NTNU – Trondheim
Norwegian University of
Science and Technology

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

Sindre Schei

Submission date: May 2016
Responsible professor: Frank Alexander Kraemer, ITEM
Supervisor: David Palma, ITEM

Norwegian University of Science and Technology
Department of Telematics

Abstract

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 transfering 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 usedin such a setting. Data can be generated by using and comparing different sensors connected to end nodes in the network.

This thesis will therefore aim to test the use of microcontrollers as end nodes in an IoT network. The main focus is to establish a connection between a Raspberry Pi as a central point in a 6LoWPAN network, and connect nRF52 devices from Nordic Semiconductor to this. Using different versions of CoAP to transfer data over a Bluetooth Low energy (BLE) connection, 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 through the network. Optimizing packet sizes, fragmentation and maximising throughput at the same time as minimizing power usage are other key words.

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.

Results from this work includes graphs and discussions in which case the two main versions of CoAP is preferred, and putting them up against each other in the form of tables and graphs from tests done on the IoT system. These show that CoAP NON is grenerally preferable if the data sent is larger than 500 bytes. It is also more stable than CoAP CON in tests performed on this system. However CoAP CON can transfer packets

at a higher rate for small packets (< 500 byte), even though each packet needs to receive an ACK, which will use some of the network connection capacity in situations where this is very limited.

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 avhandlingen 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 during the period. I would like to point out 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 both of 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.

BLE Bluetooth Low Energy.

CoAP Constrained Application Protocol.

CON Confirmable CoAP message.

CPU Central Processing Unit.

DNS Domaine 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.

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.

radvd Router Advertisement Daemon.

RAM Random Access Memory.

RTT Round Trip Time.

SPI Serial Peripheral Interface.

TCP Transmission Control Protocol.

TDMA Time Division Multiple Access.

TFTP Trivial File Transfer Protocol.

UDP User Datagram Protocol.

USB Universal Serial Bus.

Glossary

byte	Used as a synonym for *octet*, meaning 8 bits put together as one 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".
payload	The part of the transmitted data that is the intended message. Also denoted *goodput* in this thesis.
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.

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 Internet of Things (IoT) was first used by Kevin Ashton in 1999 [5], when describing a global network of objects. He later explained how he thinks 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 can be said with a high certainty that this will be a central part of the coming years of the Internet.

In order for this development from a network mostly based on human made material to a network based more on data from sensors using M2M communication, several factors have 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 impracticable to set up a power cable to all of these. For 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 glsiot system will therefore be how to transport data as efficient as possible. Raw data from sensors are seldom useful to an end user, therefore the data also needs to be analysed and often represented in another form before it can be useful to the user. A device in the network therefore

needs to analyse the data, find out what is important or unimportant, find patterns, draw graphs or figures and forward the results a monitor, a web page or to be stored on a server to be used later. Should this process of analysing take place in the end nodes, or is it more preferable to forward raw data to a central component of the network? Or maybe in some cases this is not even enough, but the central computer needs even more computational power from a supercomputer, before data can be displayed?

In order for this to works in a proper way, a number of things are needed. Protocols, batteries, so on so on.

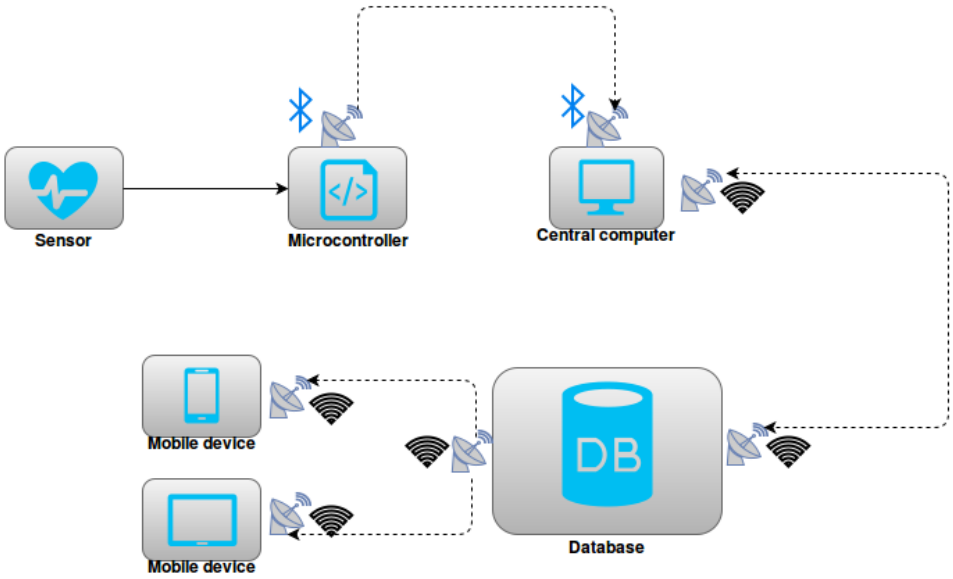

Figure 1.1: Example of IoT architecture

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 prosecutor in developing technologies to be used is Nordic Semiconductor. As described on their web page:

> *The future of electronics is wireless and portable due to an almost insatiable consumer demand for ever greater levels of freedom and flexibility. Nordic Semiconductor is playing a key role in the realization of that future, by providing ultra low power (ULP) wireless chips that can run for a long time from small power sources, like watch batteries [1].*

This

1.2 Scope and Objectives

1.2.1 Scope

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

- Gathering data from sensor efficiently, both concerning time and power consumption
- Transporting data efficiently, concerning power consumption and optimal throughput, both concerning time spent and amount of useful data that gets through
- Where in the network it is preferable to analyse the raw data, concerning energy consumption and time spent in total

To achieve this, some explanation of background protocols, devices used and network topology will be touched as well, in addition to low level details needed to set up the system architecture 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 in 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 to the end node to be sure that the data read 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 advanates 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 draw conclusions of 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 protocols tested.

R.2: What are the main limitations when it comes to 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 bottle neck 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 a higher and higher rate to see if it is possible to reach an acceptable sampling rate.

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

This is dependent on the result from R.3. This might be possible if it feels like 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 was 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 has been 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 measurements done on the network by 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 CoAP Confirmable CoAP message (CON) and Non-Confirmable CoAP message (NON) has its 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 doing in further work. At the end the chapter contains an overall evaluation of the devices and technologies used, and how the experience gained in this project can be used in the future.

Chapter 6 summarizes the entire work, done in this project and presents the final conclusion.

Chapter 2
Background

This thesis describes the setup and usage of an End-to-End IoT system. In order for this to be set up by others later to perform reproducible tests, a detailed description of components, sensors and protocols used is needed. This chapter will go through the background information of the devices, technologies and protocols used, and why these where chosen over other alternatives.

2.1 Bluetooth Low Energy

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 already is a well established technology in cell phones, laptops and several other devices. This means that few changes needs to be made to these devices to be able to work with bluetooth smart. Still to this date, a device that only implements BLE is not able to communicate with a device that only implements classic Bluetooth [7]. The 6LoWPAN Working Group has recognized the importance if BLE in IoT [8] .

Denne linken snakker om 6lowpan, flytte til etter jeg har presentert 6LoWPAN?

The protocol stack of BLE has two main parts, the controller and the host, as shown in 2.1. In the system presented in this thesis this represents the Raspberry Pi as the controller (master) and Nordic nRF52 as the host (slave). The communication between these is done through the standard Host Controller Interface (HCI). All slaves are in sleep mode by default, and are woken up by the master when communication is needed. Links are being identified by a randomly generated 32-bit code, and the Industrial Scientific Medical (ISM) band used is 2,4 GHz [7]. Other protocols used 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

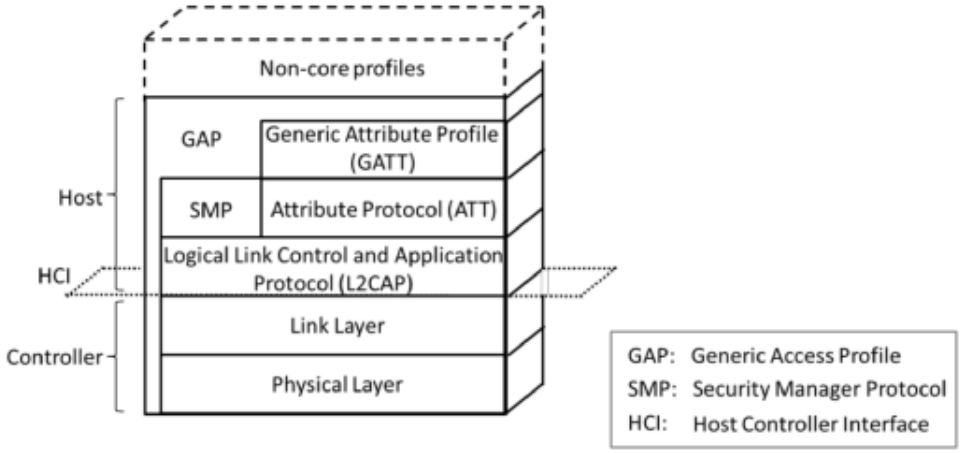

Figure 2.1: BLE protocol stack

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.

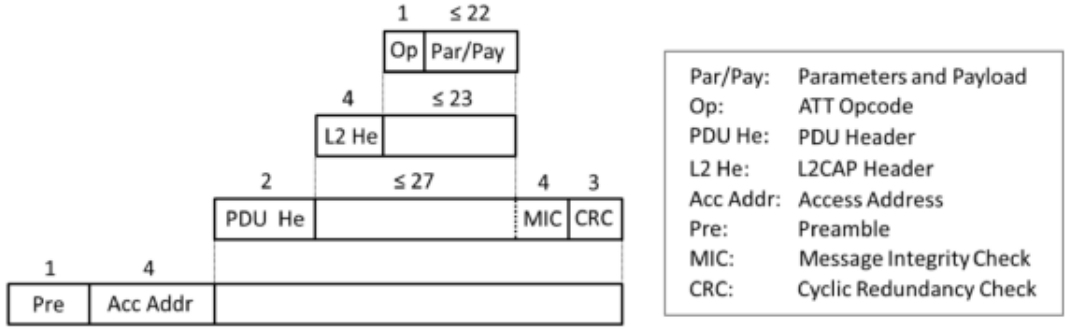

Figure 2.2: BLE Data Unit Structure

Figure 2.2 shows the data unit structure in BLE, meaning the different fields that can be used in a packet. The header fields of 4 byte of access addresses and L2CAP will be central topics of discussion later in this thesis. The figure shows that the maximal packet size in a network like this is 31 bytes per packet. In the case of

the network presented here, when the BLE slave has been connected to a master, it stops searching for other masters, and it is not possible to connect to several masters. This means that we are only able to create a *star network*, not a *mesh network*. This could possibly be an idea for improvement later. Other than this, BLE seems like a very good alternative in this project.

2.2 6LoWPAN

IPv6 over Low Power Wireless Personal Area Networks (6LoWPAN) is a defined protocol for using Internet Protocol version 6 (IPv6) in low energy networks, to identify sensors and devices, as defined in the IEEE 802.15.4.

To use the Internet Protocol in low energy networks in addition to standard networks was proposed by Geoff Mulligan and the 6LoWPAN Working Group. In [11], the advantage of 6LoWPAN is explained as

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 [11].

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 an impressive 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 [8]. The maximum packet size of the physical layer is set to be 127 bytes, far below the limit of 1280 byte [10]. It is expected that other layers will produce packets of the desired size to fit the system. In the system presented in this thesis by the example code on the nRF52 is the side set to 270 bytes for every packet, as will be shown later in the thesis.

2.3 Raspberry Pi

Developed by Newark Element 14 [3], the Raspberry Pi has become a central tool for many people wanting to get started using small computers. The device has been known as a single-board computer the size of a credit card specially designed for small network projects and to be used as an educational tool used all the way

from elementary schools to here at Norwegian University of Science and Technology (NTNU). This was therefore a natural device to use as a starting point in this system as well.

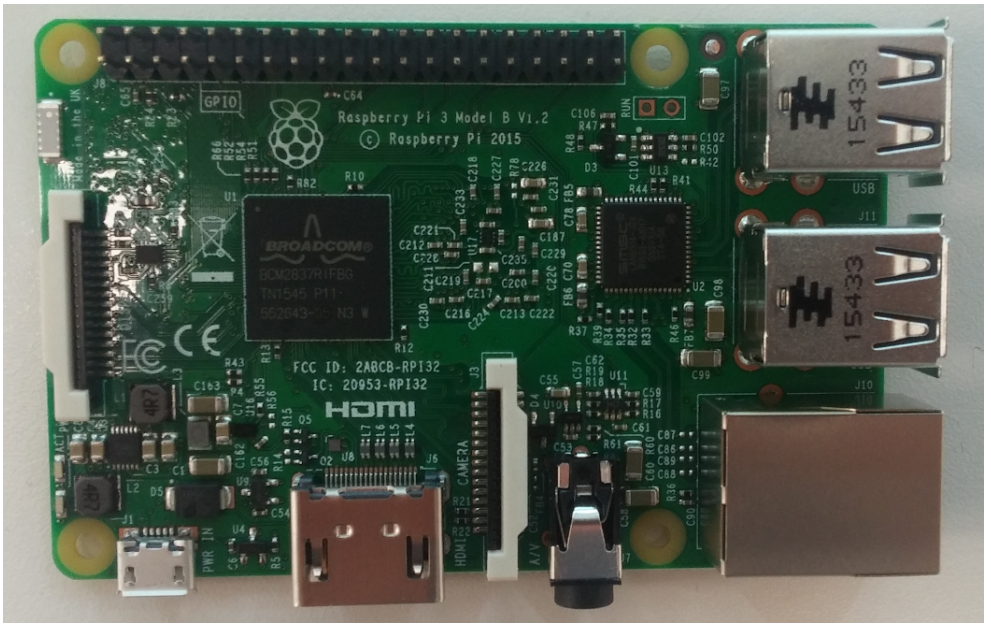

Figure 2.3: Raspberry Pi 3

The Raspberry Pi is a single-board computer on 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 is included, and it was quite easy to set up, given that the right Unix kernel has been used in the Operating System (OS) of the Pi. Along with the Raspberry Pi, we needed a good and stable operating system with a kernel that supported the 6LoWPAN architecture. For this, Ubuntu Mate version 15.10 with kernel version 4.15 was chosen, and used on the Raspberry Pi. As other versions of Ubuntu this is Unix based, and has a complete Graphical User Interface (GUI) of a full OS. The set up of this will be explained in chapter 3.1.

2.4 nRF52

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

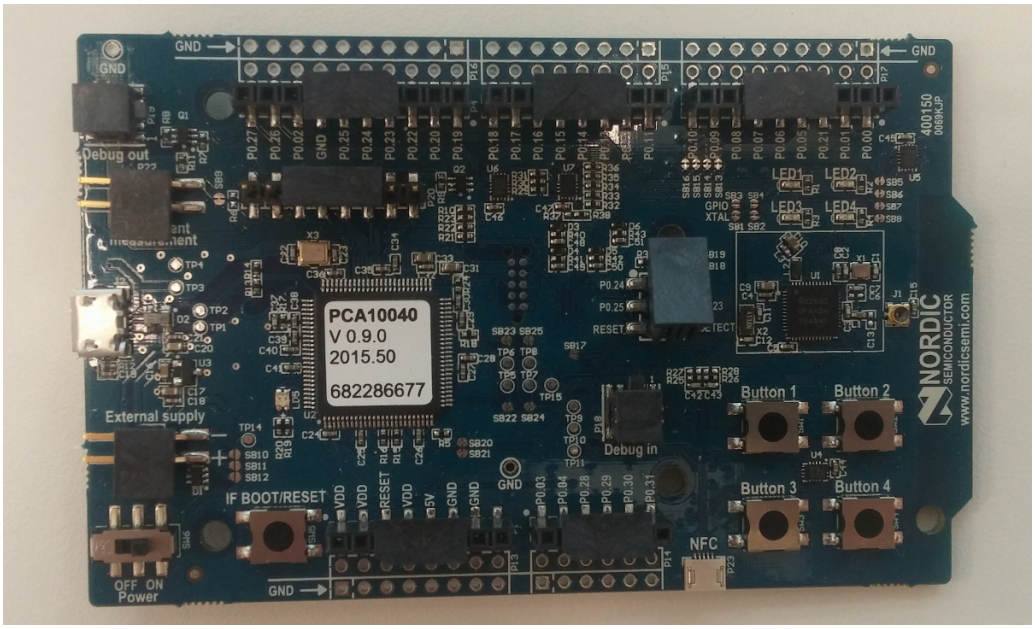

Figure 2.4: Nordic Semiconductor nRF52

[4]

This device has been advertised as a powerful multiprotocol single chip solution, with both a 32-bit ARM processor, a 512kB flash and 64kB og flash memory. The building blocks of the chip in its entirety is shown in 2.5. The key features mentioned by Nordic themselves are:

- Multi-protocol 2.4GHz radio
- 32-bit ARM Cortex M4F processor
- 512kB flash + 64kB RAM
- Software stacks available as downloads
- Application development independent from protocol stack

- On-air compatible with nRF51, nRF24AP and nRF24L Series
- Programmable output power from +4dBm to -20dBm
- RSSI
- RAM mapped FIFOs using EasyDMA
- Dynamic on air payload length up to 256 Bytes
- Flexible and configurable 32 pin GPIO
- Programmable Peripheral Interface – PPI
- Simple ON/OFF global power modes
- Full set of digital interfaces including: SPI/2-wire/UART/PDM/
- I2S, all with EasyDMA
- 12-bit/200KSPS ADC
- 128-bit AES ECB/CCM/AAR co-processor
- Quadrature demodulator
- Low cost external crystal 32MHz ± 40ppm for Bluetooth, ± 50ppm for ANT
- Low power 32MHz crystal and RC oscillators
- Ultra low-power 32kHz crystal and RC oscillators
- Wide supply voltage range (1.7 V to 3.6 V)
- On-chip DC/DC buck converter
- Individual power management for all peripherals
- Package options: 48-pin 6x6 QFN/WL-CSP

In other words much more different options than will be used in this system. The important part here is the processing power, the PPI, the Inter-Integrated Circuit (I2C) bus and the bluetooth antenna. That these things combined can run on only a 3V CR2032 battery is unbelivable.

Figure 2.5 shows how the different parts are placed on the chip.

In order to run compiled code on the nRF52, a preprogrammed *Soft device* has to be loaded onto the chip. There are several versions that can be downloaded from http://www.nordicsemi.com.

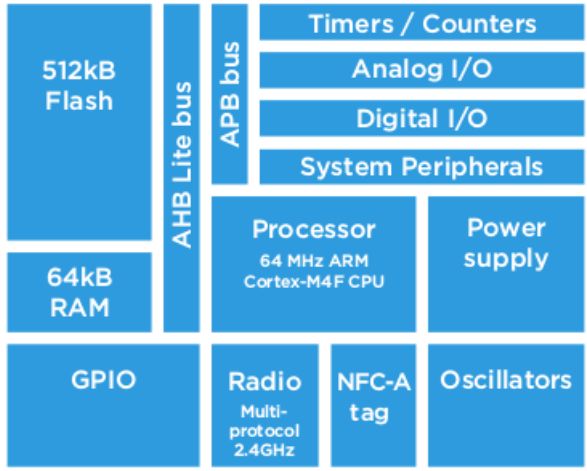

Figure 2.5: Nordic Semiconductor nRF52 chip in detail

2.5 Adafruit ADXL345 Accelerometer

In order to do collect data, a sensor needed to be connected to the Nordic Semiconductor nRF52. The main thought behind the thesis was to measure vibrations, and a good accelerometer was needed. The Adafruit ADXL345 accelerometer was chosen for several reasons.

- It is the same accelerometer built in on the Zolertia Z1 microcontroller
- It can measure acceleration in all three axes, X, Y and Z.
- It sends digital data right away. This means there 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 Serial Peripheral Interface (SPI), which makes it easy to connect to the nRF52.

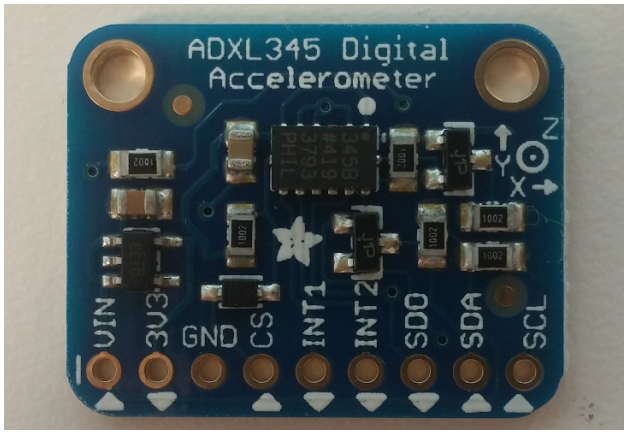

Figure 2.6: ADXL345 Accelerometer

2.6 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 sending rate is thought to be one of the main limitations, either in the form of limits of data at once or number ofsendings pr second. The protocol needs to be stable and energy efficient and work with both BLE and 6LoWPAN. Luckily, Nordic Semiconductor provides example code and examples on how to get started with this.

2.6.1 CoAP

CoAP is a transport protocol designed to be used in constrained networks for M2M communication. It is User Datagram Protocol (UDP) based, and works good in low-power and lossy networks. It can be used with microcontrollers, and with IPv6 and 6LoWPAN. Both GET and PUSH functionality can be used, as well as *observable* GET. This means that a server can "subscribe" to end notes in the network, and get updates either after a given timespan or when changes have been made. This therefore looked like a promising protocol to use, and was chosen as the main transport protocol to test in the network [12].

The main technical features described in CoAP is as follows:

- Web protocol fulfilling M2M requirements in constrained environments
- UDP binding with optional reliability supporting unicast and multicast requests.

- Asynchronous message exchanges.
- Low header overhead and parsing complexity.
- URI and Content-type support.
- Simple proxy and caching capabilities.
- A stateless HTTP mapping, allowing proxies to be built providing access to CoAP resources via HTTP in a uniform way or for HTTP simple interfaces to be realized alternatively over CoAP.
- Security binding to Datagram Transport Layer Security (DTLS).

CoAP has several similarities with working over Hyper Text Transport Protocol (HTTP), with 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 needs 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, being described as follows:

make figure of CoAP Layering

Confirmable *Some messages require an acknowledgement. These messages are called "Confirmable". When no packets are lost, each Confirmable message elicits exactly one return message of type Acknowledgement or type Reset.*

Non-confirmable Message Some other messages do not require an acknowledgement. This is particularly true for messages that are repeated regularly for application requirements, such as repeated readings from a sensor.

Acknowledgement Message An Acknowledgement message acknowledges that a specific Confirmable message arrived. By itself, an Acknowledgement message does not indicate success or failure of any request encapsulated in the Confirmable message, but the Acknowledgement message may also carry a Piggybacked Response (see below).

Other messages used is *GET, PUT, POST and DELETE*, to get or change data stored on the server.

Figure 2.9 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 2.10 shows the same for NON, where no Acknowledgements (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

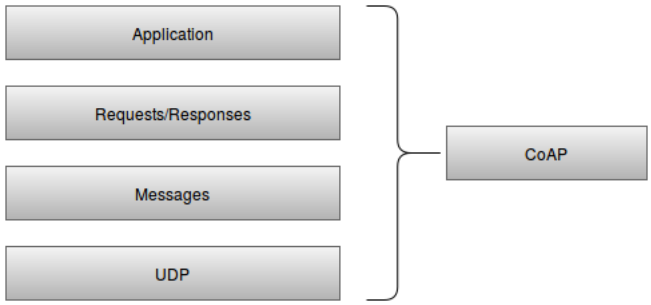

Figure 2.7: Sequence diagram CoAP

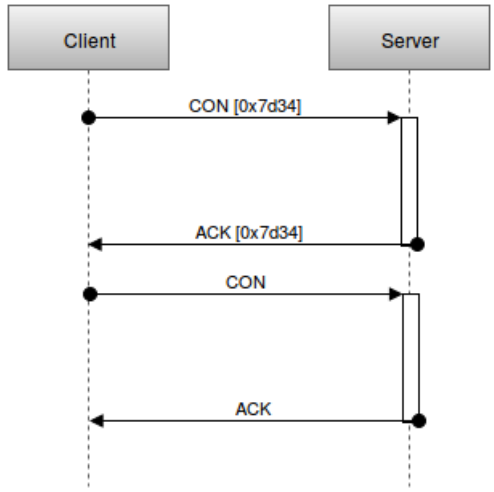

Figure 2.8: CON CoAP client server sequence diagram

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. This works in systems where every message does not need to get to its destination, for instance in a sensor based network like the network presented in this thesis. A message ID is still provided to every message to remove duplicated messages, but dropped messages are lost data. If the packages sent contained data that could not be dropped, for instance containing crucial patient information from censors on a patients body, this is not a good solution. In that case the reliable solution *must* be used.

Show examples of setting up coap NON needs acks first? Wireshark without bluetooth-sniffing?

Figure 2.11 summarizes the different message types used in CoAP. The default

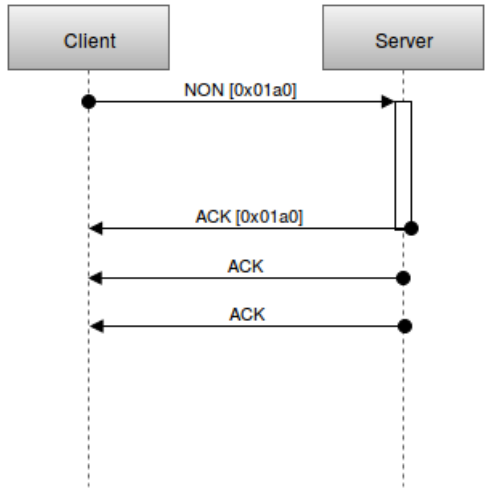

Figure 2.9: NON CoAP client server sequence diagram

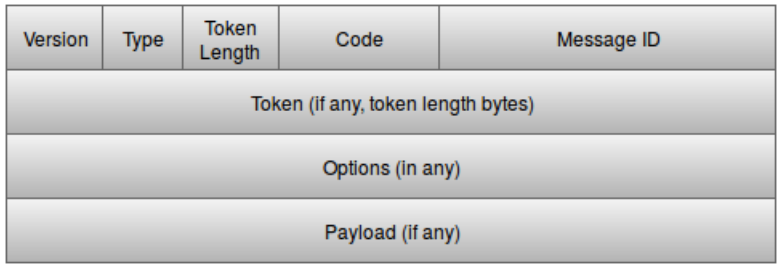

Figure 2.10: CoAP message format

setting for an empty message is that it is a *reset* message or an ACK. The only exception is the use of *ping*, to test the Round Trip Time (RTT) in the network. In this case the empty message will work as a CON.

2.6.2 MQTT

Another transport protocol that could be used in such a network of microcontrollers is MQTT. This is known as a publish-subscribe based on Transmission Control Protocol (TCP), using a MQTT broker. NTNU does have a broker that is possible to use, or a broker can be rented from several other places. Here a

[9]

	CON	NON	ACK	RST
Request	X	X	-	-
Response	X	X	X	-
Empty	*	-	X	X

Figure 2.11: CoAP usage of message types

2.7 Software tools

As Integrated Development Environment (IDE), the *KEIL Vision* was used, as recommended by Nordic Semiconductor (where?), 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.

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) [6]. . This 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.

image?

Other software used to set up, monitor and test this network include Mozilla Firefox with Copper to get basic communication with nRF52 from the Pi without the use of a script or terminal. Wireshark has been used to capture packets sent through the network, and GitHub has been used both as a backup solution. To make the most basic figures in this thesis the web based tool *draw.io* was used, unless other is specified. *polt.ly* was used to draw the graphs used. All the images of devices in the network has been taken by the author.

Chapter 3
System Architecture

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

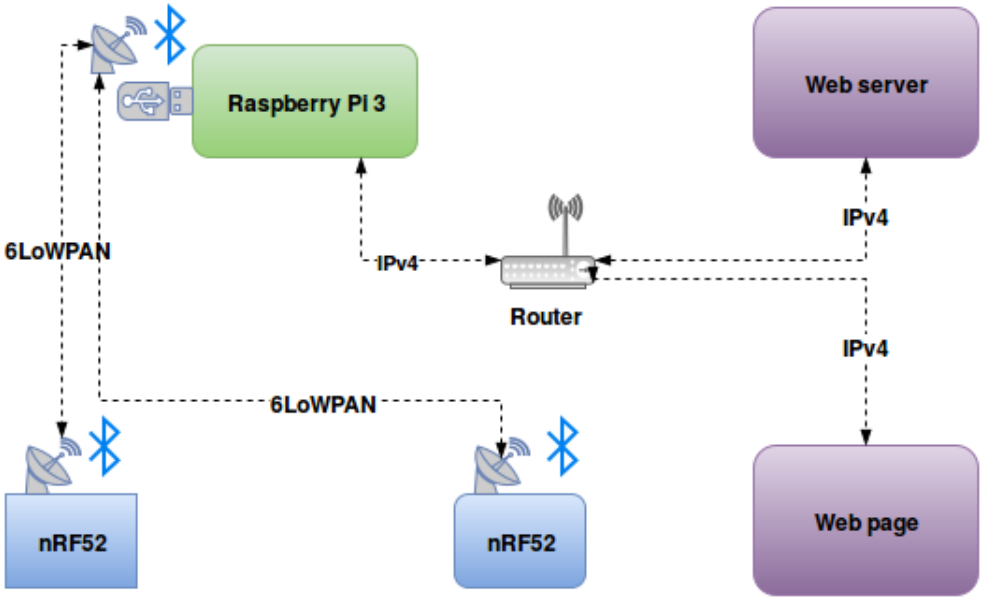

Figure 3.1: System architecture

Figure 3.1 shows how the complete end-to-end system of this thesis is set up. In short terms, the *Nordic Semiconductor nRF52* is connected to a *Raspberry Pi 3* using 6LoWPAN and BLE. This means that they are able to communicate using

IPv6 addresses, even though the nRF52 only has a bluetooth antenna built in, as shown in figure 2.5. The limitations of BLE means that the nRF52 can only connect to *one* device at the time. Several of these put together are therefore forming a *star network* using the *Raspberry Pi* as a central point of connection. Computations can now either be done in the end nodes at the *nRF52s*, at the *Raspberry Pi*, or forwarded to a web server or another computer with more computational power. Data can also be displayed directly to a web page from the *Raspberry Pi*.

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 pros and cons of doing computations in end nodes, compared to transferring information to a server with *much* more 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, from the smallest to the biggest component, to explain their computational and power capabilities, and how they can communicate efficiently.

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.

3.1 Connecting Raspberry Pi and nRF52

To set up the communication between a Raspberry Pi and the nRF52, the two code examples TWI and Observable server from Nordic Semiconductor was used as a starting point for coding on the nRF52. It was however quite tricky to set connect these two together the first time. The following recipe is what worked best.

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 (Note: Jan. 2016), but *Ubuntu Mate* is stable in version 4.15. *Ubuntu Mate* was therefore chosen as the best and most stable OS. When this is done a resizing of the file system is needed to use all the capacity of the memory card. This is not necessary, but recommended to be able to use more than 4GB of the memory card. To resize, run the following commands

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

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

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

To use BLE, install Bluez and radvd using *apt-get* in a *Unix terminal*:

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

To activate IPv6 forwarding, uncomment the following line (remove "#") in the file *etc/sysctl.conf*

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

Add the *bt0 interface* in *radvd.conf*:

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

Write in the bt0 interface

```
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*.

```
bluetooth_6lowpan
6lowpan
radvd
```

It is now possible to use the *hcitool* command.

```
hcitool lescan
```

This command will scan for BLE devices nearby, and find the bluetooth address, for instance *0211:22FF:FE33:4455*. The normal procedure in this case would be to run the following command:

```
echo 1 > /sys/kernel/debug/bluetooth/6lowpan_enable
hcitool lecc 0211:22FF:FE33:4455
service radvd restart
```

This command 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, and the following commands worked fine:

```
echo 1 > /sys/kernel/debug/bluetooth/6lowpan_enable
echo "connect 0211:22FF:FE33:4455 1" > /sys/kernel/
debug/bluetooth/6lowpan_control
service radvd restart
```

FIX verbatim new line?

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

```
ping6 2001::0211:22FF:FE33:4455
```

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.

3.2 Connecting nRF52 and ADXL345

Can possibly write much more here, code samples in appendix, things that needed to be changed and so on

The ADXL345 Accelerometer used was connected using I2C, which is quite straight forward using the nRF52. Connection scheme is as follows (nRF52 -> ADXL345):

* 5V – VIN *Power source, **green cable** in Figure 3.2*
* GND – GND *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*

After this is done it is possible to write to and read from the registers of the accelerometer over the two SDA and SCL cables.

3.2.1 Connection challenges

Following instructions from a representative from Nordic Semiconductor, the bt0 interface was set up using the recommended IPv6 prefix: *2001:bt8::1/64* in in the file *etc/sysctrl.conf*, and after that trying to connect and test the connected devices using addresses on the form:

```
ping6 2001:bt8::0211:22FF:FE33:4455
```

This turned out not to work on the IPv6 network used on NTNU, where the standard prefix is *2001::1/64*, without the *db8*. Both solutions should however work in other IPv6 networks.

Descibe bt8-problem and dev-zone.nordicsemi.c and infocen-ter.nordicsemi.co

3.3 Nordic Semiconductor example code

Describe why I dropped the accelerometer

Nordic Semiconductor has provided several examples along with the nRF52-DK documentation, that can be used as a starting point when programming applications to the device. These files are written in the programming language *C*.

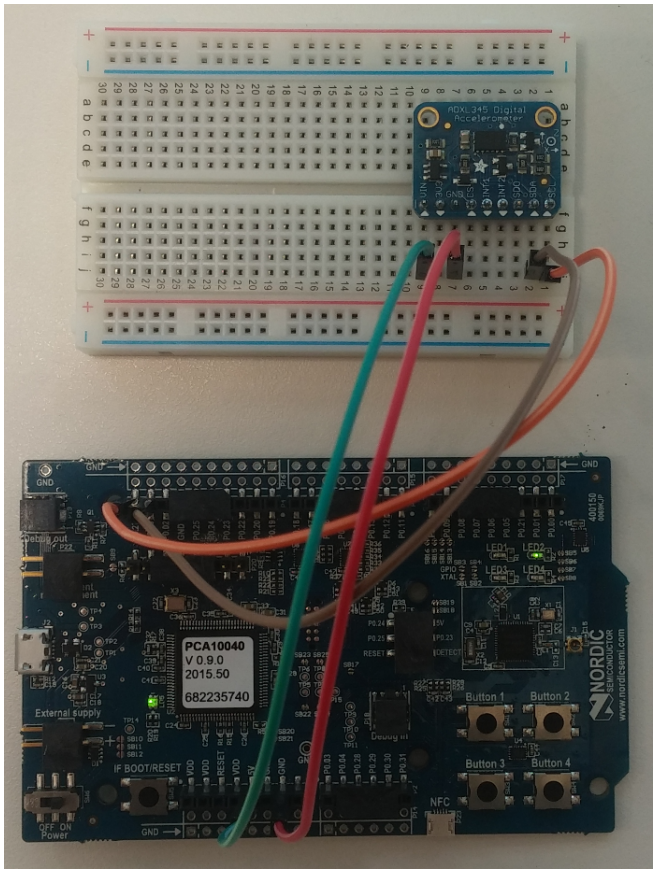

Figure 3.2: Connected nRF52 – ADXL345

3.3.1 CoAP

As a first step, the nRF52 needed to communicate with the Raspberry Pi. As explained in section 2.6, a good starting point for this is to use the CoAP trasnport protocol, and the example of the use of this on the device. The *CoAP Server* and *CoAP Observer* was loaded into two different nRF52 devices, and both of them connected to the Raspberry Pi. Now radvd had to be activated on the Pi, to be able to use this as a *forwarding server*.

Remove "Figure 1" from image

Figure 3.4 shows the initial CoAP example, by using two nRF52 devices, BLE over 6LoWPAN and a BLE enabled router (Raspberry Pi) on both ends in this case. As the figure 3.4 shows, the code example for the CoAP client includes two different

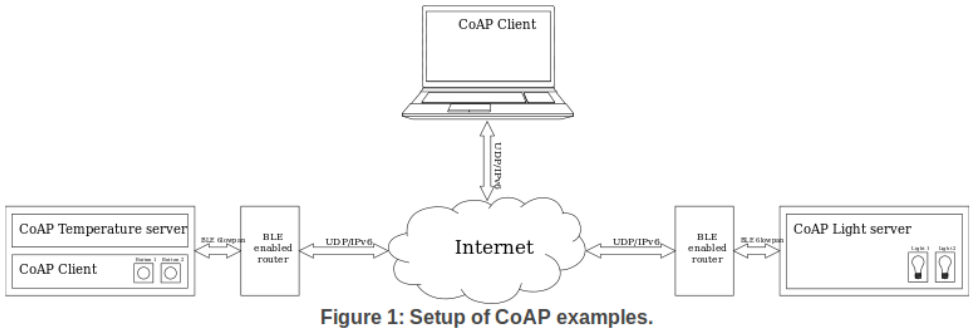

Figure 3.3: CoAP Client-Server communication example

cases:

* In the fist case the CoAP client acts as a client requesting services from the CoAP server. Button 1 and 2 will control light 1 and 2 on the nRF52 server, which will send a conformation back if the light was changed or if something went wrong.

* In the second case the CoAP server is excluded, but the CoAP client works as a server that can handle requests from the router. In this case it is possible to use *Copper* in a *Firefox browser* on the Raspberry Pi to act as a client to request values from a simulated temperature sensor on the server. The server will then send the current simulated temperature back, or tell if something went wrong.

Figure 3.4 shows the capture of packets using Wireshark.

This gives us the sequence diagram shown in Figure 3.5.

Create new sequence diagram

This works fine, and by writing a Python script running on the *Raspberry Pi* it is possible to request data by *GET messages* as soon as the *nRF52* is ready. However, as shown in figure 4.1 and figure 4.2, the network transfer rate can vary a lot. The limitations in BLE and 6LoWPAN makes this much slower than a standard IPv6 connection, and the *ping* RTT varies from 100 ms all the way up to over 600 ms. As a comparison, to *ping google.com* from the same Raspberry Pi using IPv6 takes

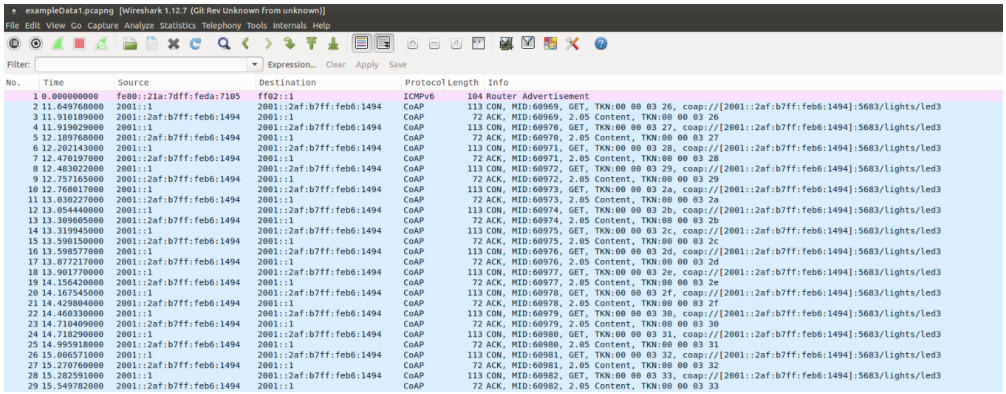

Figure 3.4: CoAP Client-Server example Wireshark capture

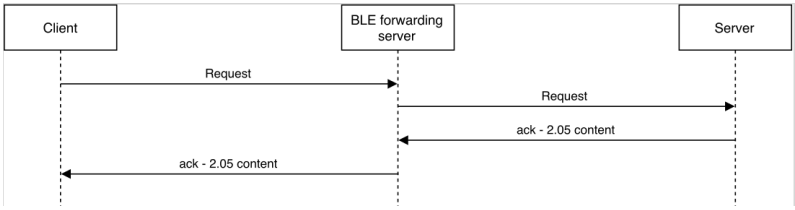

Figure 3.5: Sequence diagram CoAP Client-Server example

on average 15 ms. A comparison of this is shown in figure 4.1. This is therefore a clear limitation of sending rate in the system. Because of this, the maximum transfer rate achieved in the system using the standard CoAP server example is one message every 200 ms.

Using the standard CoAP, described as *CoAP CON* in this thesis, messages are sent with a "TCP mindset". This means that every message sent by the client must receive an ACK. This is very stable, and ensures that every message gets to its destination, but it requires a lot of unnecessary computational power in the leaf node in systems where the importance of each message is not that high. The system is meant to be used with tools for analysing, and it is not essential that data needs to get to its destination right away. In these cases there is another alternative, known in this thesis as *CoAP NON*. This is more like a "*glsudp mindset*", where each packet does *not* need an ACK. This halves the number of packages sent, saves a lot of the network as well as actions needed to be done by the nRF52 where power consumption is a huge issue. Therefore the *CoAP Observable* function looked very interesting, and will therefore be directly compared to *standard glscoap* later in this thesis.

3.3.2 Observable CoAP

In the given Nordic example of a CoAP observable server, the problem of unnecessary packets being sent can be resolved. Using this it is easy to set up a server that can be observed by either a nRF52 client or a BLE forwarding server. The observable end point sends a *CON-ACK request* in a given time interval to assure that the observing client is still there. As long as this message is being responded the server will continue to send data without any requirements of an ACK.

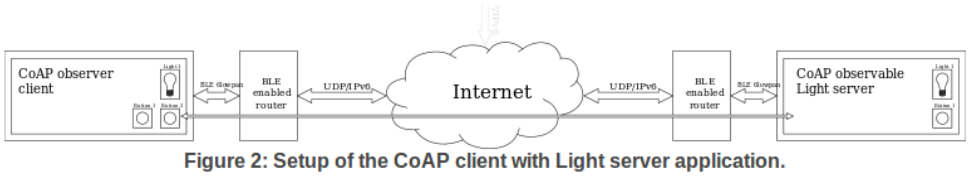

Figure 3.6: CoAP Observable Server example

Remove "Figure 1" from image

Create new sequence diagram for Observable as well

3.4 Getting acceleration values

As soon as the sending of values is being handled in a good way, the next step is to read acceleration values from the ADXL345 accelerometer, which is connected to the NRF52 using the I2C interface.

Acceleration values can only be read from the ADXL345 if this has been correctly initialized at compile time. In order to do this, code to write to and read from the registers had to be added. The initializing process is described in the accelerometer data sheet [2]. 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 before the acceleration value from the different axes can be read.

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 then stores in a simple dynamical char array before being sent and reset after one second. This turned out to be very time consuming and hard to solve in a good 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. Neither the nRF52 or the ADXL345 turns out to be fast enough to do this in the implemented solution. To not 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.

Since the network connection between the nRF52 and Raspberry Pi was already stable, it was easy to generate random values of fixed length to send on the nRF52, and do measurements to calculate the optimal throughput between these two. The next chapter will therefore describe the data analysis of the data sent through this network in detail, and how to optimize the percentage of usable data being transported.

Describe why not to gather acceleration values in this project

3.5 From Raspberry Pi to Network Computer or Server

When running the Unix based OS Ubuntu Mate, the raspberry Pi can be used more or less as a regular computer. This has a pre-installed version of the most basic programs needed, for instance *Mozilla Firefox Browser*, *Pluma text editor* and *Unix terminal*. This means that the user has several options on how to process data on this device:

* 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 analyse of the data. The results can then be posted directly to a server or displayed on a web page.

Which of these four options that is most relevant depends on the data, and how 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 great 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 about

1000 acceleration values every second, which represents a measuring rate of 1 kHz. 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 conducted on data sending rate from the nRF52 to the Raspberry Pi in the form of and graphs, tables and figures to try to determine the most efficient combination of amount of data to send, sending frequency as well as protocols to use.

4.1 Initial limitations in the Network

4.1.1 Stable transfer rate

As soon as the end nodes of the network can communicate with the Raspberry Pi, the next step is 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. The results is measured in ms used for every RTT.

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

When tested, the connection turns out to be very slow and in some cases very unstable, especially at maximum transfer rate. In the test of ping shown in figure 4.1 the connection between the Raspberry Pi and the nRF52 had 14 % packet loss, 1 out of 7 packets. Other test showed similar results, but the CoAP CON has in general got a higher stability at maximum sending rate than CoAP NON. 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. .

Make figure

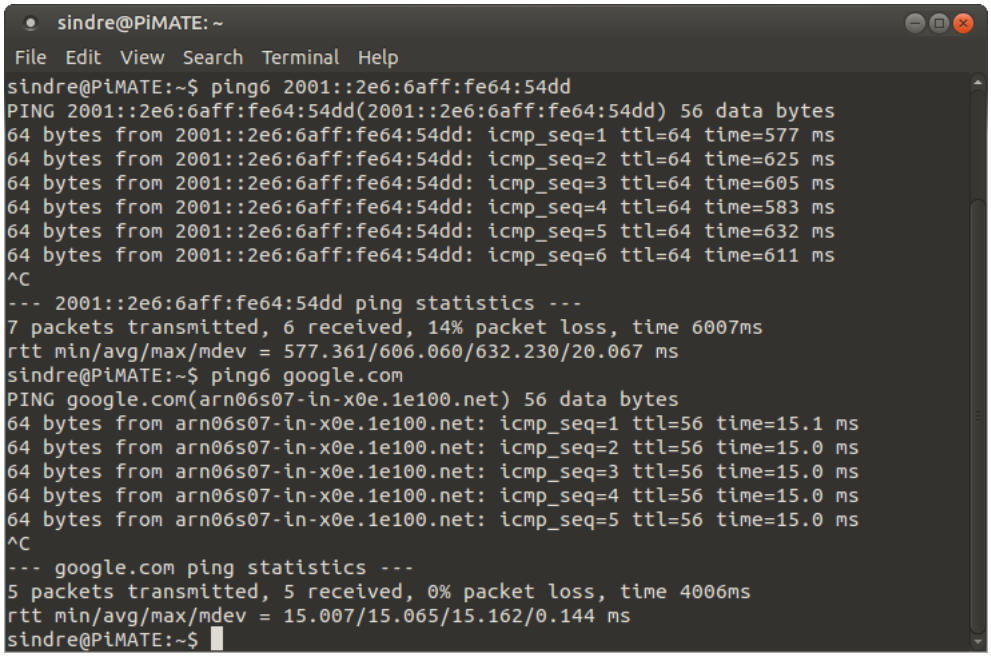

Figure 4.1: Comparison: ping nRF52 and ping google.com

$$\overline{x}_n = \frac{\sum_{i=1}^{n} x_i}{n} \tag{4.1}$$

Equation 4.1 shows the standard way used to calculate an average of the measured values. Here a is any real number and n is any integer.

Skriv om problemer med bt8 i Architecture?

After a test period, the problems explained in was fixed. A more reliable test could then be performed, as shown in figure 4.2. Using equation 4.1 to calculate average gives an approximate average of 250 ms RTT. This is very slow, and way beneath the transfer limitations of both BLE, 6LoWPAN and CoAP, as explained in chapter 2 . 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 huge limitation in the network.

Skriv spesifikt om begrensninger for disse tre i background

The conclusion from these initial tests is therefore that CoAP CON is stable at at lower transfer interval than CoAP NON, and can in best case cope with sending every 250th ms, which is very good compared to the high RTT. NON has been so unstable at initial tests that it was decided to only send every second. This makes

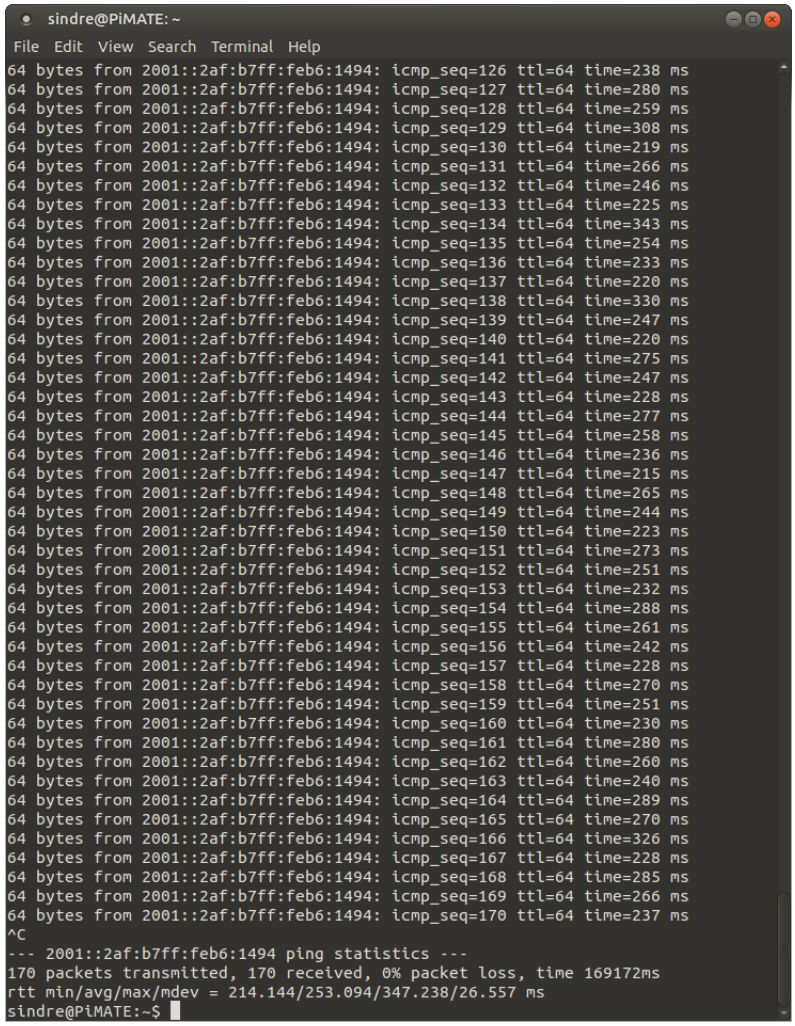

Figure 4.2: Ping nRF52, different time

quite a huge difference when testing the different protocols, as *CoAP CON* can send a lot more *goodput* per second, but *CoAP NON* requires less power to send each message, and a much higher percentage of the packets sent through the network contains useful data.

4.1.2 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 network described in this thesis). Each of these packets needs header fields of a certain size, or other requirements. In a network of microcontrollers used in this thesis fragmentation can be a factor that needs to be taken into account to optimize the goodput of the system. This will be a central part of the testing, and will therefore be explained in detail in the following section.

To better understand fragmentation, imagine a train with carriages as shown in Figure 4.3. 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.

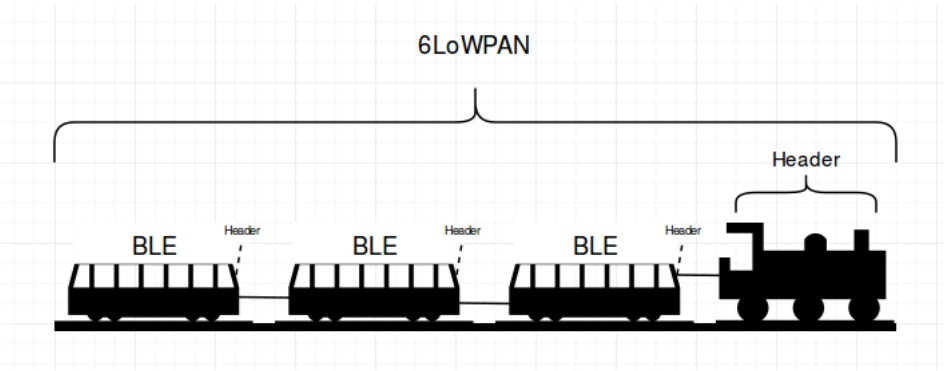

Figure 4.3: Packet fragmentation - train comparison

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 maximal number of bytes compared to the number of packets sent. 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 *goodput* vs *throughput* in the network.

4.2 Description of measurements

The following sections will show experiments performed to to test if *fragmentation* is a huge issiue 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 *googput* as possible. This will be done by sending data of constant length, capture packets using *Wireshark*, and then increasing the packet size to see the changes.

Before these experiements started, the expected results was 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 how big the packages needed to be before it would be considered *profitable* to send.

When sending BLE packets over the network, observations from the system shows that the maximum packet size over BLE is 31 bytes. 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 byte is needed, meaning three BLE packets. The ratio between *useful* and *needed* (known as *goodput* and *throughput*) data transferred therefore start out very poorly if the payload sent is very small.The best possible percentage of useful data we can hope to acheive will also be limited by this, 27 bytes goodput and 4 bytes header field, shown in the following calculation.

$$\delta = \frac{x_2}{x_1} * 100\% \tag{4.2}$$

Equation 4.2 shows the standard way used to calculate the percentage, δ, that the natural number x_2 is of the total amount x_1.

$$\frac{27b}{31b} * 100\% = 87,094776\% \approx 87,1\% \tag{4.3}$$

In the actual system the number will quite certain be a lot lower than this. Both because header files in 6LoWPAN packages needs to use some of the capacity, and because of other limitations.

Like what?

4.3 CoAP

As explained in Chapter 4.1.1, CoAP can be split into two different main sections. CON messages can be sent quite frequently, but every message needs to get an ACK before the next message can be sent. This means that it is quite fast, but a lot of

packets needs to be transported through to get usable data at the other end. The other alternative is NON, where each message does *not* need an ACK.

Table 4.1: Wireshark CoAP CON 0 bytes

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

Table 4.1 shows the most basic example of a capture of packets in Wireshark [1]. The full capture can be seen in the Appendix A. In this case an empty char array was sent, meaning an goodput equal to 0 byte. All of these bytes are therefore sent to route packages through the network. The maximum of 31 bytes per BLE packet have been exceeded twice, meaning that three packets needed to be sent. The first packet is labeled *[Reassembled in #40]*, the second *[Continuation to #38]* and the last *Connection oriented channel*. After this, the ACK packages follows, two packages of 58 and 50 bytes, respectively. The last bytes received tells how many packages was 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.

Note that packet number 46 and 47 in table 4.1 has got a timestamp that indicates that they originally was sent from the end node in the network as packet number 41 and 42, but used longer time than the other packets sent through the network at the same time. The main reason for this is of course that they are bigger than the rest of the packets, but since the network limit should be much higher than this it can maybe be seen as a weakness that this gives a clear impact even at these small values.

[1] When measuring packet sizes, all measurements had the same result when the size of data was constant. The tables therefore takes only one of the measurements at a time in account. All the measured data can be found on https://www.github.com/sische/MasterThesis

Table 4.2: Wireshark CoAP CON 100 bytes

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 goodput is being sent through. The same basic packages needed are still there, but in addition the 100 bytes of data is added. This means adding more BLE packets, but also that the percentage of useful data sent through is a lot higher, approximatley 55 % in this case. By doing several experiments like this, it was possible to create the graph in Figure 4.4. This shows the correlation between goodput and throughput compared to the number of packets sent, measured every 10th byte from 0 byte to 200 bytes large packets.

In this particular case shown in 4.4 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 %, as shown in equation 4.1. 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 has been transferred. The graph will therefore converge to 87,1 %, just as the values climbing in figure 4.4, even before packet size of 200 bytes.

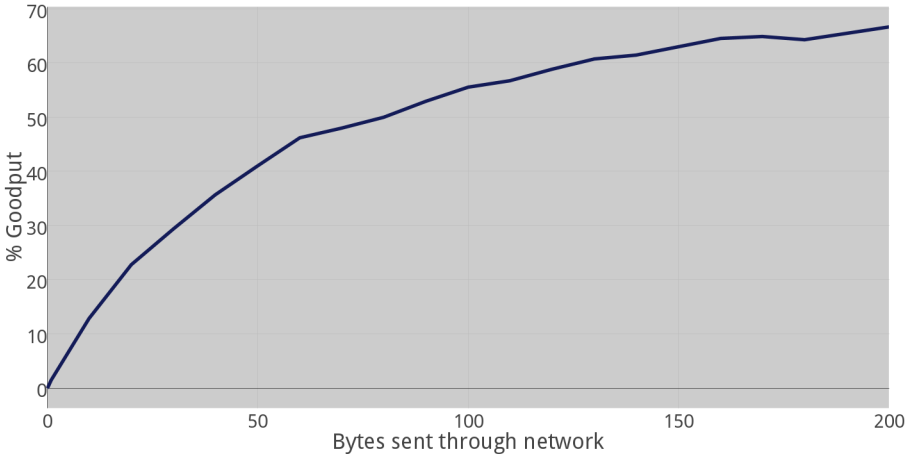

Figure 4.4: CoAP CON, 0-200 bytes sent

4.3.1 Discussion

Even though this first test was done with a very limited amount of data transferred, it is easy to see that the curve clearly flattens out and forms the shape of a *parabola* with a vertical *directrix* at $x = 0$. The next step would 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 either BLE or 6LoWPAN. This test will therefore be explained in the next section.

Since there is already a clear trend in the form of the graph in figure 4.4 even before the transfer rate reaches 200 bytes, it makes sense to also test the other version included in CoAP, CON. If it is possible to find a way of transportation that could give a higher mathematical maximum, and also get to this limit faster, it could be very profitable to the system. It also makes sense to test CoAP NON because it doesn't need to send an ACK for every packet, as CON does. Maybe this means that the header files can be split up in another way, and that the useful amount of data sent through therefore will be higher.

To see what happened when the limit of a 6LoWPAN packet was breached in the system, tests was being set up to send larger amounts of data than 200 bytes. This would also be a good way to test if the percentage of goodput will converge to 87,1 %, only considering BLE packets. 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. As before, the messages are being sent as soon as possible, meaning approximately four messages per second on average using CoAP CON. In addition will NON be

tested in detail.

4.3.2 CoAP NON comparison

A CoAP NON request does not require a response in form of an ACK. This means that the 108 bytes sent to and handled by the end node can be skipped, which means less computational power in the end node and network capacity to the node is needed. This solution makes sense to use in networks where the demand for all data is low, 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 unstable results in tests at the beginning of the set up. The transfer rate is therefore set to one per second for this protocol.

4 LEDs image?

Table 4.3: Wireshark CoAP NON 0 bytes

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

A basic example of the CoAP NON connection is shown in table 4.3. The entire Wireshark capture can be seen in Appendix A. This is directly 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 and one 6LoWPAN packet. This is less than half of what was needed using CoAP CON, where the 108 ACK packages was needed in addition. The packets are still recognizable the same way as before, the first packet is labeled *[Reassembled in #40]*, the second *[Continuation to #38]* and the last *Connection oriented channel*.

Fewer packages sent means less energy used, less network capacity needed and less computational power in the end node. Hopefully this will lead to a higher percentage of goodput compared to throughput as well. it would therefore be interesting to compare the rest of the measured values.

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 over 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

Table 4.4: Wireshark CoAP NON 100 bytes

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

sense. The NON packet size is here 177 bytes, compared to 173 bytes in CON, which represents one more header field of 4 bytes needed in NON. 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.

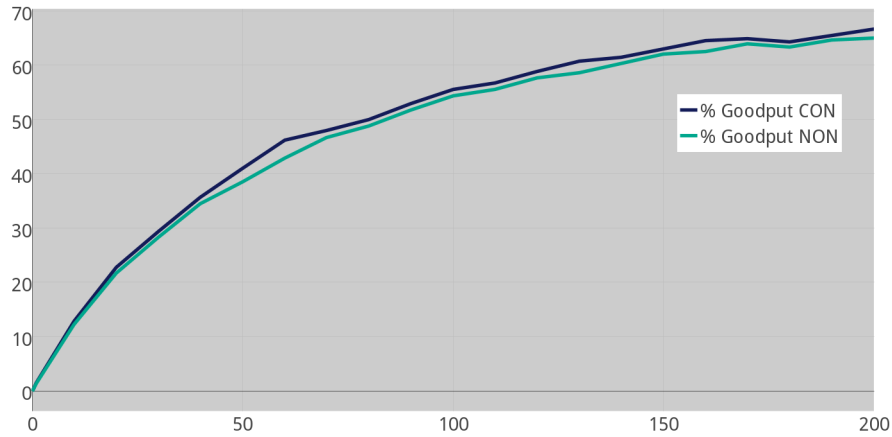

Figure 4.5: CON vs NON 0-200 bytes

Figure 4.5 shows the comparison between sending between 0 and 200 bytes of useful data through the network at once. Keep in mind that this does not show sending rate per time (which will be discussed in chapter), but percentage of the data sent that is useful data. CON can be sent much more often than NON in practice. It could therefore maybe be expected that this could give NON an advantage. .

write chapter

<- Re-write

NON does not need to send ACKs for every package, which should be an advantage concerning to maximize throughput. Still it is the more complex and more frequently sent CON that gets the best results in this test, with a small margin.

4.3.3 CoAP, testing with more data

Table 4.5 shows a bigger and more complex case, where a payload of 700 byte is being sent at once. In total 889 bytes are being sent in this process, with a CON packet size of 774 bytes. This gives a percentage of goodput at 78.74 %. The maximum packet length of a 6LoWPAN packet in this system of 270 bytes is therefore exceeded. 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. 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 in general a good example on 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

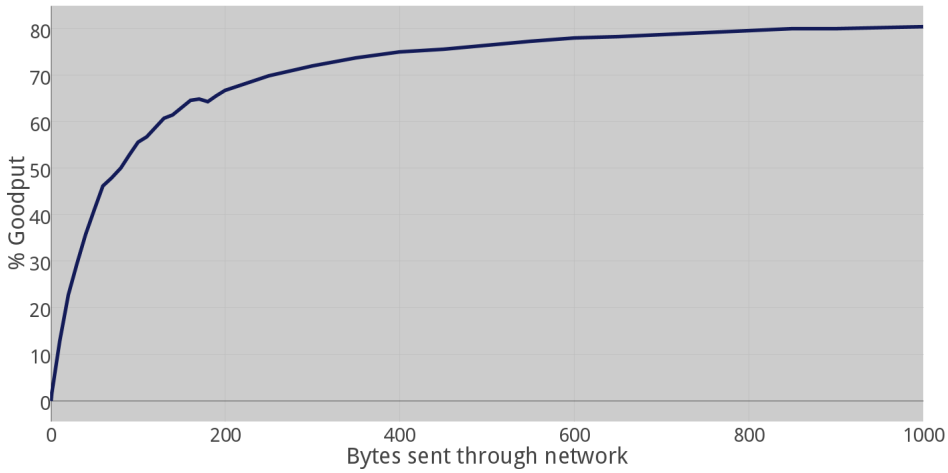

Figure 4.6: CoAP CON plot, 200 bytes to 1 kB

Using these results it was possible to plot the graph in figure 4.6 . Here it is easy to see the same trends as in 4.4, but in more detail over a wider span of bytes sent. These results are as expected after the previous tests, and in compliance with the calculations done in equation 4.1.

Make graph more clear

In this example points in the graph was denoted every 50th byte sent. This gives us

set in graph for 0-1000 before this, to see the flatness?

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 4.6 shows the case where 700 bytes where sent at once through the network, using CoAP NON. The NON packet size is at 777 bytes, which is as expected from the previous tests. This means 4 bytes larger than the same test using CON, meaning an additional header field.

$$\frac{78,47}{78,74} \approx 0,9966 \rightarrow 100\% - 99,66\% = 0,34\% \tag{4.4}$$

Calculations in equation 4.4 show that the goodput here is 78,47 %, compared to 78,74 % in CON. The difference between these two is *0,34 %*, which in this case is considered *very* small. This can also be seen clearly in 4.9. 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 was therefore not conducted in this project.

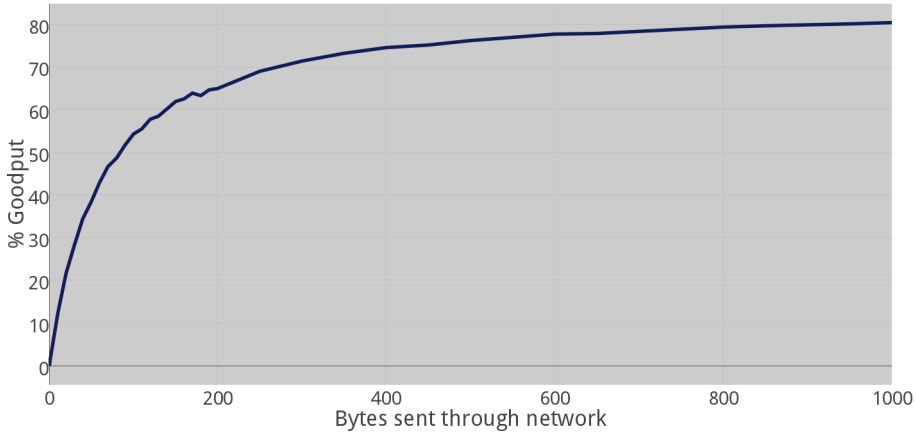

Figure 4.7: NON 0-1000 bytes

The entire scope of the tests done on CoAP NON in this system is shown in figure 4.7. 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 goodput compared to the total amount sent can be very low. On the other hand, the graph stabilizes around 75-80 %. 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 be possible to send as large packets as needed until the limitations of BLE with the same amount of goodput at about 80 %.

Remove figure 4.8 200-1000?

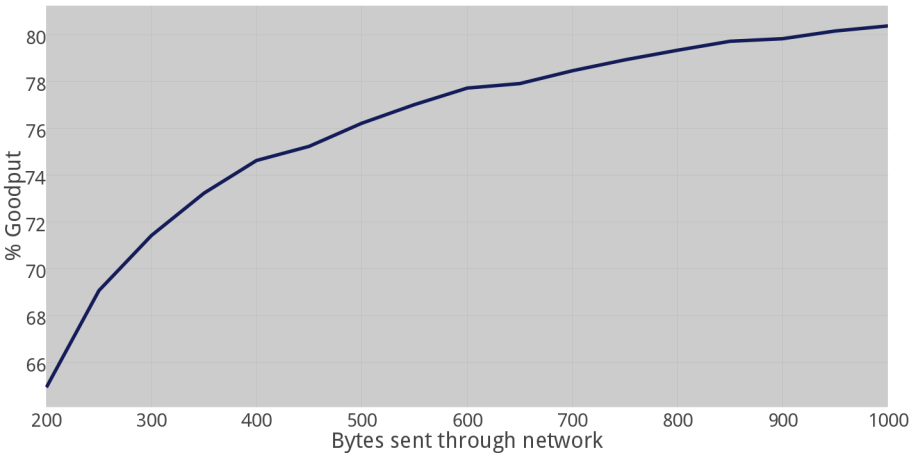

Figure 4.8: NON 200-1000 bytes

4.3.4 Discussion

4.4 Comparison

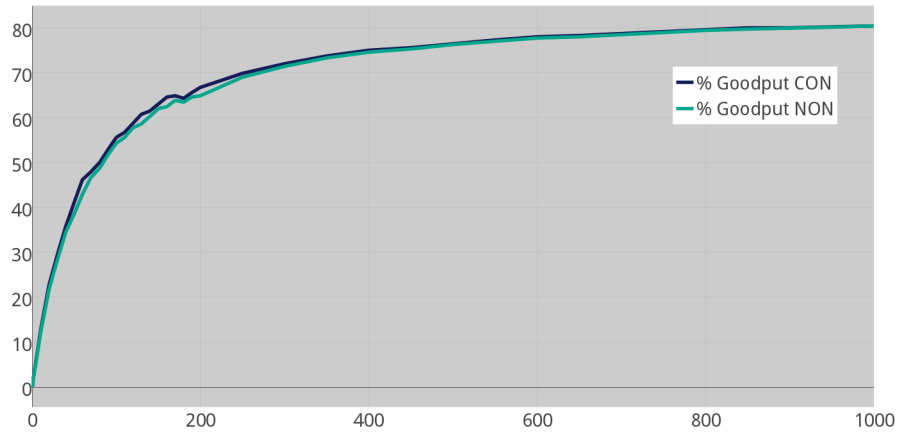

Figure 4.9: CON vs NON 0-1000 bytes

Insert graph of ALL bytes sent?

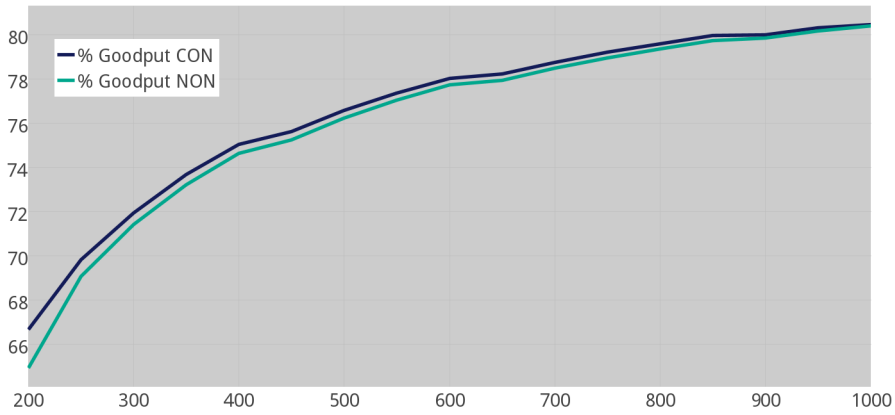

Figure 4.10: CON vs NON 200-1000 bytes

4.4.1 Goodput per time interval

Previous sections in this chapter has shown the percentage of bytes sent through that has been goodput, useful data, compared to the total amount of bytes. This is a good overview of how the protocols are able to exploit the network, and tells a lot about the different protocols. In a real world scenario however, the actual *throughput per time* would be more relevant, since this tells how much data can be transported for every second.

CoAP CON and NON has been compared in this thesis, but in the system build NON was quite unstable at high transfer rates, as explained in chapter 3.1. Because of this, CON seems like a better solution only considering the number of bytes transferred per second. As it turns out, this depends.

As seen in Table , this might be a too early conclusion. insert table

The table shows timestamps of packets received on the Raspberry Pi, sent with both CON and NON. Packets was captured with Wireshark, and the timestamp is created by Wireshark on the form *number of seconds since start of capture*. The first value for each packet size in the table therefore approximately notes the number of seconds from the connection starts until the first packet containing useful data has arrived [2]. The first thing that is noticeable here is the big difference before useful data can be received. Equation **??** shows the calculations of average start time on

[2] values far off from the expected results due to e.g. connection issues have been excluded in calculations

CON, which is a lot higher than for NON shown in equation **??**. The main reason for this is that the NON needs to have a connection establishment using CON packets with ACKs without data before NON packets can be transferred. In addition severe connection issues was experienced in a much higher grade using NON than CON, especially for packets larger than 500 bytes, as shown in table .

Refer to table

Re-write

There are in general two ways to measure throughput compared to time. Either 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, measurements shown in table was used. This represent measurements done several times, and graphs used here are drawn by average values[3].

Insert table?

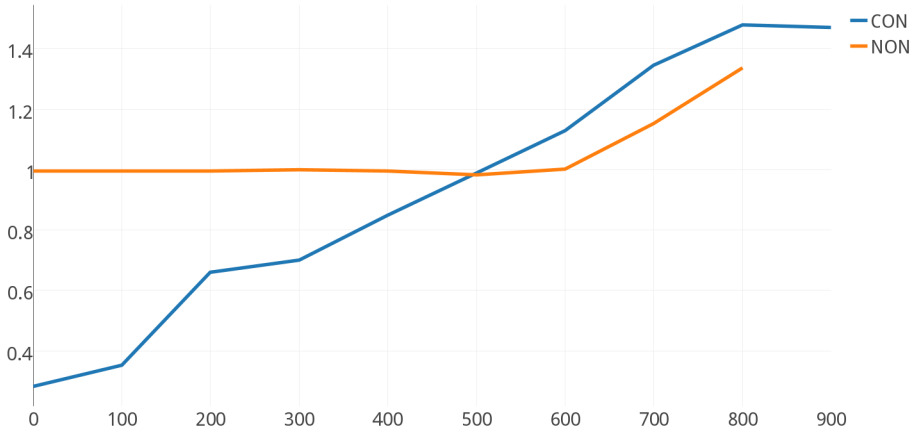

Figure 4.11: Time used per packet

To get a better understanding of this comparison, take a look at the graph in figure 4.11. As explained in chapter 3, tests has shown that CoAP NON is only stable in this system if the transfer frequency is set fixed to once every second. CoAP CON on the other hand can transfer more often if the payload is small. The figure shows how long it takes for the different protocols to transfer a fixed number of bytes of useful data. For instance, it only takes *0.35* 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 *1.12* seconds to transfer the data, while NON still is able to unly use *1* second. After this CON is about *150 ms*

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

slower than *non* to transfer the same payload, as both plots rise with linear growth. 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, glsnon will be preferable, only taking the time to transfer a given number of data one way into account.

From the other approach, it is possible to look at how many bytes it is possible to send every second using the two different versions of the protocol.Figure 4.12 shows the direct correlation between the number of bytes sent every second. For instance: using CoAP CON, it takes on average 7.93 seconds to transport 15 packets of 200 bytes goodput.

$$\frac{200byte * 15packets}{7.93seconds} \approx 378byte/second \tag{4.5}$$

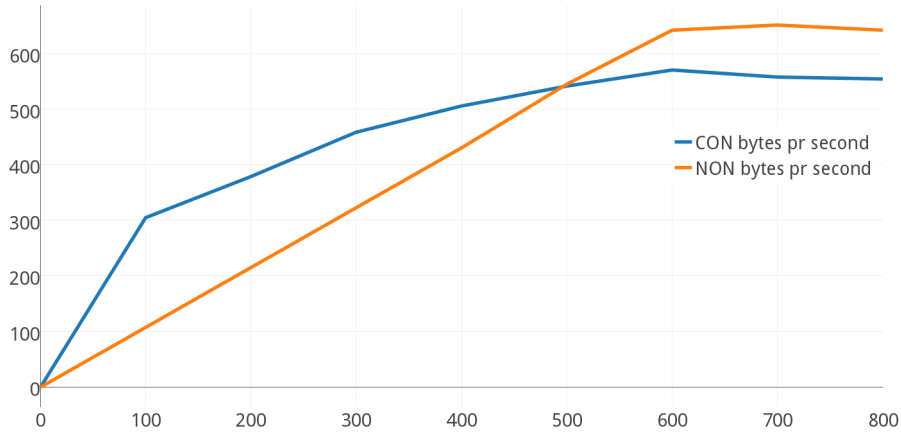

Figure 4.12: Number of bytes per second

In this case both plots starts at *0 byte/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. This makes sence, since figure 4.11 shows that all up to this size has taken *1* second. CON can transfer more often, and this difference can clearly be seen here. When transferring a payload of 100 byte, is almost three times faster at *305 byte/second* compared to *108 byte/second* using NON. The conclusion here will then be the same as in the last graph, CoAP CON is much more efficient for a small amount of data, but for payload bigger than 500 byte NON is preferable.

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 goodput compared to throughput, 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 [4].

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.

But, as seen in figure 4.13, 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 prevevent a situation where the end node sends data forever without anyone receiving the data, ACK messages are still being sent regularly, approximately every 15th second in this network. This means, as shown in figure 4.13, 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 on average:

$$\frac{500 byte payload}{\frac{791 bytes * 15 packets}{15}} * 100\% \approx 63,21\% payload \tag{4.6}$$

$$\frac{500 byte payload}{\frac{(576 bytes * 13 packets) + (856 bytes * 2 packets)}{15}} * 100\% \approx 83,56\% payload \tag{4.7}$$

[4]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

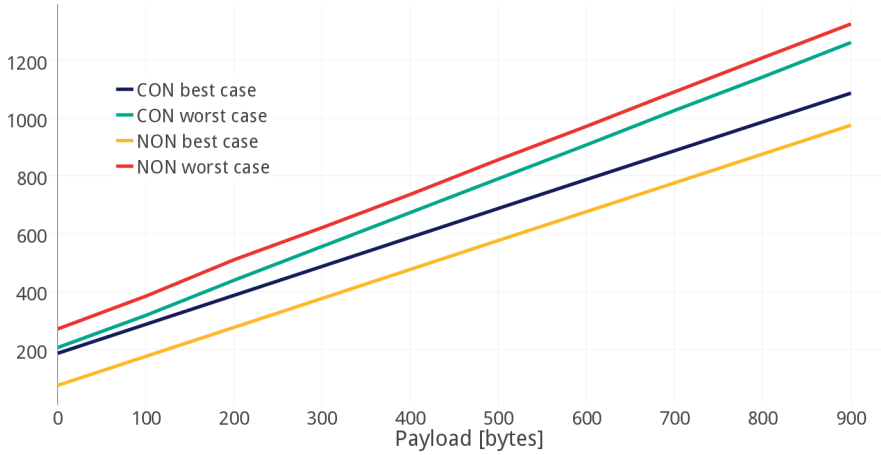

Figure 4.13: Number of bytes per second

From these results it can be concluded that even though NON is considerably worse in worst case than CON,

4.5 Quantity test

Remove?

This section will show a test of several nRF52 devices connected to the same Pi at the same time, to ensure stability and performance in the network. Even though this puts more pressure on both the network and the Pi, it is not expected to be considerably noticed in the transportation of packets.

4.6 Summary

As a summary of this chapter, the positive and negative aspects of the two versions of CoAP that has been tested will be be directly compared to get a more clear understanding of the discoveries that has been made. The results are shown in 4.7.

Table 4.7 shows all the main results found in conjunction with 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 , 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,

Add chapter ref

Table 4.7: 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	1,35	1,15	
Highest measured goodput [bytes] per second	554	642	
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

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 800 bytes. In other cases than this, CON works better in the practical experiments presented.

Chapter 5
Discussion

IDEA: "Collect" the strings put out in the first chapter, which told us the challenges in a system of microcontrollers. This chapter could tell what we learned from the system, and what is still a challenge. Or should all of this be in the Result chapter?

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 with pre-configured kernel of version 4.15 or later on Raspberry Pi 3. In addition it was important to understand how prefix in glsipv6 works, and how this can be used on a bluetooth device. In this solution the end nodes woth 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 is the obvious network technology to use to the end nodes, simply because this is the only antenna fitted to the nRF52. This version of bluetooth is designed to use a minimal amount of energy and still be reliable and fast, which is central criterias in such a network. 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. 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, which worked as expected.

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 points in the network, with the goal of gathering vibration data to be sent through the network. Problems with 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 that the scope of this thesis has a limited time frame, it was decided to measure the different aspects of the network with simulated data. This would eliminate the possibility of margin of error due to sensor error, and can easily be added later by future work with a larger time frame.

Even though this objective was not fully completed, a lot of 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 important methods 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 forward the data to another device or online storage facility. All 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 when it comes to transporting data?

Already in the first tests of *ping* shown in chapter 4.1.1, it became clear that the major limitation 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 the analysing the data sent through the network, and transport protocols used. This problem is assumed to be caused by a problem in a lower level of the protocol stack, and will be left for future work to solve. Other than this, limitations found has been to get the network stable. Several of the solutions tested was not able to transfer data at all, or only for a very short period (< 20 seconds). But when a stable solution is found the link can be open as long as needed, 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.

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, which is 1000 times every second. . 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.

Source?

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 analysing be done in the end nodes in this network?

Referring to the result from research question R.3, where the maximum capacity of the microcontroller is needed to capture acceleration data from the accelerometer to get the desired quality of the data, and forward this to a central node. The end nodes are running on battery power, and is already consuming a lot of energy. To do even more calculations in these nodes will not be preferable in this network. 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 Power consumption

5.5 Ease of use

5.5.1 Raspberry Pi

5.5.2 nRF52

In the case of the system built in this thesis, the small CR 2033 batteries was drained so fast that an additional power source was needed to keep a stable and reliable system, since tests has shown that the probability of an nRF52 disconnecting from the connected BLE service is higher at low battery. This meant connecting either a power bank with higher capacity or drain power from an outlet using a micro Universal Serial Bus (USB).

Chapter 6
Conclusion and Future Work

In this thesis I have used microcontrollers, sensors and single-board computers to build and an IoT network and test the choices when transporting data through the network, and discussed analysis of data with respect to power usage.

The results described in chapter 5 shows that both versions of CoAP tested works in a network like this, but they have different qualities and should therefore be used in different settings. CON is most efficient to send small chunks of data where the extra power usage to send ACKs isn't a problem. NON is the most efficient for payloads bigger than 500 bytes, where the average sending frequency was 200 ms faster than the same size using CON.

6.1 Future Work

6.1.1 Microcontrollers

6.1.2 Sensors

References

[1] About: nordic semiconductor. https://www.nordicsemi.com/eng/About-us. Accessed: 19-05-2016.

[2] ADXL345:digital accelerometer data sheet. http://www.analog.com/media/en/technical-documentation/data-sheets/ADXL345.pdf. Accessed: 20-04-2016.

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

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

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

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

[7] 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.

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

[9] 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.

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

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

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

Chapter

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.

Goodput	Throughput	NON packet size	(Goodput/Throughput)*100
0	71	76	0
1	73	78	1,37
10	82	87	12,20
20	92	97	21,74
30	75	107	30,0
40	85	117	47,06
50	99	127	50,51
60	109	137	55,05
70	119	147	58,82
80	133	157	60,15
90	143	167	62,94
100	153	177	65,36
110	167	187	65,87
120	177	197	67,80
130	191	207	68,06
140	207	217	69,65
150	211	227	71,09
160	225	237	71,11
170	235	247	72,34
180	253	257	71,15
190	263	267	72,24
200	277	277	72,20

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();
   }
}
```

```
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)
         {
            appendCchar(stringToSend, 150, stringa[i]);
         }
      }
   }
   numberOfMeasurements = 0;
}
```

```
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;
}
```