, it would probably have been better to send data with a slightly smaller payload. After this the standard packages for ACK and *Number of completed packages* follows. This is a good example of how fragmentation of packets works in this system. The percentage of payload compared to throughput is therefore 700 bytes/889 bytes = 78,74 % in this case.

As a direct comparison, table 4.6 represents the same goodput sent through the network using CoAP NON. In this case, the total is 892 bytes, only three more than the previous example. This confirms that CON and NON works in very similar ways when it comes to packet fragmentation, and the same pattern of 270 bytes 6LoWPAN packets is being recognised. Calculations shows that in this case the percentage is 700 bytes/892 bytes = 78,47 %.

4.3.5 Discussion

Using these results from measurements of payloads between 0 and 1000 bytes it was possible to plot the graph in figure **??**. Here it is easy to see the same trends as in figure 4.3, but over a wider span of sent bytes. These results are as expected after the previous tests, and in compliance with the calculations done concerning horizontal asymptote.

Table 4.5 and 4.6 shows the case where 700 bytes where sent at once through the network, using both CoAP CON and NON. The size of fragmented packet was very similar in both cases and the percentage of payload compared to throughput, but CON needs ACKs in addition.

$$\frac{700\,bytes\,payload}{889\,bytes\,throughput\;+\;113\,bytes\,ack} = 69,86\,\% \tag{4.4}$$

$$\frac{700\,bytes\,payload}{892\,bytes\,throughput} = 78,48\,\% \tag{4.5}$$

$$\frac{69,86}{78,48} \approx 0,8902 \rightarrow 100\% - 89,02\% = 10,98\,\% \tag{4.6}$$

Calculations in equation 4.4 show that the percentage of payload CON is 78,48 %, compared to 69,86 % in CON. The difference between these two is *10,98 %*. This can also be seen clearly in 4.5. Because of this, it was concluded that the results for using NON and CON can be considered as negligible for transmissions larger than 1 kB. Tests with larger amounts of data than this at once did therefore not seem necessary, and will not be investigated in this project.

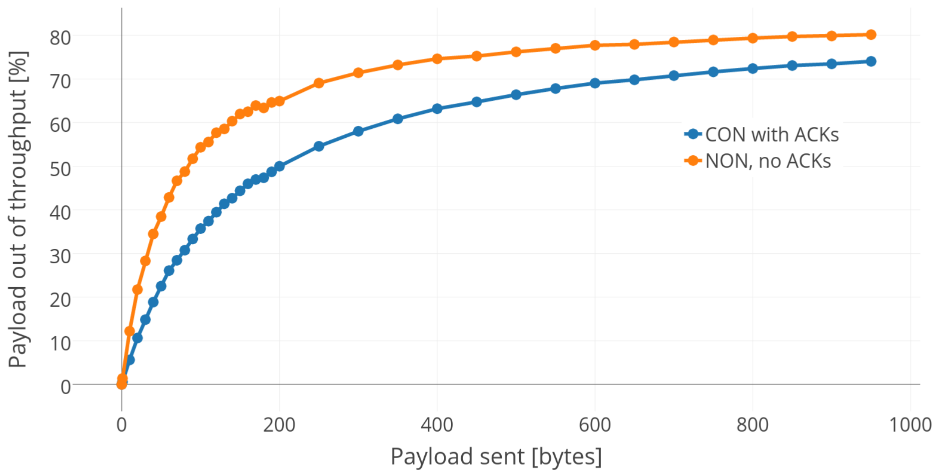


Figure 4.5: CON with ACKs vs NON 0-1000 bytes

Given these results, it can be concluded that to send less data than 200 bytes at the time is not preferable, since the percentage of payload compared to the total amount sent can be very low. On the other hand, the graph stabilizes around 65-70 % for CON and 75-80 % for NON. Given these measurements it looks like 400 bytes and bigger packets are preferable in this system. The system shows no signs of weakness as the packet size grows, and it will therefore in theory be possible to send as large packets as needed until the limitations of BLE with the same amount of goodput at about 80 %. This was however never achieved in this configuration of the network, but should be possible to do in future works. Concerning fragmentation, it was concluded after having sent payloads up to 200 bytes that fragmentation of BLE packets is not a major concern in this network. After having sent up to 1000 bytes to check the same for 6LoWPAN packets, the same thing can be concluded here. Both graphs of CON and NON shown in 4.5 show a flat and stable curve, with no special weak points in the different payloads. This would be expected right after the payload was big enough to exceed a BLE or 6LoWPAN packet, just as explained in chapter 4.2.2. Developers should therefore not consider to minimize the payload just to avoid fragmentation, in such a network.

4.4 Transfer rates

4.4.1 Time used to transfer payload

Previous sections in this chapter have shown the percentage of payload compared to the total throughput, to measure if fragmentation was a major concern in this network. This is a good overview of how the protocols are able to exploit the network, and tells a lot about the different protocols. Developers could possibly use these results to build the IoT network. For end users in a real world scenario however, the actual *throughput per time* would be more relevant, since this tells how much data can be transported every second. A common measuring unit for this is known as *goodput*, measured in bytes per second.

There are several ways to measure throughput compared to time used. For instance by calculating the number of seconds it takes to transfer a known number of bytes, or the number of bytes that are transported on average every second. To calculate this, values from measurements shown in table B.1 and B.2 in appendix B were used. The numbers shown in these tables and used in the following experiments have been measured by sending a considerable amount of packets through the network[3]. In most cases more than 100 CoAP packets was sent for each constant payload, to find the minimum, average and maximum value for both the time used to transfer a packet and the goodput in each case.

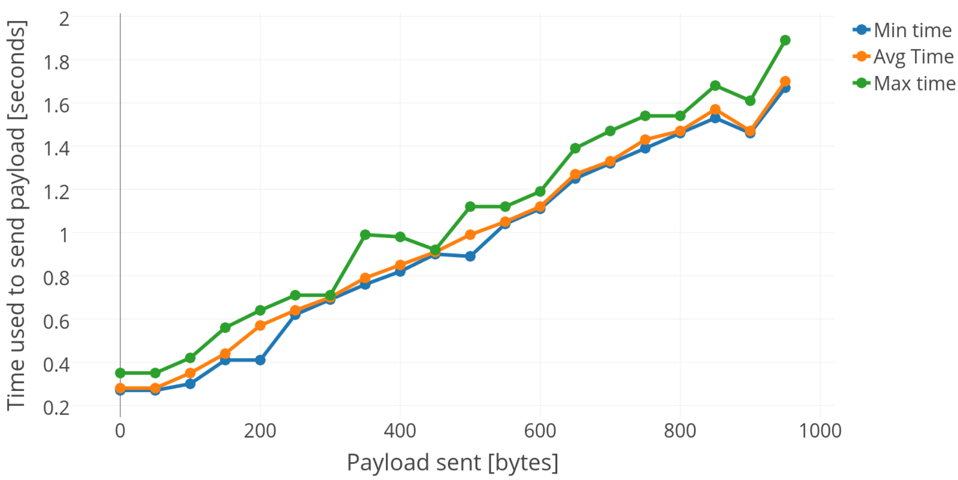


Figure 4.6: Time used to transfer payload CON

[3]All measurements can be seen on GitHub: https://github.com/sische/MasterThesis

Figure 4.6 shows the minimum, average and maximum time it took to send a constant payload through the network using CON. This variation is relevant to see the stability in the network, how much it can be trusted. A system carrying important data needs a stable transfer rate at all times, not only a good average value. In this case the largest deviation from the average value is at 350 bytes, where the max. value is almost 1 second, while the average is approximately 0.7 seconds. Both the average and minimum graph has a quite linear development, with the exception of 900 bytes. The lowest transfer rate for CON is when the payload is below 100 bytes. The transfer time at this payload is about 300 *ms*, which seems logical concerning the ping tests in the beginning of the chapter at 250 *ms*. The slowest transfer time is the max value when transferring 950 bytes, when the time is approaching 2 seconds.

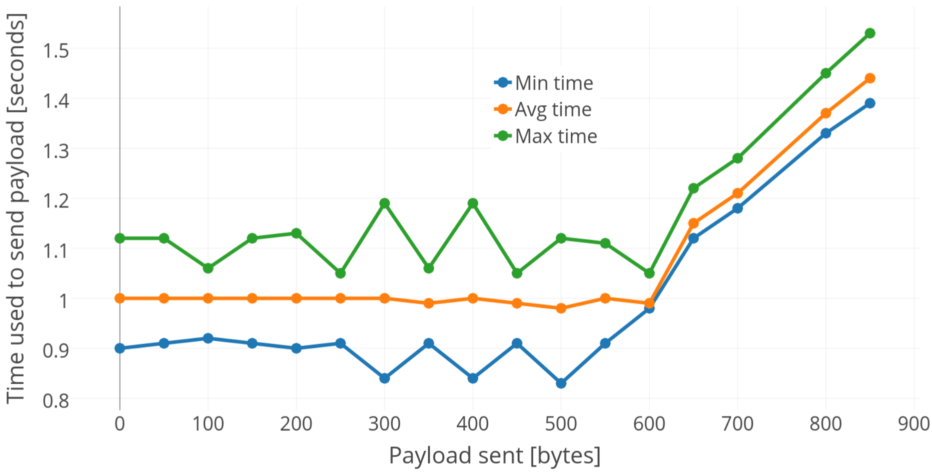


Figure 4.7: Time used to transfer payload NON

Time used to transfer a given payload using NON can be seen in figure 4.7. Even thought there are some slight variations from min. to max. time used, the average transfer frequency is very stable. Because of the described difficulties concerning Max-Age of the measured field in NON, the fastest transfer frequency achieved in the system was 1 second. On average the transfer rate is very close to this, but the min and max values vary from 0.85 seconds to 1.2 seconds in both 300 and 400 bytes payload. 600 bytes is the maximum payload where the system is still able to transfer data at 1 second rate, after this the graph starts to climb for larger payloads. For NON with a payload larger than 650 bytes, the network was too unstable to do the same amount of data as in previous tests. These values (NON $>$ 650 bytes payload) are therefore not as certain as the rest of the tests considering amount of packets sent.

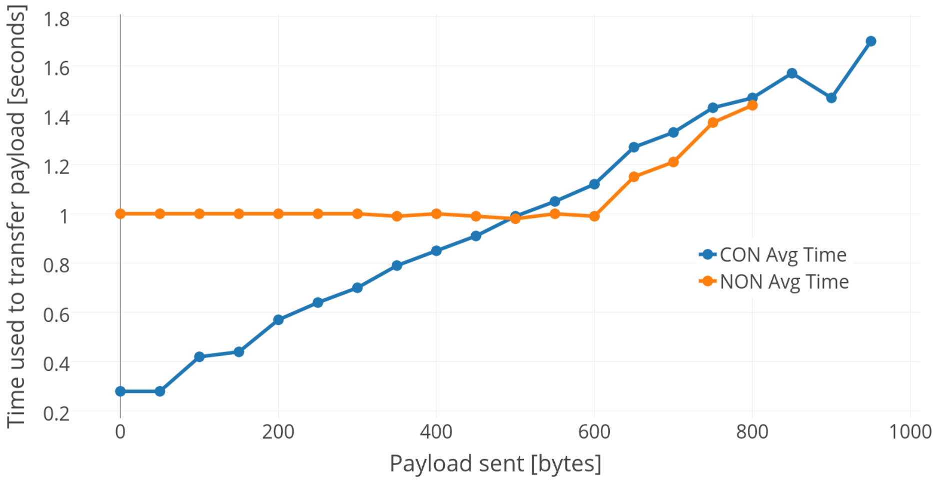


Figure 4.8: Average time to transfer payload, CON vs NON

To get a direct comparison of this data, the average values from CON and NON are being compared in figure 4.8. This makes the differences between the two are easy to spot. It takes 0.4 *seconds* to transfer 100 bytes goodput in CON, while this takes 1 *second* using NON. However, as the graph shows, if the goodput is 500 byte, both versions of the protocol uses *1* second to transfer the data. When the payload reaches 600 byte, CON needs to use 1,15 *seconds* to transfer the data, while NON is still able to only use *1* second. After this CON is about 150 *ms* slower than *non* to transfer the same payload, but this gap closes as the payload gets higher. In this case it is quite easy to see a trend – CON is definitely faster at small rates of data, below 500 byte. If the payload is bigger than this, NON will be preferable, only taking the time to transfer a given number of data one way into account.

From another approach, it is possible to look at how many bytes can be sent for every given time using the two different versions of the protocol. This is profitable because this will show which payload size that gives the maximal throughput for every time interval, and can possibly be exploited by both a developer and an end user in such a system. Figure 4.9 shows the direct correlation between the number of bytes sent every second.

Same as in the previous tests it is preferable to have a stable goodput through the network, to be able to calculate and predict the capacity of the network. Figure 4.9 shows the minimum, average and maximum values of goodput using CON. The

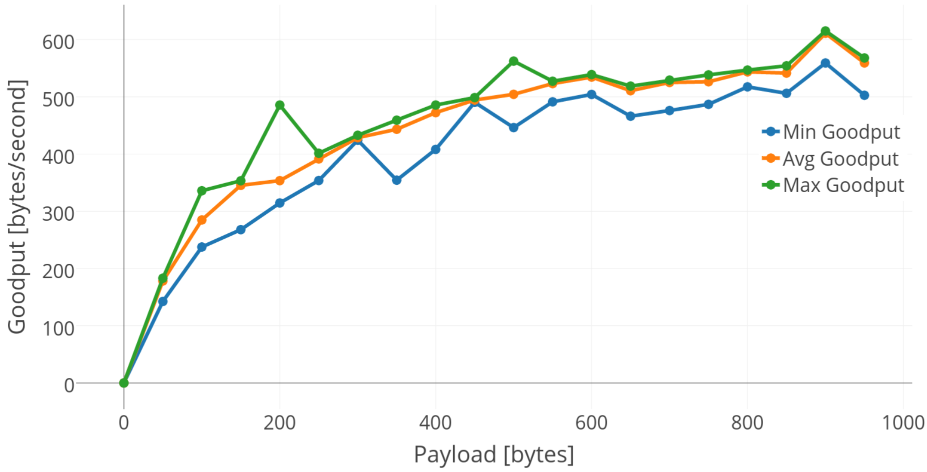


Figure 4.9: Goodput compared to payload CoAP CON

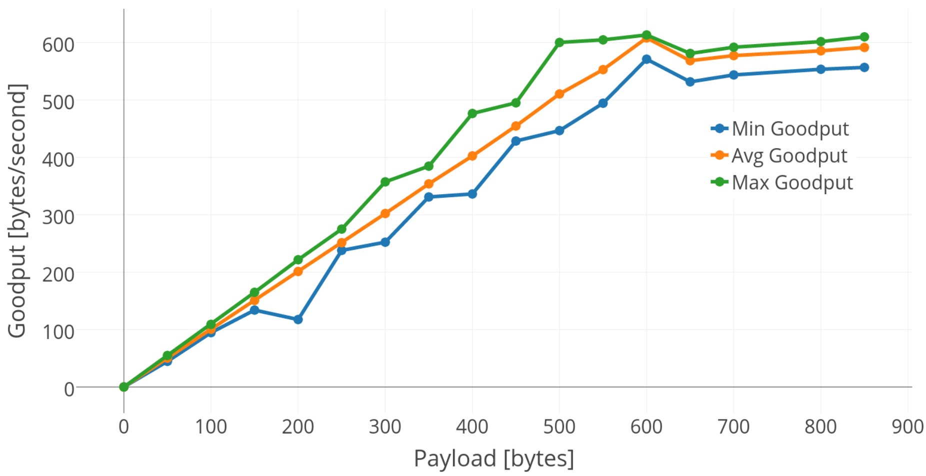


Figure 4.10: Goodput compared to payload CoAP NON

values are in general quite stable, with a few exceptions of the min and max values measured probably caused by disturbance in the network. The graph rises fast at low payload, and later flattens out. The highest achieved goodput in these tests was

611 bytes/second when the payload was 900 bytes. It reached 500 bytes/second at 500 bytes payload. From this graph it looks like the maximum transfer rate using CON is approximately 700 bytes, and will probably never hit 1 kB per second.

In the same case using NON, the graph starts to climb almost linear with a climbing rate of 100 bytes/second added for every 100 byte payload added, as seen in figure 4.10. This makes sense, given that the transfer interval is constant at once per second for payload sizes from 0 to 600 bytes. Variation to min and max values occurs, but not more than expected compared to previous results. The highest achieved goodput is 608 bytes/second, almost exactly the same as the maximum achieved in CON. For payloads higher than this the goodput drops down to about 580 and stays more or less constant there [4].

The direct comparison of the average values of goodput using CON and NON can be seen in figure 4.11. In this case both plots starts at 0 bytes/second, since the payload is 0. After this, the graphs have quite different forms. NON has a very linear rise, all the way up to 600 byte payload. When transferring a payload of 100 bytes, CON is almost three times faster at *305 bytes/second* compared to *108 byte/second* using NON. CON therefore looks much more efficient for a small amount of data, but for payload bigger than 500 byte NON is preferable in this system.

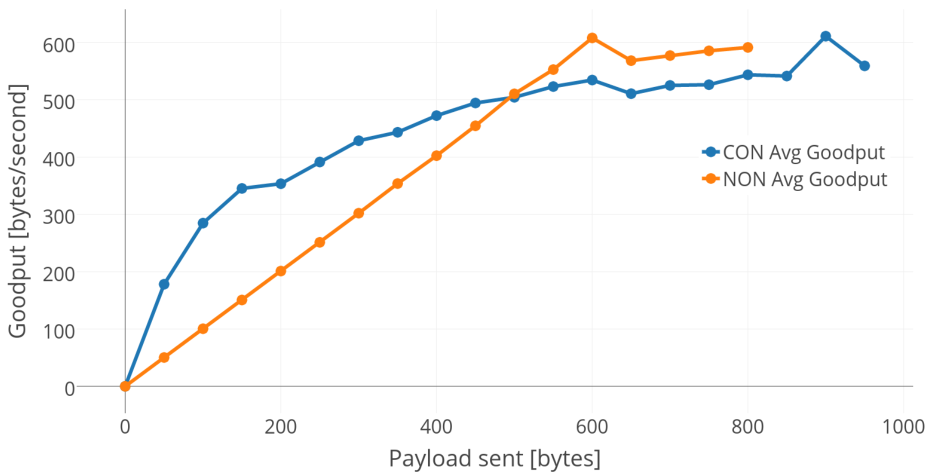


Figure 4.11: Goodput for given payload, CON vs NON

[4]Keep in mind that measurements for payload > 650 bytes in NON are unstable links, and does not rely on the same number of measurements as the rest of the tests

4.4.2 Bytes sent through network, best and worst case

These previous two tests shows the throughput per time, while the tests earlier in the thesis has focused on the amount of throughput that is payload, how much of the data sent is useful data. Since these microcontrollers are used as end nodes in the network, it is natural to assume that they will run on battery power only. In this case it is preferable that they do as little work as possible, meaning also handle as few packets as possible. It will therefore be compared how many packets that needs to go through the end nodes in the different cases, CON and NON [5].

The main argument for trying the NON version of CoAP in this network, was that fewer bytes needed to be sent through the network, meaning less overall usage of the network. This is very relevant in a case where the network should be taken to its maximum capacity with several sensors and microcontrollers.

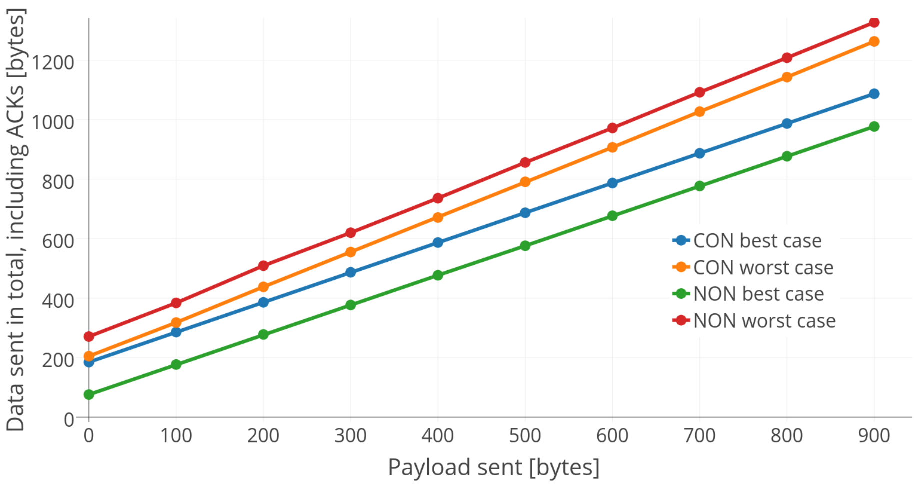


Figure 4.12: Number of bytes per second

But, as seen in figure 4.12, this turns out not to always be the case. The figure shows best and worst case of how many bytes that needs to be sent to transfer a given number of payload. For example, using NON it takes 576 byte to transfer 500 byte payload in best case, but 856 byte in worst case. This is because of the architecture of the two different versions of CoAP. Even though NON does not require ACKs, underlying parts of the bluetooth architecture like ACL adds on some support for this. Also, to prevent a situation where the end node sends data forever without

[5] Before the test, it was clear that fewer packets are needed using NON, since no ACKs are required. The test is therefore done to see *how much* fewer packets are needed here

anyone receiving the data, ACK messages are still being sent regularly, approximately every 15th second in this network. This means, as shown in figure 4.12, that NON *normally* requires a lower amount of bytes to transport a given payload, but in *worst case scenario* it needs even more bytes than CON.

The main question to comparison comes down to how often the worst case occurs in comparison to best case. Sending payload of 500 byte in both cases, NON results in most best case scenarios, while CON results in *only* worst case scenarios. In specific numbers, this gives an average of *598 bytes* sent for NON, and *791 bytes.* This gives us the following equations to calculate how much of the total bytes sent that constitutes of the payload. These calculations are based on measurements similar to the one shown in table 4.6 and 4.5[6].

$$\frac{500\,byte\,payload}{\frac{791\,bytes*15\,packets}{15}} * 100\% \approx 63,21\%\,payload\,(CON) \tag{4.7}$$

$$\frac{500\,byte\,payload}{\frac{(576\,bytes*13\,packets)+(856\,bytes*2\,packets)}{15}} * 100\% \approx 83,56\%\,payload\,(NON) \tag{4.8}$$

From these results it can be concluded that even though NON is considerably worse in worst case than CON, it is still needed about 20 % less packets sent through the network to get the same amount of information through. This result is about the same as what was expected before the test.

4.5 Chapter summary

In this chapter the most central experiments performed in the project have been presented and compared. The first problem up for discussion was if fragmentation of packets is a major issue in an IoT network like the network presented in this thesis. Payloads from 0 to 1000 bytes were sent through the network, and the fragmented packets sent were being captured on the client side using Wireshark. The comparison between CON and NON shows both that there are very small differences between the two versions of the protocol, and that fragmentation of packets is not a major issue in such a network.

The next experiments focused on goodput in the system, how much data can be sent per second. CON is faster at small payloads, mostly because this is requested by GET-requests. At approximately 500 bytes payload CON start to use longer

[6]The complete table containing the measurements can be found on https://www.github.com/sische/MasterThesis/measurements

than 1 second to transfer, and is being bypassed by NON, which can send 600 bytes in one second. After this, both versions start to climb with approximately a rate of 1 second per 500 bytes of payload. Comparisons to the capacity of the different technologies are being discussed, to find out that these results are much lower than expected. Another comparison experiment presented shows that between 500 and 900 bytes payload in CON, while 600 bytes payload is the optimal for NON. This gives a maximal goodput of approximately 600 bytes/second.

Because there are different additional protocols in the two versions of CoAP, the amount of bytes sent in total through the network is not constant, even though the payload is constant. Measurements show that NON require the least packets in best case, but also the most in worst case. Despite this, NON most of the time manages to stay on best case, giving it the best % payload of all packets sent at 500 bytes, with 83,56 % compared to 63,21 % in CON.

Chapter 5

Discussion

5.1 Set up network

O.1: Build a star network of microcontrollers

This objective was fulfilled by using the Raspberry Pi as a central node and nRF52s as end nodes. Central points in the solution was the use of a version of Linux OS with pre-configured kernel of version 4.15 or later on the Raspberry Pi. In addition it was important to understand how prefix in IPv6 works, and how this can be used to identify a device connected using BLE. In this solution the end nodes with nRF52s works as servers, while the central Raspberry Pi works as a client requesting services from these servers.

R.1: Which technologies and transport protocols are suitable in such a network?

BLE was chosen over ANT, as explained in chapter 2.2.3. Bluetooth is a widely used technology that is very interesting in an IoT setting, and was therefore the obvious choice in this network. The BLE version of bluetooth is designed to use a minimal amount of energy and still be reliable and fast, which is central criteria in the testbit. As a result of this 6LoWPAN also seemed like a suitable communication protocol, both because it is made to work together with BLE and because it is an up an coming technology that is assumed to be more and more used in the coming years. *Zigbee* and *Zensys* was the main other options, which did not seem as fit in this case mostly because of different solutions in routing. In the application layer the two main choices was between CoAP and MQTT. Because of time restrictions CoAP was chosen to be studied in depth in this thesis.

5.2 Gather sensor data

O.2: Connect sensors to the end-nodes to collect data

This objective was partially fulfilled. An accelerometer was connected to two of the end nodes in the network, with the goal of gathering vibration data to be sent through the network. Problems occurred when getting the accelerometer to communicate properly with the end node, meaning that it was possible to gather acceleration data, but not as frequently as expected. Getting reliable vibration data was therefore not possible. Due to these problems, in addition to the main scope of this thesis, it was decided to measure the different aspects of the network with simulated data instead of real world vibration data. This would eliminate the possibility of errors due to problems with the sensor, letting this objective to be completed later by future work.

Even though this objective was not fully completed, related coding was done on the nRF52 to be able to initialize and use the accelerometer connected. This code can be useful for later projects in future works, and central aspects from the code will therefore be included in appendix C.

5.3 Send data through network

O.3: Gather information of the data sent through the network

This object was fulfilled by using both Python scripts and Wireshark on the Raspberry Pi to measure the packets sent through the network. Different Python scripts was used to get data from the servers, save data locally after receiving, drawing graphs directly to represent the data, or to forward the data to another device or online storage facility. The most central of these code samples can be seen in appendix A. Wireshark was used to monitor the live capture of packets, and to manually do a detailed analysis of how packets where fragmented differently in the different scenarios.

R.2: What are the main limitations concerning transporting data?

Already in the first tests of RTT shown in chapter 4.1.1, it became clear that one of the major limitations in this network would be the initial transfer speed. This was not expected, and was not included as one of the main objectives in this thesis, which concentrates more on analysing and discussing the data sent through the network, and transport protocols used. It is assumed that this is a problem in a lower level of the protocol stack, and will be left for future work to solve. Other than this, limitations regarding network stability were found. Several of the tested solutions were not able to transfer data at all, or only for a very short period (< 20 seconds). But when a stable solution was found, the link could be open as long as needed, and successful tests have been stable for several days. During these tests it was discovered that the message ID implemented in CoAP uses an 16 bit counter. After 65536 messages have been sent the counter will be reset, and changed to 18 bits. This did not affect the performance in this system, and will most likely not be a problem in other systems either. Other limitations discussed in detail in this thesis was the frequency data can be sent through the network, unstable connections when the payload gets too big, device failure at runtime then the payload gets too big and power consumption increases.

R.3: Are the microcontrollers powerful enough to gather data this frequently?

To gather detailed acceleration data to be analysed in another node of the network, it is assumed that a measuring frequency of 1MHz is needed. The maximum achieved in this system was to call a method from the main loop 11 times every second, and then read a measured values from the accelerometer 150 times for each of these 11 method calls. This adds up to 1650 measuring points every second in a best case scenario. Yes, this is fast enough to gather data, even though problems explained in chapter 3 means this practical experiment will be left for future work in this thesis. The programming code referred to here can be seen in appendix C.

5.4 Analyse data

O.4: Analyse and discuss the gathered information

This object was fulfilled by analysing the data by printing out table and plotting graphs. Using basic tools like this it was possible to document both the differences and similarities in the different protocols tested in this thesis. The results where presented and discussed in detail in chapter 4.

R.4: Could data analysis be done in the end nodes in this network?

Referring to the result from research question R.3. The maximum capacity of the microcontroller is needed to capture acceleration data from the accelerometer, get the desired quality of the data, and forward this to a central node. The end nodes were running on battery power. To do even more calculations in these nodes will not be preferable in this network. This will be possible if the node is one of many nodes in a complete network, where every node gets some time of sleep between every measurement. Even here it is probably not a good idea, concerning the use of batteries. The answer is therefore no, more detailed analysing and representation of the data should be done in a central node with more computational power and better access to power sources.

5.4.1 Measurements

As a summary of the measurements presented in this thesis, the positive and negative aspects of the two versions of CoAP that have been tested will be be directly compared to get a more clear understanding of the differences, advantages and disadvantages.

Table 5.1: Comparison of CON and NON

Case	CON	NON	Comments
Need of ACKs	Yes	**No**	Accational connection test at 16+7+7 bytes in both cases
Minimum number of bytes needed for empty CoAP message	74+104	**76**	
Fastest for payload <500 byte	**X**		Almost 3x faster payload <10 byte
Fastest for payload >500 byte		**X**	On average 0,2 s faster for payload >600 byte
Average time [seconds] to send 200 bytes payload	**0,66**	1,00	
Average time [seconds] to send 700 bytes payload	1,32	**1,21**	
Highest measured goodput [bytes] per second	**608**	**611**	
Overall best stability in the network	**X**		
Calculated lowest power concumption		**X**	
% payload of all packets sent through network (in case of 500 byte sent)	63,21	**83,56**	On average 20% more efficient in the number of packet sent in total

Table 5.1 shows a summary of the main results found in the comparison of CoAP CON and NON in the network presented in this thesis. Results from the table shows that in 7 of the 10 cases presented, NON shows the best results. This should still not be a final result without discussion. As discussed in chapter 4, NON has been considerably more unstable than CON during the test period. It was not possible to get a stable connection when sending more frequently than once every second, and the connection became unstable if the payload was greater than 800 bytes. As a conclusion from this table it is clear to say that NON works best if the conditions are right, sending data every second with a payload between 500 and 700 bytes. In other cases than this, CON works better in the practical experiments presented.

Looking at the network in general using CoAP, the slowest transfer time is 2 seconds to transfer 1 kB of payload. This is considered very slow through a network like this. BLE has a given MTU of 1 MB per second, far beyond what has been achieved in the tests presented here. The highest achieved goodput was about 500 bytes/second, or 1/2 MB/second. BLE is therefore not the main limitation in the network. CoAP creates the messages sent, but should not influence on the network throughput, if not the action of making the packets in the end node is the limitation. 6LoWPAN is only used as a way of transporting data in the network, and will therefore not affect the throughput in the system. Final possibility is that the limitation is in the computational power of one of the devices communicating with each other. In this case it makes sense to assume the nRF52, both since this is the sending part of the network needing to process the data and construct the packets being sent, and because it has significantly less power capacity than the Raspberry Pi.

This brings the question up for discussion if it is profitable to send data larger than 1 kB at all. If the sending rate keeps climbing at the same rate as measured here, with a rate of 1 second slower for every 500 bytes. As a comparison it will take 1000 seconds to transfer 500 kb, which is more than 15 minutes.

5.5 Ease of use

5.5.1 Raspberry Pi

As a central device for the microcontrollers, the Raspberry Pi has worked great. There are several good options when it comes to choosing an OS that can be customised for a specific system, and several good guides that makes this a simple and fast small device for most end users.

5.5.2 nRF52

The nRF52 requires a lot more experience both in programming and general understanding of computers to be used properly than the Raspberry Pi. Online documentation is available and is usually very good, but also some times a bit confusing and messy. A great tool is the Nordic Semiconductor forum specially designed for questions regarding this and other Nordic devices[1]. Example code is also provided by Nordic, to show how the device can be used with different technologies. These works as a starting point, but it is very time consuming and difficult for an end user without specific experience on this device to familiarize themselves with how this code works, in order to change to something else instead. The example code used as an example in this thesis was not optimized for battery consumption, and as a result the small battery drained very fast during the testing. This is of course possible to optimize for an experienced used, but should be taken into account for the average end user of such a device in a complete system.

[1] https://devzone.nordicsemi.com

Chapter 6
Conclusion and Future Work

In this thesis I have used microcontrollers, sensors, single-board computers and a stationary computer to build an IoT network. Using this network I have tested some of the choices a system developer can face when transporting data through the network, particularly emphasising Bluetooth Low Energy and 6LoWPAN. Central topics for discussion has been analysis of data with respect to network capacity, network exploitation, transfer rate and power usage.

From the results presented concerning fragmentation of data, the thesis argues that neither in CoAP CON or NON is fragmentation a major concern. Both the max size of 31 bytes of BLE packets and 270 bytes of 6LoWPAN packets was exceeded in the test, without having major affect on the percentage of payload sent through the network compared to the total throughput. This would otherwise have resulted in a more uneven slope of the graphs presented.

In addition to this, the thesis has presented experiments to analyse and discuss the goodput in the system. Results shows that CoAP CON is faster for smaller payloads, while NON if the payload is bigger than 500 bytes. The highest goodput achieved was 611 bytes/second, and overall the system needs almost 1 additional second for every 500 bytes of payload being added. This is quite slow compared to the limitations of the different technologies used, and it is being discussed what can be the main reason for this. The author did not have the resources to investigate this in detail within the time frame, but it seems reasonable to assume that the limitation is not BLE or 6LoWPAN, but rather limitations in the computational power or wireless antennas of one of the devices used.

The last experiment presented shows that NON require the least packets in best case, but also the most in worst case to transfer the same payload. Despite this, NON most of the time manages to stay on best case, giving it the best % payload of all packets sent at 500 bytes, with 83,56 % compared to 63,21 % in CON.

All results put together shows that both BLE and 6LoWPAN works in such a system, together with both versions of CoAP tested. In some cases the network was stable and reliable, but there where also several limitations which lead to problems during the testing. Limited sending frequency, limited payloads and difficulties when connecting to other devices is the most central. Still I will say the network is a very interesting stating point to a more complex IoT system, that can definitely be used as a starting point for future projects.

6.1 Future work

This thesis does not have any direct previous works, but explains how to build a basic IoT system using devices of different size and limitations. The project has had a limited time frame, and naturally this leads to several possibilities concerning future works, both by adding other objectives and to complete the objective that was not fully completed in this thesis.

A proposed future work is to build the network further both with microcontrollers and computational power, as a direct addition to the system presented. The system can be expanded a lot both concerning bigger and smaller devices, for instance can several sensors be connected to the same microcontroller, and several microcontrollers, both nRF52s and others, can be connected at the same time to get a more real world environment to test data. In the other end the system can be set up both to ask for computational power from a supercomputer or to automatically display the results on a web page on a database. It would then be possible create a more user friendly User Interface (UI), which would make it easier to test and analyse even bigger loads of data sent through the system.

References

[1]

[2] ANT and BLE: whats the main difference?, howpublished = http://electronicdesign.com/mobile/what-s-difference-between-bluetooth-low-energy-and-ant, note = Accessed: 26-05-2016.

[3] Copper (Cu): coap under-agent for firefox. http://people.inf.ethz.ch/ mkovatsc/copper.php. Accessed: 29-05-2016.

[4] MQTT: web page. http://mqtt.org. Accessed: 07-04-2016.

[5] Newark: element 14. https://www.newark.com/. Accessed: 13-05-2016.

[6] Nordic Semiconductor: iot sdk documentation. https://infocenter.nordicsemi.com/index.jsp?topic=

[7] Nordic Semiconductor: nrf52 series soc. https://www.nordicsemi.com/Products/nRF52-Series-SoC. Accessed: 25-03-2016.

[8] Nordic Semiconductor: s110 softdevice. https://www.nordicsemi.com/eng/Products/S110-SoftDevice-v7.0. Accessed: 26-05-2016.

[9] Ubuntu Mate: installation for raspberry pi, howpublished = https://ubuntu-mate.org/raspberry-pi/, note = Accessed: 31-01-2016.

[10] Ashton, K. (2009). That 'internet of things' thing. *RFiD Journal 22*(7), 97–114.

[11] Chown, T. and S. Venaas (2011). Rogue ipv6 router advertisement problem statement.

[12] Devices, A. (2009). Digital accelerometer adxl345. *Analog Devices 21*.

[13] Gomez, C., J. Oller, and J. Paradells (2012). Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology. *Sensors 12*(9), 11734–11753.

[14] Hui, J. W. and D. E. Culler (2008). Extending ip to low-power, wireless personal area networks. *Internet Computing, IEEE 12*(4), 37–45.

[15] Hunkeler, U., H. L. Truong, and A. Stanford-Clark (2008). Mqtt-s—a publish/subscribe protocol for wireless sensor networks. In *Communication systems software and middleware and workshops, 2008. comsware 2008. 3rd international conference on*, pp. 791–798. IEEE.

[16] Kovatsch, M. (2011). Demo abstract: human-coap interaction with copper. In *Distributed Computing in Sensor Systems and Workshops (DCOSS), 2011 International Conference on*, pp. 1–2. IEEE.

[17] Kushalnagar, N., G. Montenegro, D. E. Culler, and J. W. Hui (2007). Transmission of ipv6 packets over ieee 802.15. 4 networks.

[18] Lamping, U. and E. Warnicke (2004). Wireshark user's guide. *Interface 4*(6).

[19] Montenegro, G., N. Kushalnagar, J. Hui, and D. Culler (2007). Transmission of ipv6 packets over ieee 802.15. 4 networks. *Internet proposed standard RFC 4944*.

[20] Mulligan, G. (2007). The 6lowpan architecture. In *Proceedings of the 4th workshop on Embedded networked sensors*, pp. 78–82. ACM.

[21] Semiconductors, P. (2000). The i2c-bus specification. *Philips Semiconductors 9397*(750), 00954.

[22] Shelby, Z., K. Hartke, and C. Bormann (2014). The constrained application protocol (coap).

Chapter A
Appendix A

Appendix A contains samples of programming code used to gather and transfer data in the IoT system described in this thesis.

This first example is the most simple, using *GET* commands to get the measured values from CoAP CON. All the python scripts uses example code from Nordic Semiconductor in

```
import asyncio
from aiocoap import *

SERVER_ADDR = '2001::2AF:B7FF:FEB6:1494'
SERVER_PORT = '5683'
SERVER_URI = 'coap://[' + SERVER_ADDR + ']:' + SERVER_PORT

@asyncio.coroutine
def main():
    protocol = yield from Context.create_client_context()
    sequence_number = 1
    number_of_measurements = 200
    while sequence_number < number_of_measurements:
        request_acceleration = Message(code=GET)
        request_acceleration.set_request_uri(SERVER_URI + '/lights/led3')
        response = yield from protocol.request(request_acceleration).response
        print('Acceleration'+str(sequence_number)+': %s Response Code:
            %s\n'%(response.payload, response.code))
        sequence_number += 1

if __name__ == "__main__":
    asyncio.get_event_loop().run_until_complete(main())
```

This script was written to get observable values stored in a local file.

```
import asyncio
from aiocoap import *

#SERVER_ADDR = '2001::211:64ff:fea5:8542'
SERVER_ADDR = '2001::2e6:6aff:fe64:54dd'
#SERVER_ADDR = '2001::2af:b7ff:feb6:1494'
SERVER_PORT = '5683'
SERVER_URI = 'coap://[' + SERVER_ADDR + ']:' + SERVER_PORT

responseList = []
def observe_handle(response):
   f = open('/home/sindre/Desktop/desktopAccelValues', 'a')
   if response.code.is_successful():
      responseList = bytes.decode(response.payload)
      for i in range(0, (len(responseList))):
         f.write((str(responseList[i]) + ' '))
      f.write(responseList)
      f.write('\n')
      print("Written to file!")
   else:
      print('Error code %s' % response.code)
   f.close()
@asyncio.coroutine
def main():
    protocol = yield from Context.create_client_context()
    request = Message(code=GET)
    request.set_request_uri(SERVER_URI + '/lights/led3')
    request.opt.observe = 0
    observation_is_over = asyncio.Future()
    try:
        requester = protocol.request(request)
        requester.observation.register_callback(observe_handle)
        response = yield from requester.response
        exit_reason = yield from observation_is_over
        print('Observation is over: %r' % exit_reason)
    finally:
        if not requester.response.done():
            requester.response.cancel()
        if not requester.observation.cancelled:
            requester.observation.cancel()

if __name__ == "__main__":
    asyncio.get_event_loop().run_until_complete(main())
```

This example is to get observable measurements directly displayed in a graph:

```
import asyncio
from aiocoap import *
import matplotlib.pyplot as plt

#SERVER_ADDR = '2001::211:64ff:fea5:8542'
SERVER_ADDR = '2001::2e6:6aff:fe64:54dd'
#SERVER_ADDR = '2001::2af:b7ff:feb6:1494'

SERVER_PORT = '5683'
SERVER_URI = 'coap://[' + SERVER_ADDR + ']:' + SERVER_PORT

responseList = []
drawValuesList = []

def observe_handle(response):
   if response.code.is_successful():
      responseList = bytes.decode(response.payload)
      print(responseList)

      for i in range (0,len(responseList)):
         drawValuesListappend(int(responseList[i]))
      plt.plot(drawValuesList)
      plt.xlabel('Measurement number')
      plt.ylabel('Acceleration values')
      plt.show()

   else:
      print('Error code %s' % response.code)
@asyncio.coroutine

def main():
    protocol = yield from Context.create_client_context()
    request = Message(code=GET)
    request.set_request_uri(SERVER_URI + '/lights/led3')
    request.opt.observe = 0
    observation_is_over = asyncio.Future()
    try:
        requester = protocol.request(request)
        requester.observation.register_callback(observe_handle)
        response = yield from requester.response
        exit_reason = yield from observation_is_over
        print('Observation is over: %r' % exit_reason)
    finally:
```

```
        if not requester.response.done():
            requester.response.cancel()
        if not requester.observation.cancelled:
            requester.observation.cancel()

if __name__ == "__main__":
    asyncio.get_event_loop().run_until_complete(main())
```

Chapter B

Appendix B

Appendix B contains screenshots and detailed figures from the measurements done in the network built in this thesis. This is meant to be a supplement to the figures presented earlier in the thesis to give the reader a deeper understanding of the system. In addition to the measurements presented here, all data gathered from the system and code used will be public GitHub, http://github.com/sische/MasterThesis.

The following two tables present values discussed in chapter 4. These are minimum, average and maximum values of goodput and time between each CoAP packet, using both CON and NON. These values have been found by monitoring more than 100 packets in each case.

Table B.1: Goodput and time, CoAP CON

Payload	Min goodput	Avg Goodput	Max Goodput	Min Time	Avg Time	Max Time
0	0	0	0	0.27	0.28	0.35
50	142.41	178.12	183.09	0.27	0.28	0.35
100	237.56	284.8	335.87	0.3	0.42	0.35
150	267.9	345.33	353.34	0.41	0.44	0.56
200	314.52	353.53	485.6	0.41	0.57	0.64
250	353.74	391.43	401.35	0.62	0.64	0.71
300	424.29	428.58	432.81	0.69	0.7	0.71
350	354.35	443.3	459.17	0.76	0.79	0.99
400	408.16	472.34	485.63	0.82	0.85	0.98
450	490.37	494.4	498.64	0.9	0.91	0.92
500	446.23	504.35	562.4	0.89	0.99	1.12
550	491.24	523.1	527.21	1.04	1.05	1.12
600	504.16	534.45	538.85	1.11	1.12	1.19
650	466.14	510.67	518.67	1.25	1.27	1.39
700	476.04	525.06	529.18	1.32	1.33	1.47
750	486.82	526.31	538.33	1.39	1.43	1.54
800	517.42	543.6	546.74	1.46	1.47	1.54
850	506.12	541.44	554.19	1.53	1.57	1.68
900	559.14	611	615.06	1.46	1.47	1.61
950	502.66	559.18	567.89	1.67	1.7	1.89

Table B.2: Goodput and time, CoAP NON

Payload	Min goodput	Avg Goodput	Max Goodput	Min Time	Avg Time	Max Time
0	0	0	0	0.9	1	1.12
50	44.65	50.27	54.89	0.91	1	1.12
100	94.6	100.49	109.03	0.92	1	1.06
150	133.93	150.66	164.85	0.91	1	1.12
200	117.46	201.11	221.58	0.9	1	1.13
250	237.88	251.28	275	0.91	1	1.05
300	252.18	302.09	357.26	0.84	1	1.19
350	330.96	353.79	384.61	0.91	0.99	1.06
400	336.15	402.46	476.34	0.84	1	1.19
450	428.41	454.6	494.93	0.91	0.99	1.05
500	446.52	510.38	600.09	0.83	0.98	1.12
550	494.21	552.72	604.61	0.91	1	1.11
600	570.93	607.94	613.1	0.98	0.99	1.05
650	531.48	568.29	580.91	1.12	1.15	1.22
700	543.52	577.18	591.76	1.18	1.21	1.28
800	553.34	585.45	601.51	1.33	1.37	1.45
850	556.62	591.37	609.83	1.39	1.44	1.53

The following capture shows a part of a Wireshark capture, as described in chapter 4.1. This capture shows a payload of 0 byte sent using CoAP NON, and was the base for values used in table 4.1.

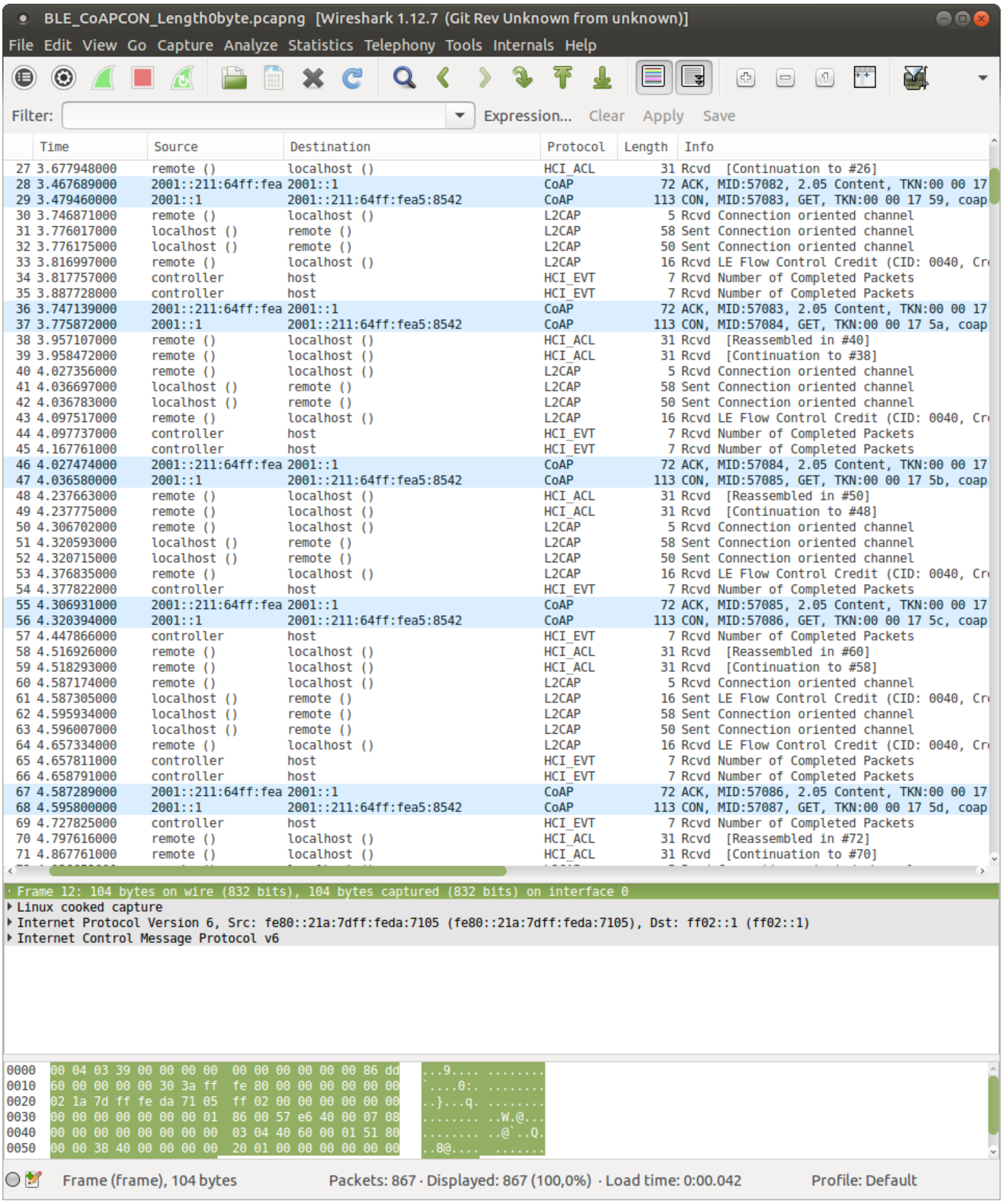


Figure B.1: Wireshark capture, 0 bytes CON

Chapter C
Appendix C

Appendix C contains samples of programming code written to read acceleration data from the Adafruit ADXL345 accelerometer connected to the nRF52 using the I2C interface. This code was not being used in the testing of this thesis, as explained in chapter 3. The code has been included and explained so it can be used by others in later projects.

The following code sample in C programming is parts of the main function in the file *main.c*. From here methods *accelerometer_init* and *start_measuring* are being called to intitialize the different registers of the accelerometer, and start the measuring from the main loop.

```
int main(void){
  uint32_t err_code;

  app_trace_init();
  leds_init();
  timers_init();
  accelerometer_init();

  ...

  for (;;)
  {
    power_manage();
    start_measuring();
  }
}
```

accelerometer_init will initialize the different registers to be able to read from the accelerometer. These registers should first be defined in a header file along with information about the slave address and which nRF52 pins that represented SCL and SDA, to clarify the code.

```
// Part of header file:

   #define ADXL345_SLAVE_ADDRESS    0x53

    #define TWI_SCL_M               27   //!< Master SCL pin
    #define TWI_SDA_M               26   //!< Master SDA pin

   #define X_AXIS_OFFSET            0x1E
   #define Y_AXIS_OFFSET            0x1F
   #define Z_AXIS_OFFSET            0x20
   #define DATA_RATE_AND_POWER_INIT  0x2C
   #define POWER_SAVING_INIT        0x2D
   #define INTERRUPT_ENABLE_CONTROL  0x2E
   #define DATA_FORMAT_CONTROL        0x31
   #define READ_X_AXIS              0x32
   #define READ_Y_AXIS              0x34
   #define READ_Z_AXIS              0x36
```

```
// Initialize accelerometer in main.c file:

static void accelerometer_init()
{
   write_reg(DATA_FORMAT_CONTROL, 0x00, 2);
   write_reg(POWER_SAVING_INIT, 0xFF, 2);
   write_reg(INTERRUPT_ENABLE_CONTROL, 0xFF, 2);
   write_reg(X_AXIS_OFFSET, 0xFF, 2);
   write_reg(Y_AXIS_OFFSET, 0xFF, 2);
   write_reg(Z_AXIS_OFFSET, 0xFF, 2);
}
```

The initialization of the accelerometer calls the function *write_reg*, which is used to write to a register.

```
static uint32_t write_reg(uint8_t register_address, uint8_t data_to_write,
    uint8_t size)
{
   ret_code_t ret;
```

```
    uint8_t addr8 = (uint8_t)register_address;
    ret = nrf_drv_twi_tx(&m_twi_master, ADXL345_SLAVE_ADDRESS, &addr8, 1,
        true);
    if(NRF_SUCCESS != ret)
    {
       break;
    }
    ret = nrf_drv_twi_tx(&m_twi_master, ADXL345_SLAVE_ADDRESS,
        &data_to_write, size, false);
    return ret;
}
```

After this initialisation process is successful, the *start_measuring* can begin.

```
static void start_measuring()
{
   char stringa[150];
   char anotherString[150];

   for (int j = 0; j < 150; j++)
   {
      int r = read_reg(READ_Z_AXIS, 0x00);
      int t = numberOfMeasurements++;
   }

   sprintf(stringa, "%d,", numberOfMeasurements);

   for  (in i=0; i < 200; i++)
   {
      if (stringa[0] == '\0')
      {
         measuringCounter = i;
         break;
      }
      else
      {
         if (!stringToSendOccupied)
         {
            appendChar(stringToSend, 150, stringa[i]);
         }
      }
   }
   numberOfMeasurements = 0;
```

```
}
```

This method calls the function *read_reg*, to read the registers that have been set to update earlier.

```
static uint16_t read_reg(uint8_t register_address, uint8_t data_returnValue)
{
   uint16_t rd;
   ret_code_t ret;
   uint8_t buff[2];
    uint8_t addr8 = (uint8_t)register_address;

    ret = nrf_drv_twi_tx(&m_twi_master, ADXL345_SLAVE_ADDRESS, &addr8, 1,
         true);
    if(NRF_SUCCESS != ret)
    {
        break;
    }
    ret = nrf_drv_twi_rx(&m_twi_master, ADXL345_SLAVE_ADDRESS, buff, 2,
         false);
   rd = (uint16_t)(buff[0] | (buff[1] << 8));

    return rd;
}
```

After this it is possible to get the acceleration value from another point in the code, to be stores in a char array **str* until it is being sent.

```
static void acceleration_value_get(coap_content_type_t content_type, char **
     str)
{
   stringToSendOccupied = true;

   strcpy(newString, stringToSend);
   *str = newString;
   stringToSend[0] = '\0';

   stringToSendOccupied = false;
}
```