Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Kristian Fjelde Pedersen

Mesh Networking for IoT

Implemented on robots using Bluetooth Low Energy Mesh and Thread

Master's thesis in Cybernetics and Robotics
Supervisor: Tor Onshus
June 2019

NTNU
Norwegian University of
Science and Technology

Kristian Fjelde Pedersen

Mesh Networking for IoT

Implemented on robots using Bluetooth Low Energy Mesh and Thread

Master's thesis in Cybernetics and Robotics
Supervisor: Tor Onshus
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Problem Description

The objective with this thesis is to expand the work done in the project thesis by Pedersen[1] conducted during the autumn 2018. This involves completing and implement two fully functional mesh network applications and interfaces for robots doing simultaneous localization and mapping, known as SLAM. The applications are going to work along and together with existing robot software design and hardware architecture.

The two mesh network protocols should be Bluetooth Low Energy Mesh as well as Thread by using the open source implementation OpenThread which is also mesh oriented to compare differences and possibilities. Both should be implemented with Nordic Semiconductor's Software Development Kits and their System-on-Chip solutions.

Both systems should in the end be compliant with the following bullet points:

- Be a robust communication system
- Connection and network set up should be done automatically on start up.
- Ability for multiple robots to communicate simultaneously using mesh network topology
- Utilizing routing or relaying mechanisms to increase range from today's solution.
- Have a light interface towards the existing robot software to relieve computing power on the robot's CPU
- Be easily scalable with reliable throughput

The student should also elaborate on the benefits and weaknesses with the two systems regarding to be suitable towards the "LEGO project".

Summary

This thesis describes the implementation, use, and analysis of two different communication protocols designed for the internet of things, IoT, implemented as two separate solutions to serve robots doing Simultaneous Localization and Mapping (SLAM). The network topology chosen was mesh, and the two protocols were Bluetooth Low Energy Mesh and Thread. The thesis also covers the relevant theory and a look at the hardware and software used.

The first network protocol was Bluetooth Low Energy Mesh, which was implemented using Nordic Semiconductor's software development kit for mesh and system-on-chip solutions. New models were implemented accordingly to be compliant with the defined Mesh Profile and Model specification, though not all features were taken into consideration. Besides, a UART and a USB interface were set up to communicate with the robots and the host computer as they already did with the previous solution, and to see how the implementation behave.

The second protocol, Thread, was also implemented with a software development kit by Nordic Semiconductor based on the open-source implementation of Thread, OpenThread. The application layer of the Thread implementation features MQTT-SN, which is a publish/subscribe message transport system. A MQTT-SN topic structure for message publishing and subscribing was made to suit the system in the way of best practice. A UART interface was also implemented with the solution to accommodate messages from the robots. The messages were published to an external broker through a gateway located on a Raspberry Pi 3+, which also acts as a Thread Border Router.

Some sensors on the robot were calibrated and tested towards a new server that was implemented at the same time as this thesis. Also, a new communication stack was made on the robot, removing some of its previous parts, to make a more lightweight interface to the new applications.

Different test setups were made to get a deeper understanding of how the applications performed in terms of reliability, scalability, latency, and throughput. Network monitors and sniffer tools were set up for the two applications to enable real-time message transmission between the devices in the mesh networks. Data were extracted to visualize the message flow, if they were lost and how they were processed where that was needed.

Oppsummering

Denne oppgaven beskriver implementering, bruk og analyse av to forskjellige kommunikasjonsprotokoller designet for IoT implementert som to separate løsninger for å betjene roboter som utførerer ”Simultaneous Localization And Mapping” (SLAM). Nettverkstopologien som ble valgt, var mesh, og de to protokollene er Bluetooth Low Energy Mesh og Thread. Oppgaven inneholder også relevant teori, informasjon om både hardware og software brukt i oppgaven, og robotapplikasjonen slik den virker.

Den første nettverksprotokollen implementert var Bluetooth Low Energy Mesh, som ble implementert ved hjelp av Nordic Semiconductors Software Development Kit (SDK) for Mesh- og System-on-Chip-løsninger. Nye modeller ble implementert for å være kompatible med Bluetooth Low Energy Mesh-spesifikasjonen, til tross for at ikke all funksjonalitet ble tatt i betraktning. Dessuten ble et UART- og USB-grensesnitt satt opp for å kommunisere med robotene, som de allerede gjorde med den forrige løsningen, for å se hvordan implementeringen oppførte seg.

Den andre protokollen, Thread, ble også implementert med et SDK fra Nordic Semiconductor basert på en Open-Source-implementering av Thread, OpenThread. Applikasjonslaget i Thread-implementasjonen bruker MQTT-SN, som er et publish/subscribe system for meldingsutveksling. Et UART-grensesnitt ble også implementert med løsningen for å imøtekomme meldinger fra robotene. Meldingene ble sendt til en ekstern server gjennom en gateway lokalisert på en Raspberry Pi 3+, som også fungerer som en Thread router.

Noen av sensorene på roboten ble kalibrert og testet mot en ny server som ble implementert på samme tid som denne oppgaven. En ny kommunikasjonsstack ble også laget til robotprogramvaren, og fjernet dermed noe av den tidligere programvaren, for å lage et mer minimalistisk grensesnitt til de nye applikasjonene.

Ulike testoppsett ble satt opp for å få en dypere forståelse av hvordan programmene presterte med hensyn til pålitelighet, skalerbarhet, forsinkelse og dataflyt. Verktøy for monitorering og analyse av nettverk ble satt opp for de to programmene for å få en oversikt over meldingsoverføringer mellom enheter i Mesh-nettverkene i sanntid. Data ble deretter hentet ut for å visualisere meldingsflyt, hvis og hvor meldinger gikk tapt og hvordan de ble behandlet på ulike punkt med tanke på blant annet forsinkelse.

Conclusion

The goal of this master's thesis has been to complete and analyze the use of two different mesh network protocols and investigate how both can contribute to the ”LEGO robot” project. Through this thesis, I have completed the implementation of two functioning mesh networks utilizing both the Bluetooth Low Energy Mesh and the Thread network protocol for the ”LEGO robot” project at NTNU. The protocols have been used in two separate communication systems that are capable of running on Nordic Semiconductor's nRF52840 hardware with their SDK for BLE Mesh and Thread.

The BLE Mesh implementation features specification compliant models and message exchange for the robots. Once the application has started, it advertises its presence, waiting to be provisioned and get access to a BLE Mesh network. The Thread implementation, using OpenThread, contains features to connect a MQTT broker for message publication and to subscribe to incoming messages. Once started, it looks for a gateway to connect to. Both implementations send messages received from a robot using a UART interface.

The main difference between the two protocols was that they utilize a message flood-based approach and a message routing approach. By monitoring and testing the network, differences have been seen in terms of behavior and reliability. Also, the use of acknowledged messages and unacknowledged message have had a different impact on the systems.

BLE Mesh showed weaknesses when used with the robots, as the message payload is very limited, especially when used acknowledged message sending. Also, many messages were seen lost when they had to propagate through devices to reach their destination.

Thread's utilization of existing and well-proven protocols had a greater performance both in terms of the message payload, throughput, and reliability in the tests conducted.

Through extensive testing, I consider Thread to be the best solution both in terms of reliability, latency, ease of use, throughput and stability. Hence, recommended for further use in the ”LEGO robot”-project. However, based on the test results I have come to the conclusion that BLE Mesh needs throughput enhancements and further development to become more competitive. BLE Mesh proves to be more applicable for lightweight command/control networks where the required data throughput is low.

Preface

This is a master's thesis for a 5-year master's degree at the Department of Engineering Cybernetics at the Norwegian University of Science and Technology (NTNU). The thesis was written during the spring of 2019 and forms the basis of evaluation in the course *TTK4900 - Engineering Cybernetics, Master's Thesis* which counts 30 credits and concludes the master's degree programme.

I was given a robot and its software as well as a java server in the beginning of the autumn 2018 developed by former students. The task was to make a Bluetooth Low Energy Mesh network to suit the robots. This was unfortunately not able to work fully on the robots as it lacked some software components. This thesis' scope have an extended overall objective and scope as the project with making a fully working Bluetooth Low Energy Mesh network and a Thread network. At the start of the thesis I was given both a robot, a desktop computer, and a nRF52832 development kit. A lot of time have been spent doing research and problem solving towards the Software Development Kit from Nordic Semiconductor and the existing robot software which turned out not to work sufficiently.

First of, thanks to Nordic Semiconductor for fast and positive response to provide the thesis with the newest hardware, both dongles and development kits. A huge thanks is given to Omega Verksted, or Omega Workshop, for great expertise and making equipment available for both soldering, doing modifications and signal analysis using their oscilloscope.

Also a big thanks to the fellow students at the workplace on room G242 for excellent companionship and technical discussions. They have been great contributors creating a good and pleasant working environment. I wish them all the best of luck and fortune in the future. At last, a thank to my supervisor Tor Onshus for facilitating and guiding throughout this project and period.

Kristian Fjelde Pedersen

Kristian Fjelde Pedersen — Trondheim, June 2019

Table of Contents

List of Tables

List of Figures

Abbreviations

SLAM	=	Simultaneous Localization and Mapping
PCB	=	Printed Circuit Board
IR	=	Infrared Radiation
CPU	=	Central Processing Unit
GPIO	=	General-purpose input/output
OS	=	Operating System
RAM	=	Random Access Memory
UART	=	Universal Asynchronous Receiver-Transmitter
nRF5*	=	A SoC family distributed by Nordic Semiconductor
SoC	=	System-on-Chip
BLE	=	Bluetooth Low Energy
SIG	=	Special Interest Group
UUID	=	Universally Unique Identifier
TTL	=	Time-to-live
UDP	=	User Datagram Protocol
FTD	=	Full Thread Device
MED	=	Minimal End Device
MQTT	=	Message Queuing Telemetry Transaction
MQTT-SN	=	Message Queuing Telemetry Transaction Sensor Network
QoS	=	Quality of Service
SDK	=	Software Development Kit
API	=	Application Programming Interface
NCP	=	Network Co-Processor
ARQ	=	Automatic Repeat Request
SysML	=	System Modelling Language
RTT	=	Real-Time Transfer

Chapter 1

Introduction

1.1 Background and motivation

Instead of having a thought that the internet is only for humans, an idea was brought to life early in the 90's that even machines could have great advantages being connected to networks. That made the basis for the term internet of things or IoT. And from the internet of things has new protocols and standards emerged to enable even low power devices connectivity by accomodate lower bandwith and higher link failures.

As hardware and software develop and get faster and more optimized, opportunities are getting wider. In the past years, robots have taken more and more over tasks earlier meant for humans due to higher efficiency and lower costs. It can also be because they can get to places humans can not or that the risk of doing so is too severe.

A fundamental challenge with robots have been to be aware of their surroundings and locate themselves correctly in new and changing environments. The "LEGO robot"-project at the Norwegian University of Science and Technology is trying to approach this problem and both reliable communication and processing has so far proven to be key factors for success.

1.2 Previous work

The LEGO robot project was started at NTNU in 2004 where the initial goal was to create a robot, based on LEGO Mindstorms, that mapped an area and reported to a host computer. By utilizing hardware and sensors, a simultaneous localization and mapping application was made to enable the robot to navigate an unknown maze. Since 2004, many students have contributed to the project in one way or another. More robots have been created with different off-the-shelf hardware and specifications, but the goal has mainly stayed the same. The end-goal is presumably a swarm of robots working together to solve a SLAM problem.

This thesis is not a direct continuation of other students' work, but the last student, besides my specialization project[1], working on the communication system was Lien [2] in 2017. He enabled a new communication stack on the robot to reduce communication errors by implementing error detection and recovery by retransmission.

The work done in the specialization project mentioned during the autumn 2018 makes a basis for the work in this thesis.

1.3 Limitations

The work conducted in this thesis is limited to developing software and making alterations to the communication system on the robot only except for a change in one sensor driver. No other software, like the different tasks on the robot, have been modified.

1.4 Thesis Structure

This thesis is written and compiled in LaTeX using the online compiler Overleaf v2. The table of content, figures and tables, cross-references and links are hyper linked to make digital navigation easy.

Chapter 1 - Introduction

Chapter 2 - Theory: Gives an brief introduction to Bluetooth and mesh network. A more in-depth look at Bluetooth Mesh, what makes up Thread, MQTT and serial interfaces used in this thesis are also covered.

Chapter 3 - Development tools and hardware: Covers development tools and hardware used in this thesis as well as what hardware the robot exist of that has to be taken into consideration.

Chapter 4 - Robot Application: Goes into what software related to communication is being used and short on what in the system today and which earlier students and theses these are a result off.

Chapter 5 - Bluetooth Low Energy Mesh Communication System: Shows the way necessary models have been implemented and what have been thought of in the process. It does also mention the drivers and additional method and features of these.

Chapter 6 - OpenThread Communication System: Shows how the MQTT client API have been utilized to make the application.

Chapter 7 - Test setup and experiment: Walkthrough of how tools were utilized to capture packets and analyze performance as well as physical test set-ups.

Chapter 8 - Results: Results based on data from the test set-ups. Showing different properties of the two applications.

Chapter 9 - Discussion and further work: Digs into the results, what was being found out during the thesis and an overview of the main tasks for further work.

Chapter 2

Theory

This chapter encapsulates the fundamental theoretical basis to understand what is described further in this thesis. Most of the theory making up section 2.2 to 2.3 is from the project report [1] and is equally relevant for this thesis.

2.1 Mesh Networks

Mesh network is a term used for a network topology where there are several devices that are either directly or dynamically connected depending on how it is set up. It enables multiple devices without a direct connection to communicate through other *relaying* devices. The way relaying works is either by a flood-based or a routing approach. Flood-based means that a device sends a message to multiple directly connected devices for them to relay further until it reaches its destination device. The routing approach involves having a defined path that messages propagates through, hopping from node to node until arrival at its destination. Such paths are commonly stored in a routing table which gets updated when there are changes in the network.

2.2 Bluetooth Low Energy

Bluetooth is a standard for wireless data transmission over short distances. It was first introduced by the Swedish company Ericsson in the year 1999 and was later formalized by Bluetooth Special Interest Group which today is having 30.000 member companies[3]. The standard is broadly used in both fixed and mobile devices and shortly after its release it was found in many cell phones introducing a new way of exchanging data and files between users.

It aimed to replace wires and consume a low amount of energy to able to run easily. As Bluetooth was standardized, it became, and still is, highly compatible across devices. The benefits of Bluetooth are that it has low interference by using frequency hopping mechanisms and that it is automatic. The latter in terms of that there are neither any buttons to be pushed, nor any connection to manually be set up rather than accepting to connect with a device or not. It is estimated that 4 billion devices with Bluetooth were shipped in 2018 alone [4].

The term Bluetooth Low Energy (BLE) originates from version 4.0 of Bluetooth that hit the market in 2011. The main difference between Bluetooth 4.0 and earlier versions is that it uses less power by going into sleep when no connections are initiated. In December 2016 Bluetooth 5.0 core specification was released and had the intention to increase both range, message capacity and speed of applications.

2.3 Bluetooth Mesh

In July 2017 Bluetooth SIG introduced a standard to enable mesh networking using Bluetooth Low Energy. Its main goal was to enable many-to-many connections between communicating nodes in, for example, large-scale industrial applications. For mesh networking, the group chose a managed flood-based approach[5] where devices can be configured to either relay messages or not based on power settings and requirements. The group reasons their choice of being simple, easily scalable and reliable. BLE Mesh is based on the Bluetooth 4.0 protocol and not the newest 5.0. The network supports at maximum 32768 devices connected, but in terms of scalability, Bluetooth SIG has released another phrase stating:

”Scalability: The maximum, total number of mesh messages per second which can be communicated between nodes in direct radio range of each other, with no more than x.x% message loss.”[6]

Where the message loss is up to each vendor or user.

2.3.1 Layers

The system architecture making up BLE Mesh is divided into different abstraction layers called the Mesh Profile Specification[7] often referred to as the ”Mesh system architecture”. These are models, foundation models, access, upper and lower transport, network and bearer layer. The core specification is what makes Bluetooth 4.0. I will only cover some of them as not every layer is of importance for this thesis.

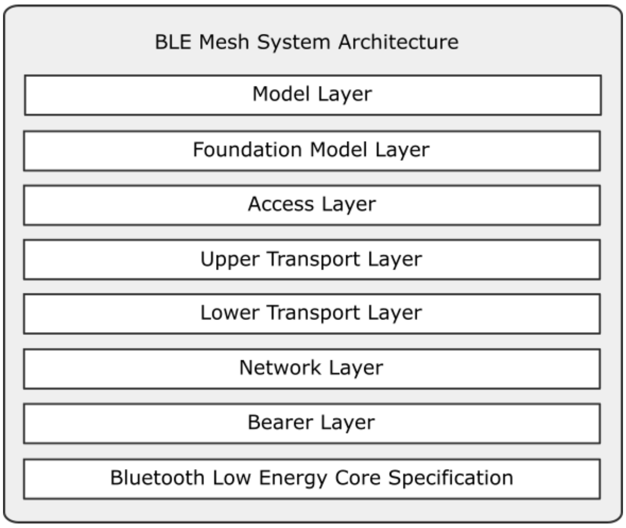


Figure 2.1: Stack overview

As for this thesis, Nordic Semiconductor is providing a software development kit abstracting the implementation of the layers, but not the models. Those are the ones we have to implement in order to define how communications are taking place. More on this in section 2.3.2. Though, it is important to know of the access layer as it is the highest layer to the models and contains the API needed to use BLE mesh. The access layer is responsible for controlling authentication, encryption, and decryption of data being sent between the application and the transport layer to send and receive messages. It is also through the access layer that models get registered onto the device model database.

2.3.2 Models and roles

To define the functionality of nodes in a network Bluetooth SIG has introduced Mesh Model Specification[8]. The idea of a model is that each model has a set of associated states, messages and behavior that are used for interaction. In the Mesh Model Specifi-

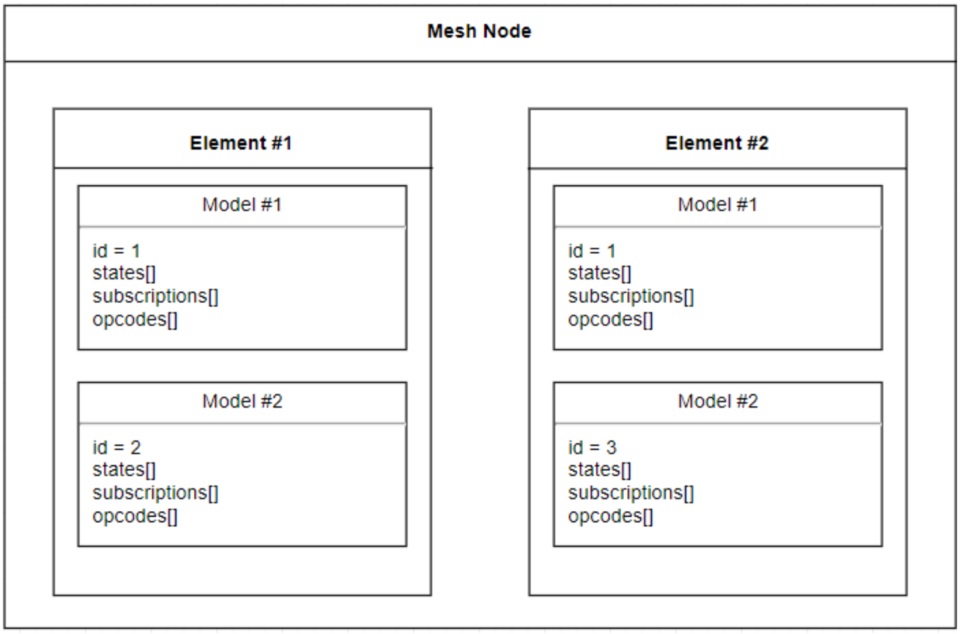


Figure 2.2: Mesh Node Access Structure

cation, Bluetooth SIG has a vast list of generic models to use as a base for model development. However, one is always free to develop custom models, but since the specification covers many use cases, it can be useful to use generic models to ensure specification compliance. Bluetooth SIG is still working on making new generic models in their specification for the increasing demand[9]. In a network, each node is given models based on which role it contains in the network. We divide these roles into provisioner, client, and server which will be presented below.

Apart from models, a node also contains one or more elements. Elements work like virtual entities on the node within the mesh with its own address. This enables us to have multiple instances of the same model within the same node with equal operation codes and handlers. Each model, with its associated behavior, is assigned to an element. Figure 2.2 shows an example of a mesh node with two elements with two associated models where both elements contain the model with id 1.

Sending a message onto the network is known as publishing. Some nodes might contain models defined to publish periodical messages containing states such as sensor data or to publish messages when needed or requested. On the other hand, we have subscription which is the act of listening to incoming messages of certain addresses and handle these.

Any nodes that have models in need of data for processing or control to be received are subscribing.

Upon receiving, the models receive an opcode, operation code, from the access layer to be handled in the application. These opcodes are commonly defined for each model depending on what its tasks are. An opcode's size is either 1, 2 or 3 octets, depending on whether it belongs to a manufacturer specific or a generic model.

Models that are subscribing to addresses and which receives messages and commands are known as server models. These models are also subject to change in states. On the other side, we have the models that send messages and these are known as client models. Unlike the server model, the client model does not contain any form of state and therefore its main tasks are to send messages and receive acknowledgments and statuses. For many tasks, it would be convenient to have an application running both server and client models on the same device.

Another important role in mesh networking besides client and server is the provisioner. Its main task is to add devices to the network, making it a node in the network. Its properties and functionalities are vastly described in the Mesh Profile Specification[7]. The provisioner starts by scanning for unprovisioned devices which are advertising their UUID. When an UUID is detected and it matches a defined filter, the provisioner sends an invite to the unprovisioned device. If the invite is accepted, the device is being called a provisionee and the provisioner will ask for the capabilities and provisioning method of the provisionee. After that, the provisioner will exchange network keys and the provisionee returns them to validate that they are correct and has not been corrupted. After this is completed, a provisioning complete message is being transmitted and the provisioning is complete.

For a device to be a part of a mesh network and be provisioned it also needs two additional models I have not yet mentioned, a configuration model and a health model. Upon successful provisioning, the provisioner starts a process known as configuration. This procedure involves fetching relevant composition data and setting statuses through a configuration client and server model.

The health model contains functionality to send messages periodically to the provisioner so that the provisioner knows if a node is still connected to the network or not. These kinds of messages are commonly known as keepalive or heartbeat messages. The mesh device specification specifies that the configuration and health model is supposed to be located at the first indices at the first element of the node. Note that a node can be connected to a mesh network if it fails to be configured, but it will not be able to either subscribe nor publish to any other node or addresses.

2.3.3 Messages and structure

Messages, according to the mesh profile specification, are consisting of an opcode, associated parameters, and behavior. The transport layer of the mesh system architecture enables a segmentation and reassembly mechanism enabling up to 32 segments as a message where each segment consists of 12 octets excluding the opcode. This allows for a maximum of 384 octets for each message. On the network layer, messages consist of information whether the message is segmented or not, which segment number, source address, destination address, and data.

Messages do also contain a value not mentioned which is Time-to-live (TTL). TTL is a value being used for flood-based mesh networks and decides how long a message will be relayed. For each time a message is relayed the TTL is decreased. When it is zero, it will be discarded.

2.3.4 Security and reliability

As mentioned above, the provisioner exchanges a network key with the provisionee, and it does so with every device being connected to the network. This is what defines the node to be part of the network - to have the key in its possession. Once a node receives a message, it checks if the network identifier is known. If not, it gets discarded. Then it checks if the network key it has can verify the message integrity. If this fails, the message is discarded. After verifying the integrity it validates the source and destination addresses and checks if it already exists in the cache. If invalid or already received, it will again be discarded. In the end, it gets forwarded to the upper layer and gets relayed if the TTL is not zero.

Also during configuration, after the provisioning phase, the provisioner binds an application key to the models ensuring secure communication between the access layer and the model. The application key is held by the provisioner and is shared across all nodes participating in that particular application.

The last key is the device key which is unique for each node. The node exchange this key with the provisioner to ensure secure communication between the device and the provisioner. By all these concerns and specifications Bluetooth Mesh communication is considered heavily secure.

2.4 IEEE 802.15.4, IPv6, MLE and 6LoWPAN

IEEE 802.15.4

The IEEE 802.15.4 technical standard was first released by the Institute of Electrical and Electronics Engineers in 2003 with several revisions since then. It covers the physical and the data-link (MAC) layer for low-rate wireless personal area networks (LR-WPAN) and is therefore suitable for low-cost and low-power devices communicating wirelessly, such as IoT devices. The basic framework supports a range of communication of about 10 meters with a transfer rate of 250 kbit per second. There is also support for lower transfer rates for devices with very low power requirements. The data-link layer features an acknowledgment mechanism to verify a message's integrity when transmitted.

IPv6

IPv6 is known as Internet Protocol version 6 provides a localization and identification system for devices on networks and traffic routing across the internet. All devices on the internet have a unique IP address. The former version of the internet protocol, IPv4, enabled 32-bit addresses for connected nodes which is 2^{32} unique possibilities, or approximately 4.3 billion. However, by the growth of connected devices, a new standard had to be developed to cope with the increasing amount. IPv6 enables 128-bit addresses which increases the number of unique addresses humongously.

6LoWPAN

According to the IPv6 standard [10], the smallest MTU (maximum transmission unit) of a packet is required to be 1280 bytes. Yet, the largest MTU for IEEE 802.15.4 frames is 127 bytes. To achieve IPv6 packets being sent over IEEE 802.15.4, it is needed to compress the packets and that is where 6LoWPAN comes in. Although it leads to that the amount of payload is drastically reduced, but a large amount of payload is often not needed for low powered applications and devices.

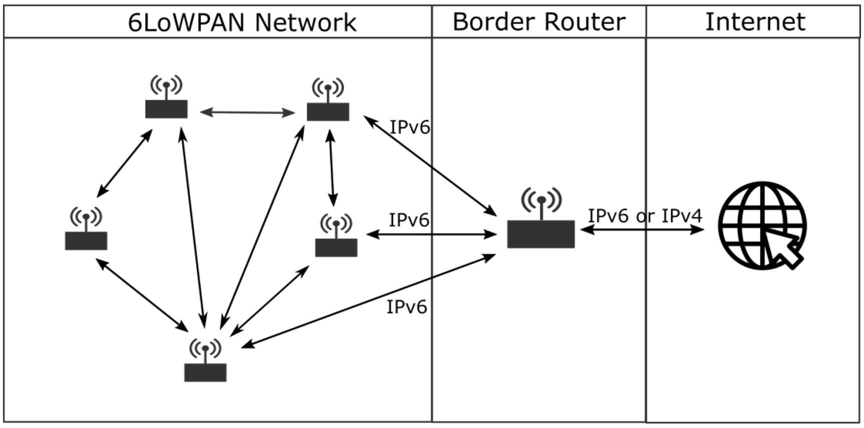


Figure 2.3: Illustration of 6LoWPAN

MLE

MLE stands for Mesh Link Establishment and is a protocol[11] for defining as well as establishing and configuring radio links securely in IEEE 802.15.4 mesh networks [12]. According to its standard, MLE extends IEEE 802.15.4 for use in multihop mesh networks utilizing three capabilities:

1. Dynamic configuration and securing of radio links
2. Enabling network-wide changes to radio parameters
3. Detecting neighboring devices

2.5 OpenThread

OpenThread is an open-source implementation of the network protocol Thread which is released by Nest. Nest is a company founded in 2010 which develops consumer products related to the smart home industry from connected door locks to smoke detectors and thermostats. The implementation of Thread was mainly released to enable broader access for developers to advance the development of products related to smart home technology or wherever it may be applicable.

The idea behind the network protocol is to bring embedded devices onto the internet by using open standards, those being IPv6, 6LoWPAN and IEEE 802.5.14, those explained in 2.4. Figure 2.4 shows the protocols so far described as well as MQTT which will be explained in detail in 2.6.

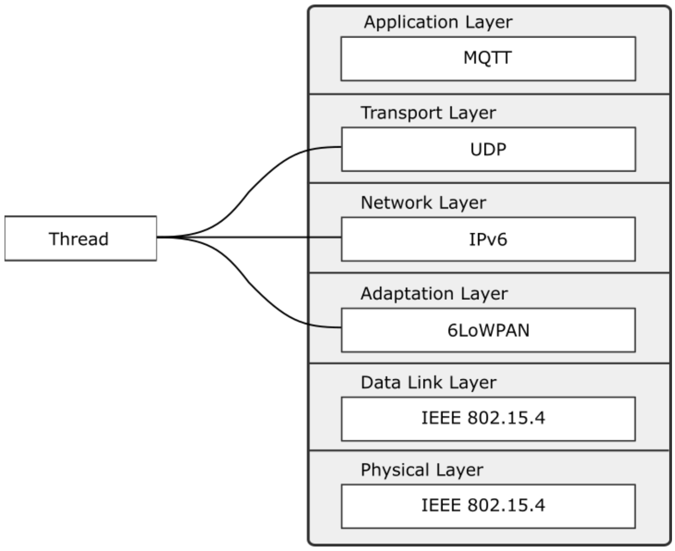


Figure 2.4: Adapted OSI model representation

2.5.1 Routing, commissioning and roles

Within a Thread network, there are different roles that devices can have. The two main roles are Router and End Device and can also be referred to as parent and child. A router's responsibility is to forward network packets to connected devices and to provide secure commissioning for new devices wanting to join the network. A router needs to have its radio active at all times. An end device is considered a child of a router and can thereby only communicate with a single router and it can not forward any network packets.

Other roles are the leader and border router. The leader is a self-elected router class that is responsible for managing the routers within the network and distributing configuration information across the network. The Border Router is also a router class, but is unique in the way that it is connected to an external network such as the internet or any other local networks via Ethernet, WiFi, etc. It forwards the information in either direction, to and from the Thread-network. Any device can operate as a Border Router, but only one device can operate as a leader.

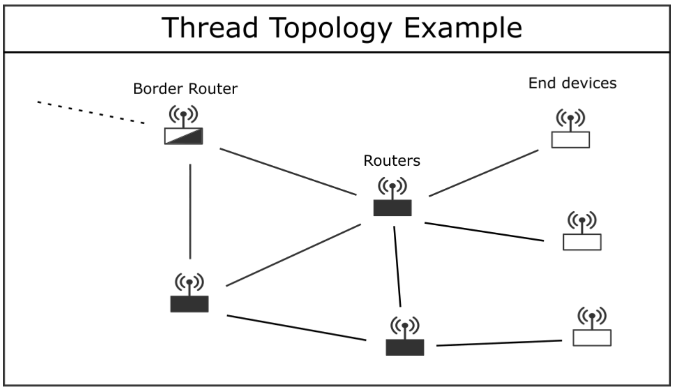


Figure 2.5: Thread Topology Example

Figure 2.5 shows a brief example on how the topology is within the network. Note that any of the routers can be self-elected a leader thus it is not illustrated.

Also, there are terms describing each network device as they can have additional features and responsibilities. These are categorized as follows:

- **Full Thread Device (FTD)**

  Always has it radio on, maintains IPv6 addresses and listens to multicast addresses.

  1. Router
  2. Router Eligible End Device (REED)

     The router can be self-promoted to a router
  3. Full End Device (FED)

     Cannot be promoted to router

- **Minimal Thread Device (MTD)**

  Listens and forwards only to its parent router.

  1. Minimal End Device (MED)

     Radio on
  2. Sleepy End Device (SED)

     Radio off normally and polls messages when awake

All FTDs can operate as either parent or child. MTDs can only operate as children. For a Thread Network, the maximal amount of routers are 32, and the maximum number of end devices are 511 per router.

2.5.2 Network creation and addressing

A Thread network consists of three identifiers:

- **Personal Area Network ID** (PAN ID) of 2 bytes
- **Extended Personal Area Network ID** (XPAN ID) of 8 bytes
- **Character-based name** of optional size

When a device wants to connect to a network or create one it starts by broadcasting an 802.15.4 beacon request to search for other 802.15.4 networks on a specific channel. If a network already exists or is found, any routers or REEDs in the range of the new device broadcast a beacon that contains their three identifiers as listed above. The device continues to do this on multiple channels and when it decides that all available networks have been found, it can either connect to one or form its own. To configure links between devices, MLE as mentioned in 2.4 is used. The way linking and routing of messages works is similar to the Routing Information Protocol (RIP) [13] which is a standardized Distance Vector protocol.

In Thread mesh addressing, we divide addresses into different ranges. These are Link-Local, Mesh-Local and Global.

1. **Link-Local** is the closest addressable devices in radio range

   Addresses used: ff02::1 and ff02::2

2. **Mesh-Local** is all addressable devices within the same mesh network

   Addresses used: ff03::1 and ff03::1

3. **Global** is all addressable interfaces outside the Thread Network, for example, the internet.

   Addresses used: IPv6 addresses

With in the two first, it is possible to do a multicast message sending where information is communicated to multiple devices simultaneously.

2.6 MQTT

Message Queuing Telemetry Transport (MQTT) is an application layer protocol that implements a lightweight, simple and open publish/subscribe messaging transport for machine to machine communication. It uses the TCP protocol for its transport. MQTT is an

OASIS standard whereas OASIS is a global nonprofit organization for open standards for many different areas such as the Internet of Things and security. It became an OASIS standard in 2013 with its 3.1.1 version and the newest version 5.0 is also a separate standard in the OASIS system as of 07 March 2019 [14].

A client/server publish/subscribe pattern is different from a client/server request/response architecture wherein the latter, the client communicates directly with the server by requesting data or services and the server responds. The publish/subscribe pattern decouples the clients that send messages, known as publishers, from the other clients that receive messages known as subscribers. Instead, for a publish/subscribe pattern, there is a third component which in MQTT is called the broker or server. Both terms are used, but the MQTT standard[15] specifies server as correct notation. However, the word broker itself is a more accurate notion of the server's properties and characteristics and to not be easily confused with other server notions. The broker's job is mainly to *distribute* and *filter* messages between the publishers and subscribers.

Figure 2.6 shows a basic principle on how the messaging works. The subscriber client at the right sends to the broker that it will subscribe to a certain topic. When another client, a publisher, then publish a message onto that topic, the broker will distribute that message to all subscribing clients.

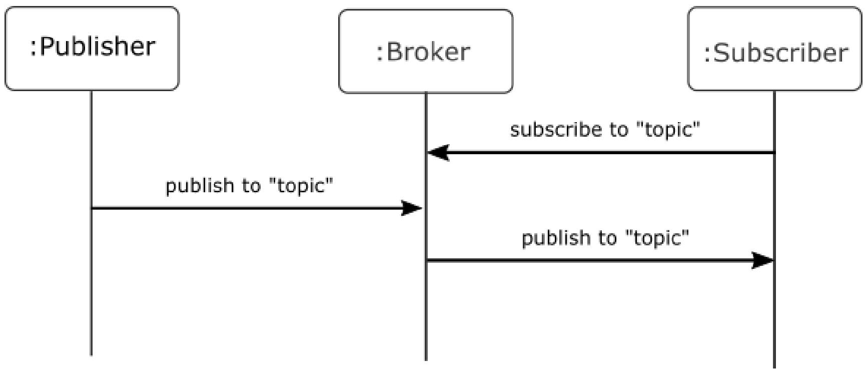


Figure 2.6: Minimalistic MQTT illustration

A major benefit with this approach is that the publishers and subscribers do not need to know of each other, neither do they have to run at the same time. This is called decoupling and makes it easier for clients in terms that they do not have to be aware of each others IP addresses or ports, only the one belonging to the broker.

2.6.1 Topics, will and retain - A use case

In figure 2.6, the term topic used is also a central part of MQTT not yet explained. The topic of a message is used by the broker to decide to which client the message should be distributed to. Topics are strings which follows a hierarchical structure. By hierarchical structure, topics can be in different topic levels separated with a forward slash or topic separator like file system paths presented in figure 2.7.

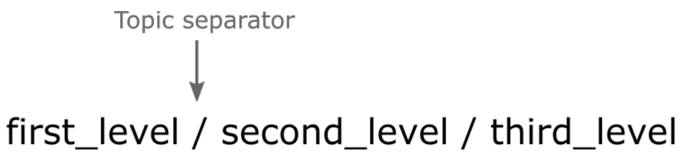


Figure 2.7: MQTT Topic Hierarchy

A typical use case for this topic levelling is for example sensors with certain properties like locations, types, serial numbers etc. For that use case a topic structure could be as in figure 2.8 where the location could be Trondheim, specified by the following value NTNU. Additionally, for example which room and which property or type of sensor. Room G232 and temperature respectively.

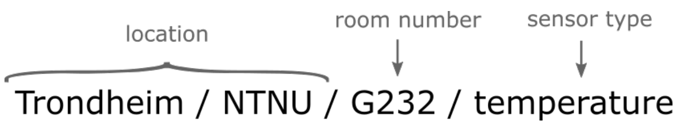


Figure 2.8: MQTT Topic Hierarchy Example

A client, which would be a temperature device, in room G232 can send a message onto this topic containing the temperature values. A subscribing client on the other hand can subscribe to the same topic in order to receive the temperature values to process or store them. Imagine now that there is a server based on NTNU which collects temperature values for all available rooms, or even every available data, it would be very ineffective to subscribe to each topic.

For solving this issue, MQTT topics implements wildcard functionality enabling a client to subscribe to multiple topics simultaneously. There are to wildcard characters that can be used:

- **# (Hashtag)**, used for multi level wildcard
- **+ (Plus)**, used for single level wildcard

By using this, a client can subscribe to for example *all* room's temperature devices by using the single level character as the upper topic string in figure 2.9. The lower topic string is for all room's devices despite which type of sensor.

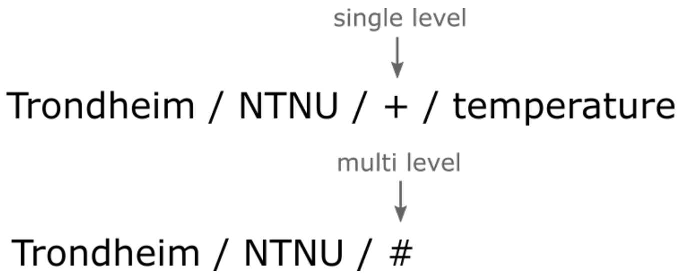


Figure 2.9: MQTT Topic Hierarchy Example

Two more useful features in MQTT are retain and will. A message can be retained, which involves that the broker will store the last message to the specific topic the message was sent to. While always having the last message of a topic available locally, the broker will publish this message to every new subscribing client to that topic. For instance if a client connects and subscribes to the topic for the temperature at G232 as in 2.8, the broker will send the latest temperature value update available for that room immediately after subscribing.

Will is a feature that enables clients to notify, on a specific, topic that it has been disconnected from the MQTT network unexpectedly. Normally, disconnection from a network happens through a DISCONNECT message to the broker with a reason code, if that is the case, the will message will not be published. However, a client can also request the will message to be published even if the disconnect procedure is normal. The will topic and message is sent to the broker upon connection.

2.6.2 Quality of Service

Quality of Service (QoS) is a term used to describe an agreement of message handling between publisher and receiver and is a key feature of the MQTT protocol. It is commonly known as a level of guarantee of delivery for a message. The QoS level is a number between 0 and 2 with these properties:

QoS	Property
0	At most once
1	At least once
2	Exactly once

Table 2.1: The levels of Quality of Service

For a QoS level 0, the publisher publishes the message without any need of storage or acknowledgment of it being received. This can be compared to a UDP packet. Figure 2.10 shows a simple sequence diagram on how the transaction happens.

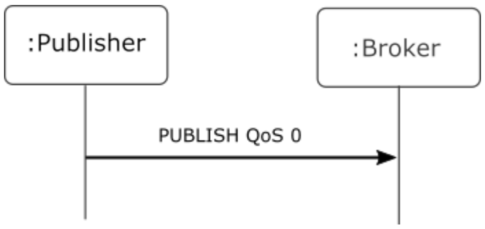


Figure 2.10: QoS Level 0 Sequence Diagram

As for QoS level 1, it is guaranteed that a message is received *at least once*. This implies that the message can be received multiple times. The sender stores the message until it gets a publication acknowledgment as shown in Figure 2.11. At first sight, an application going from QoS 0 to QoS 1 may suffer a greater processing problem.

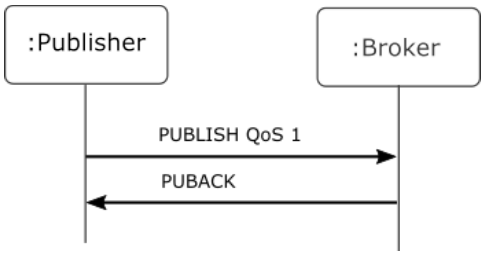


Figure 2.11: QoS Level 1 Sequence Diagram

At last, for QoS level 2, we can guarantee that it is received *exactly once*. This is without a doubt the safest, but at the same time the slowest QoS level. The process includes a four-part handshake. And relating to processing complexity, this is the most difficult QoS level, and for an embedded system it might get pretty tricky.

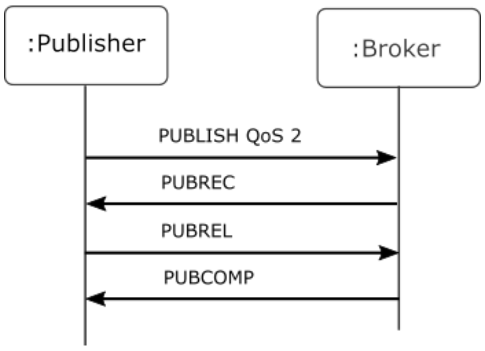


Figure 2.12: QoS Level 2 Sequence Diagram

2.7 MQTT-SN

MQTT-SN is MQTT for Sensor Networks and has several differences from the traditional MQTT protocol specification though it is designed to be as similar as possible. It has been designed to adapt to the differing characteristics of wireless communication networks with lower bandwidth and higher link failures, in addition, to being optimized for implementation on devices with limited storage, resources, and processing power.

The MQTT-SN standard inherits the traditional MQTT structure consisting of clients and a server (broker) but adds a gateway as a component. Its role is somewhat similar to that of the border router described in section 2.5.1. Figure 2.13 shows an example of how the communication flow between the broker, gateway and the wireless devices.

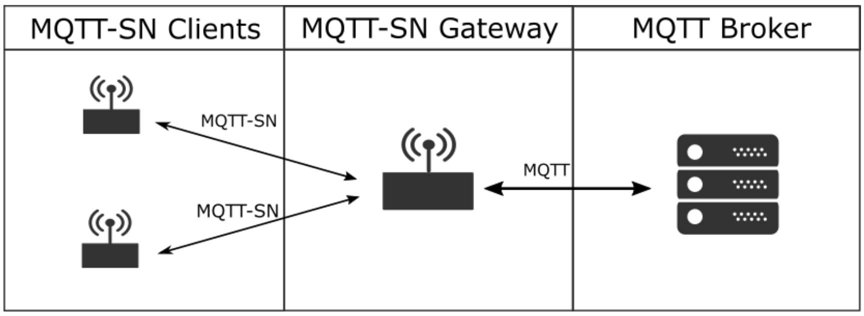


Figure 2.13: MQTT-SN Gateway Illustration

The gateway can be either transparent or aggregating. Transparent involves that the gateway will set up and maintain a MQTT connection to the broker for each connected MQTT-SN client. The number of connections to the gateway will be equal to the number of connections from the gateway to the broker and the gateway only works as a translator or syntax paraphraser between MQTT and MQTT-SN. The aggregating gateway does not require one broker connection for each client, instead of the gateway only maintains one connection despite the number of clients. This might come in handy for brokers handling large amounts of clients concurrently. The gateway do also stores client registered topics and assigns each an ID to be returned to the client, such that the client only uses the ID to publish messages instead of the entire topic string which optimizes message payload.

2.7.1 Gateway connection procedures

A gateway advertises its presence in a set interval when it is connected to a broker. This procedure is, for example, done to notify connected devices if there are getting more gateways connected to a network. When a client search for a gateway, it transmit a search gateway signal and wait for an eventually gateway to respond with info. If there are multiple gateways, the client will choose one of these dynamically based on the information exchanged.

Once the client has decided, it sends a connection request with the relevant properties or flags. This is the point where the gateway checks if will topic and message is wanted to be registered, and it will following ask for these if set. The gateway stores the data and transmit them to the broker as well. If the connection is refused by either the broker or gateway, the client shall receive a connect acknowledgment containing a reason for refusal. If it is successful, the client is granted connection and can proceed by register topics or thereafter either publish or subscribe to these.

2.8 Serial Interfaces

Communication across embedded devices is required to have a defined protocol and fixed way of data transfer so that the errors are minimized, the applications can be optimized and stay efficient. Over the years, the world has seen many different protocols for different uses where some are more common than others and are being used literally everywhere. From computer processing units to sensors or simply your keyboard and mouse to your computer.

2.8.1 UART

UART stands for Universal Asynchronous Receiver-Transmitter and is a hardware serial interface which is the name of the device that allows a computer or a microcontroller to communicate via serial line. The term asynchronous means that there are no clock data involved transmitted directly with the serial data. The UART takes bytes of data, transmits each individual bits in a sequence and shifts them out of the device one at a time. One bit for each pulse of the serial clock. The receiving UART is locking onto the stream of bits, re-assembles them and sends them to the CPU.

Bits sent from the UART module is commonly represented by logic level voltages. There are standards defining how these voltage levels are interpreted. The most common ones are RS-232, RS-485

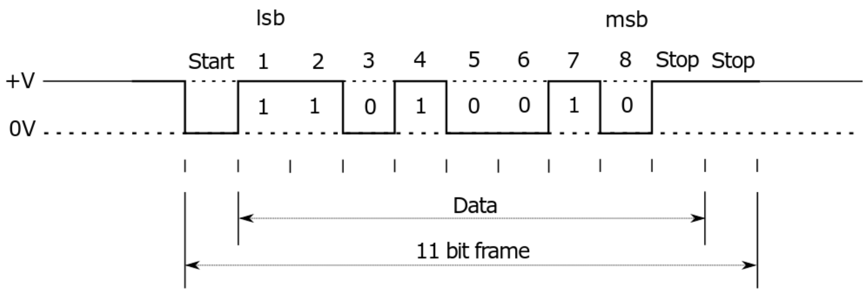


Figure 2.14: Asynchronous serial data

Figure 2.14 illustrates asynchronous serial data communication where this transmission format starts by using one start bit at the beginning of the sequence and two bits at the end to specify stop. Upon receiving a start bit, the receiver synchronizes its clock and samples the following bits. When the stop bits are received, the receiver assumes the transfer was successful and all bytes were valid. If the receiver does not receive the stop bits, it would raise an error which is either a framing error, clock phase error or a bit alignment error. For example for Nordic's devices, the application has to handle these errors when they occur. More on this in section 5.1.2.

The robot, which I will go into in chapter 4 is containing an implementation for a USART driver. USART stands for Universal Synchronous/Asynchronous Receiver/Transmitter, and has, apart from UART, the ability to be set up in a synchronous mode. As stated earlier, for UART, it is the start bit that makes the microcontroller synchronize the clock signal. In synchronous mode, USART will generate a clock that can be recovered from the datastream on the receiver side without knowing the baud rate on beforehand. Also

USART supports a greater number of protocols.

2.8.2 USB CDC ACM

USB is known as Universal Serial Bus and is a serial bus commonly used to connect devices to a computers. It is in fact a industry standard which defines both cables, connectors, communication protocols and power supply between electrical equipment and computers. A USB device's functionality is divided into a series of class codes which are sent to the USB host to allow loading of software to support new devices. Among these device classes we find audio, image, human interface device and CDC. CDC stands for Communications Device Class and is used for computer networking such as modem and WiFi adapters, and serial protocols such as RS-232.

ACM is a subclass of CDC and stands for Abstract Control Model. The subclass describes the bidirectional communication flow known as Virtual Serial Port which emulates a serial port, and is a vendor-independent publicly documented protocol. The class contains two interfaces which are:

- **COM:** This interface, component object model, contains one endpoint at the host which is to notify the host about the serial state
- **DATA:** This interface contains two endpoint to enable sending and receiving of bulk data

Chapter 3

Development tools and hardware

3.1 Developement tools and Software

During this thesis a broad variety of software and integrated development environments (IDE) have been used for making implementations of all the previous mentioned systems able to run on embedded devices and to debug and investigate behaviours and errors. I will cover two of them and list the rest below. All source code have during this thesis been written in C except for some utility tools made in Python 3.7, covered in chapter 7.

SEGGER Embedded Studios

SEGGER Embedded Studios (SES) is a powerful and easy-to-use IDE with different tools for project editing and debugging for ARM Cortex devices. It is free for all users of Nordic Semiconductor's Software Development Kit. Different compilers for embedded development are also included and there is support for cross-platform development for both Windows, macOS and Linux.

Atmel Studio 7

The integrated development environment used to modify the robot software has been Atmel Studio 7 which is a suite built on Microsoft's Visual Studio. It is mainly for developing and debugging AVR and SAM microcontollers. It features broad support for the Atmel-ICE debugger used to debug the robots CPU. It does also come with a vast code library including drivers and example.

Other software used

- nRF Connect - A graphical tool for flash programming nRF SoC
- SEGGER RTT - Collection of software for Real-Time Transfer of data between host and nRF SoC
- PuTTY - Terminal used for CLI, UART and USB debugging
- Termite - Terminal used for UART debugging
- PyCharm - Python Project IDE to make scripts to gather data and results.
- Wireshark - Network packet capture software. Covered in chapter 7
- Thread Topology Monitor - Interface to monitor Thread network. Covered in 7.1
- Git - Version Control System used on all software developed in this thesis

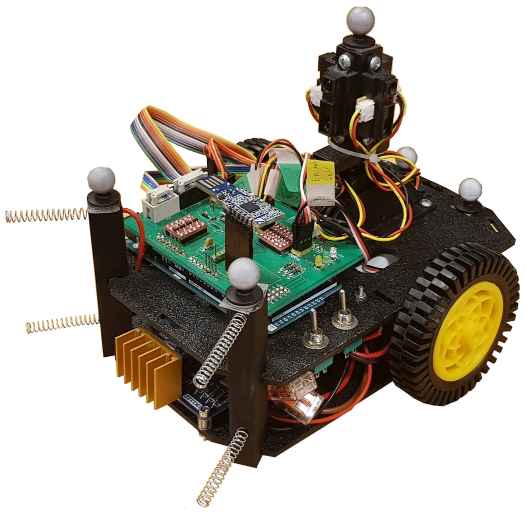


Figure 3.1: The robot

3.2 Robot overview

Figure 3.1 shows how the robot looks. The only change made during the work on this thesis was to replace the blue dongle approximately in the middle of the picture with another dongle which looks very similar. The printed circuit board (PCB) located on the top is designed by Nilsen [16] in 2018 and has connectors for external devices such as IR sensors on the sensor tower, the motor controller and the encoders. Beneath the PCB, a Arduino Mega is located and this board contains a ATmega2560 chip from Microchip and is the main processor of the robot. More on this device in section 3.3. The blue dongle located above the PCB, as mentioned above, is a nRF51 dongle designed and developed by Nordic Semiconductor. This handles communication with a server or host. Device specification and information is presented in section 3.4.

3.3 ATmega2560

The robot's main processor is an ATmega2560 microcontroller embedded on a PCB called Arduino Mega. This microcontroller is responsible for handling sensor data and move the robot to the desired location by using the motors. The PCB exploits the microcontroller's features and GPIOs, making it easier to work and develop with.

The microcontroller features 54 general-purpose input/outputs (GPIO) on the PCB where 16 is dedicated to analog inputs, 4 to UART hardware serial ports and more. It does also have 256 kB flash memory and 8 kB RAM.

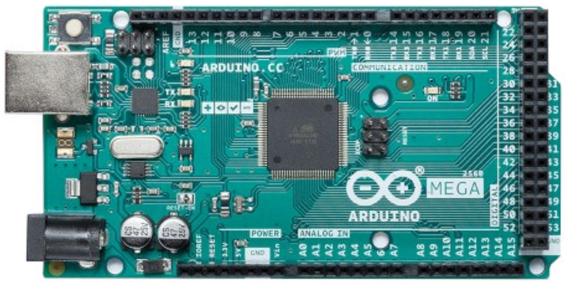


Figure 3.2: Arduino Mega with ATmega2560

3.4 Nordic Semiconductor SoC

The presented SoC here is the ones used in this thesis. Table 3.1 shows a technical comparison between the SoCs.

3.4.1 nRF51422 Dongle

The device responsible for the communication between the robot and the server were initially the nRF51422 dongle which is a product of Nordic Semiconductor. It is a versatile and complete development platform for BLE, ANT and 2.4 GHz proprietary applications. It supports development for the entire nRF51 SoC series, and contains programmable GPIOs, LEDs and buttons. It features an onboard SEGGER J-link debugger which enables debugging and programming directly with a USB connection through either Keil or SES.

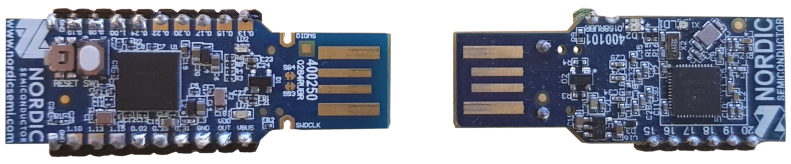


Figure 3.3: nRF52840 and nRF51 dongle

3.4.2 nRF52840 Dongle

The nRF52840 dongle is a small and low-cost USB dongle similar to the nRF51422 dongle. It comes with nRF52840 SoC which enables both BLE, ANT, Bluetooth Mesh, Thread, Zigbee, 802.15.4 and 2.4 GHz proprietary applications. It does not feature a onboard debugger, but instead it can be programmed easily with nRF Connect software provided by Nordic Semiconductor. This requires the software for the dongle to be developed and debugged on a nRF52840 development kit which has an onboard debugger.

3.4.3 nRF52840 Development Kit

The nRF52840 Develepment Kit (DK) has the same SoC as the dongle described in 3.4.2, but features a greater interface for exposing the SoC's features. It does also come with more buttons and leds for user interactions, USB connector, UART interface through a virtual COM port, but most importantly an onboard SEGGER J-link OB debugger with debug out functionality to enable debugging of custom solutions with the same SoC onboard.

	nRF51422	nRF52832	nRF52840
Processor	32-bit ARM Cortex-M0 16 MHz	32-bit ARM Cortex-M4 64 MHz	32-bit ARM Cortex-M4F 64MHz
RAM	16 KB / 32 KB	32 / 64 KB	256 KB
Flash	128 / 256 KB	256 / 512 KB	1 MB
TX Power	4 dBm	4 dBm	8 dBm
RX Sensitivity	-93 dBm @ 1Mbs - BLE	-96 dBm @ 1Mbs - BLE	-96 dBm @ 1Mbs - Bluetooth 5 -100 dBm @ 250kbs - 802.15.4
GPIO	31	32	48
On-Air data rate	2Mbs/1Mbs/250kbs	2Mbs/1Mbs - BLE 1Mbs - ANT 2Mbs/1Mbs - 2.4GHz proprietary	2Mbs/1Mbs/500kbs/125kbs - BLE 250kbs - 802.15.4 2Mbs/1Mbs - 2.4GHz proprietary
Serial interfaces	2 x I2C 1 x UART CTS/RTS	2 x I2C Master/slave 3 x SPI master/slave 1 x UART CTS/RTS 1 x I2S 1 x PDM 1 x QDEC	2 x I2C Master/slave 4 x SPI master/slave 1 x QSPI 2 x UART CTS/RTS 1 x I2S 1 x PDM 1 x QDEC

Table 3.1: Specification comparison between the nRF SoC used in this thesis

Chapter 4

Robot Application

4.1 How it works

The robot application as of january 2019 runs on a Arduino Mega ATmega2560 microcontroller featured the hardware section 3.3. As it only contains one core, it lacks ability to run complex operating systems with multi-threading abilities. However, by introducing tasks and a scheduler this can be solved to enable a fair share of the computing power when needed. In the robot's case, it is running in all seven tasks with different level of priorities.

To handle these tasks, the application uses FreeRTOS which is a free real-time operating system that are both well documented and well-proven. One of its main features is that it ships in a package small enough to run on microcontrollers. The kernel itself requires between 5 to 10 kB of flash memory depending on compiler and configurations. The scheduler it self 236 bytes, each queue 76 bytes and each task 64 bytes of RAM. All this enables core real-time scheduling, inter-task communication, timing and synchronization primitives.

A brief overview of the seven tasks with their priorities:

- **vMainCommunicationTask**, Priority **3**

  This task is the responsible for handling the variety of messages received from the server which can be either ping response, setpoint order, pause/unpause or start/stop. It is also the task that initiates the connection and handshake with the server. It continuously checks for a semaphore signalling a new message is ready for handling.

- **vMainPoseEstimatorTask**, Priority **4**

  This is an independent task which runs every 40 milliseconds and uses ticks from the motor encoders and sensor data to estimate its position by using a Kalman filter.
- **vMainControllerTask**, Priority **3**

  By using the values provided by the estimator, the controller task is to get the robot to the desired position initially given by the server. It also uses a PI-controller for turns. It is synchronized to run immediately after the estimator is finished.
- **vMainSensorTowerTask**, Priority **2**

  The sensor tower task turns the tower one servo step and reads the IR sensors, in best case, every 200 milliseconds. This servo step rate depends on how the robot moves. If it turns, the tower will not rotate and if it drives forward, it will rotate five times faster. It is also responsible to send updates to the server.
- **vARQTask**, Priority **3**

  This task can be seen as a sub-task of vMainCommunicationTask as its created by it. It is responsible to check for outgoing messages if they have been acknowledged or require retransmission. Scheduled to run every 10 milliseconds.
- **vUartSendTask**, Priority **4**

  Short, but important task which is responsible for writing messages going to to the nRF51 dongle and further onto Bluetooth to the server. Checks for a shared buffer to be non-empty.
- **vFrameReaderTask**, Priority **4**

  Almost the oppsite as vUartSendTask is this task receiving bytes over USART once an interrupt is triggered. It then decodes the message and hands it to vMainCommunicationTask.

FreeRTOS is configured to use preemption, which means that a task with higher priority can temporarily interrupt and block a lower priority task. As we see from the priorities given above, it is important that neither the estimator nor the USART tasks can be blocked.

4.2 Communication stack

The robot software also contains features such as multiple driver interfaces and functions for various functionality and utilities. However, the functionality that is most interesting for this thesis is the communication stack. Besides the tasks responsible for it's part of communication flow, we have five layers present which are the application, server_communication, protocol which is either ARQ or ”simple protocol”, network and USART. In addition to these, we must also take the nRF51 dongle into consideration as it is also a part of the system externally.

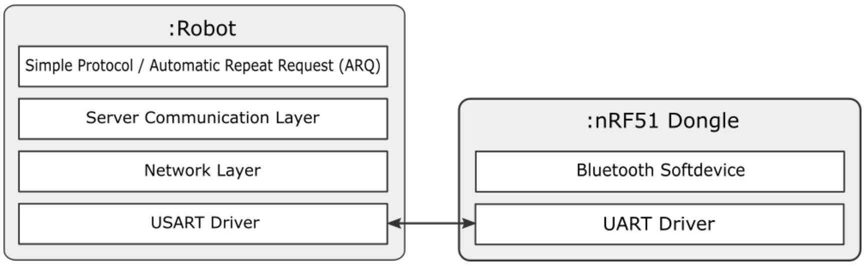


Figure 4.1: The robot communication stack

Figure 4.1 shows the stack overview and shows which modules that communicate with which externally. The Bluetooth Softdevice is responsible for handling the radio and Bluetooth Protocol.

The server communication layer is the interface for the rest of the application to initialize the rest of the communication and send messages. The ARQ, or Automatic Repeat Request, protocol was implemented by Lien[2] to enhance the system and to address errors. It features SAR functionality and retransmissions of messages based on timeout intervals. The ”simple protocol” is a protocol which do a single shot message sending without the need of acknowledgment. The network layer is responsible to add destination and source to the message, as well as a protocol flag. After doing that, it does a circular redundancy check (CRC) which adds a number of bytes to the end of the message so that the receiver can check the message integrity. Then it encodes the message with Consistent Overhead Byte Stuffing (COBS) which transforms the message in a defined way such that reserved values no longer occur. The reasoning behind this is because the null value 0x00 is considered a delimiter on UART transfers, meaning that when the UART driver receives a 0x00 value, it knows that is the end of the message and processes it accordingly.

4.3 Changes made

Note that all changes made described here is done independently of the communication systems being described in the next chapters and their relevant changes.

4.3.1 Debugging abilities

The debugging ability on the robot was minimal and was implemented by enabling a global variable called DEBUG. That enabled the application to send custom messages via Bluetooth to the server also running in debug mode for thereby print these in the desired IDE where the server application ran or at the host computer. That is a very inconvenient approach since it takes up many resources for the ATMega2560 as it has to initialize the message and send it through the ARQ, network and, USART tasks, not to mention that it's a useless way to use while debugging a new communication system.

Instead of using this method, I initially included functionality for serial printing on another UART interface by using *printf()*. However, after a couple months in, the *printf()*, while implementing the new communication interface, I noticed that it ran out of memory when handling many messages, so it seemed better to avoid this method. After a while, I figured out how to debug the software through Atmel Studio. There were apparently a fuse, or the chip's configuration parameter, that was not set properly. Although programming this fuse had to be done multiple times, but it worked flawlessly.

4.3.2 Calibrating and rework IR sensors

When I did initial testing of the robot early in the thesis phase, I quickly noticed that the IR sensor data was not very reliable nor trustworthy. To fix this problem I found a ruler of one meter which I placed on a table with the robot center at 0, and then I read the analog values every 5 cm from 80 cm till 10. 80 cm is the threshold of the sensors, and 10 cm is practically on the robot. The sensors showed a significant variance and fluctuation between greater distances, which makes sense as the IR radiation forms a cone the longer it travels. Though, after sampling all the sensors a sufficient amount, I plotted the values in Excel and found a function for each sensor rather than having a look-up table as they used before. That showed great results which can be seen later in figure 8.14 in chapter 8.3 on page 66.

Chapter 5

Bluetooth Mesh Communication System

5.1 Method

5.1.1 Design choices

I started out early on by using the application developed in the specialization project[1], but as the state of Nordic Semiconductors' implementation of BLE Mesh had developed drastically since the start of the project in the autumn 2018 I decided to start fresh with the existing knowledge I had earned through the project. The Nordic Semiconductor general SDK had been updated to 15.2.0 and its Mesh SDK to 3.1.0 from 1.0.1 which was used previously eliminating backward compability, thus more specification compliant. The Mesh SDK version 1.0.1 also used parts from SDK 14.2.0 and the previous project some from 12.3.0 which made it difficult to follow and complex.

By using the theory in section 2.3.2 and the knowledge on how the robot works from chapter 4 we can derive and choose a design for the mesh network regarding the different models, states and way of communication. As the robot sends updates to the server with sensor information, position and angles, we would need a model to suit the same purpose. The robot does also incorporates pinging, but that is irrelevant as the Bluetooth mesh features health models mentioned at the end of section 2.3.2.

In addition to the communication done by the robot, a server or a host is also responsible for a part of the communication flow. This involves sending position commands to the robots which in turn sets a new target position to move to. For this, only a x and y value given in centimetres is provided. It does also decide when the robots can start, stop, pause and unpause. For the latter, Nordic Semiconductor already have a generic Bluetooth SIG model which contains a boolean state which is either on or off.

The following figure (5.1) shows an illustration of how we want to design the mesh network to support the robot/server communication.

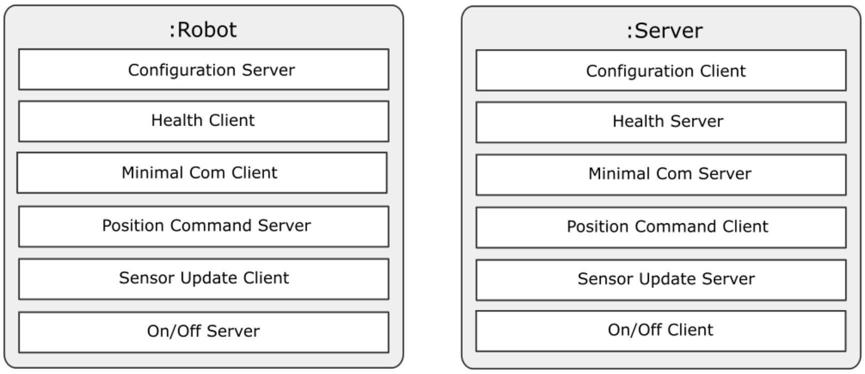


Figure 5.1: The robot

5.1.2 Driver support and implementation

nRF52840 Dongle

Due to limitations of the nRF51422 chip on the nRF51 dongle, the nRF52 dongle was chosen for this project. Bluetooth is developing fast and in order to keep up, it was crucial to change the device. Along with the fact that Bluetooth mesh no longer is supported on the nRF51 dongle after SDK version 1.0.1.

Enabling and using UART

As table 3.1 shows, the nRF52840 chip features two UARTs. One of them being a traditional UART while the other being a UARTE, which stands for Universal Asynchronous Receiver/Transmitter with EasyDMA. EasyDMA is ”an easy-to-use memory access module that some peripheral implement to gain direct access to Data RAM” [17]. EasyDMA is not featured on the nRF51422 chip, and only accessible on nRF52. It can also be said to be automated data transfer between memory and peripherals.

The first UART, UART0, can be used with EasyDMA as well, however the second UART, UART1, can only be used with EasyDMA. So the notions are UART0, UARTE0 and UARTE1. Also, UART0 can not be used with UARTE0. Only one instance of UART0 can be active, either with or without EasyDMA.

The UART handler was set up to receive data when a UART event occurs. It then loads each byte into a static buffer until there are either no more bytes to be received or if the byte corresponds to a COBS delimiter. In the latter, it will get sent to the server model immidiately. If an error occurs on the UART event handler during receiving, the loaded message will be discarded as well as the rest of it as it is corrupt.

USB CDC ACM driver

I decided to implement a USB CDC ACM interface from the development kit, or dongle, to a host or server. This removes the need of UART on the server side on the computer to make it work with the old java server if needed in the future. Whats important to know about is that the write call is asynchronous, so we cannot use a local defined array of values to be sent. Instead, a global buffer was used and the array got copied into the buffer every time the write call was used.

5.2 Implementation

5.2.1 Creating the models

Initially, it was decided to make a model which contains no states but only has the ability to send an arbitrary array of 1 byte sized integers. I call it a minimal communication model and the reason behind the choice was to have a possibility to send whatever desirable to make it easier to debug, check features and ensuring the procedure with setting up publication status, subscriptions works properly.

For creating a complete model, we need both a client model and a server model. The client shall have the ability to send either a reliable message or an unacknowledged set message, as well as a status get message. The corresponding server is then in need of handling incoming messages and send status message in response to both a reliable message and a get status message. Then in turn, the client also have to handle the incoming status message as a reply. As there is a possibility that the server does not respond, the client model also need to handle a timeout.

I started out building the minimal communication model out of the ”simple_on_off” model made by Nordic Semiconductor[18] which can be found in their SDK for Mesh version 3.1.0.

The client type was declared as follows:

```c
typedef struct {
    access_model_handle_t model_handle;
    min_com_status_cb_t status_cb;
    min_com_timeout_cb_t timeout_cb;
    struct
    {
        bool reliable_transfer_active;
        uint8_t * data;
    } state;
} min_com_client_t;
```

The model_handle variable is used by the various functions in the access layer to do modifications to the model. The status_cb is a callback function for status messages received by the server model, the timeout_cb for when the server does not respond and the state struct is for holding the state which in this case is a integer array.

The corresponding server model were set up like this:

```c
typedef struct {
    access_model_handle_t model_handle;
    min_com_get_cb_t get_cb;
    min_com_set_cb_t set_cb;
} min_com_server_t;
```

Where the handle is working the same way as for the client model and the two callbacks functions are for the get and set request respectively.

The client model id was set to 0x0004 and the server model to 0x0005 to not to be confused with the foundation models or the simple_on_off models. The company id was set to be Nordic Semiconductor’s, 0x59, as the application runs on their devices and since company ”None” is reserved for generic Bluetooth SIG models. The opcodes for the minimal communication model is shown in table 5.1

Opcode name	Identifier (Hex)
MIN_COM_OPCODE_SET	0xC5
MIN_COM_OPCODE_GET	0xC6
MIN_COM_OPCODE_SET_UNRELIABLE	0xC7
MIN_COM_OPCODE_STATUS	0xC8

Table 5.1: Operation codes for minimal communication models

The reason for the choice of opcodes is because the Mesh Profile Specification tells that 3-octet opcodes should be used by manufacturer-specific opcodes where the first two bits have to be 1 and 1, making 0xC1 the first valid opcodes outside of Bluetooth SIG's own defined application opcodes. However, since Nordic Semiconductor already has a model with 0xC1 to 0xC4, I found it is best practice to choose something different to avoid the same opcodes being used.

The sensor update client was given id 0x0006 and the server 0x0007. The position command client 0x0008 and its corresponding server 0x0009. Table 5.2 shows their opcodes which is following along the ones of the minimal communication models in 5.1

Opcode name	Identifier (Hex)
SENSOR_UPDATE_OPCODE_SET	0xC9
SENSOR_UPDATE_OPCODE_GET	0xCA
SENSOR_UPDATE_OPCODE_SET_UNRELIABLE	0xCB
SENSOR_UPDATE_OPCODE_STATUS	0xCC
POS_CMD_OPCODE_SET	0xCD
POS_CMD_OPCODE_GET	0xCE
POS_CMD_OPCODE_SET_UNRELIABLE	0xCF
POS_CMD_OPCODE_STATUS	0xD0

Table 5.2: Operation codes for the sensor update models and position command models

After the models were complete, and the application relevant callbacks implemented, the instances had to be initialized in each application they are to run in. By that the API call *access_model_add* are used and which element it should be added to have to be specified. I chose to have all models on the single element as there is no need for two as all have unique model ids.

The last, crucial step was to configure the provisioner and configuration client to set up the robot applications correctly once they are connected. As described in 2.3.2, this involves binding the application key and to set both subscription and publish states and addresses for each model.

The process itself takes place in a separate node setup interface which is application dependent and is gone through by the provisioner when it is configuring the devices.

The subscription address for the different models was set to be a group address. For the server development kit, the minimal communication server model subscribes to 0xC002, the sensor update server model subscribes to the address 0xC003, and the position command server model to 0xC004. All three addresses are specification compliant to be a group address and the reason behind this choice is that all robots can publish to that group address, the server handles it and replies directly to the robot's address. That is a better solution in terms of scalability to avoid the server to add new subscription addresses for each connected device.

Chapter 6

OpenThread Communication System

6.1 Method

6.1.1 MQTT broker

The MQTT broker is a Mosquitto 1.5.3 broker which is released by Eclipse Foundation. The main reasons behind this choice were that it is an open source implementation of a broker for both the 5.0 and 3.1.1 version of the MQTT protocol and it ships with various utilities for easy debugging and logging.

The broker is deployed on a virtual server in Amsterdam provided by DigitalOcean. The advantage of doing this is that it stays online and easy to access with a static IP address all day, making it more accessible for both me and the other students that need the MQTT broker to work within their projects. However, the location may contribute to some latency which after a pinging session was found to be 34 ms at max.

6.1.2 Border router

The border router connects a thread network to other IP-based networks such as WiFi and Ethernet. As OpenThread is supported with Nordic Semiconductor devices running on the nRF52840 SoC, Nordic Semiconductor has provided a disc image file containing Raspian for use on a Raspberry Pi B+ version 3. Raspbian is a lightweight Linux Debian distribution optimized for Raspberry Pi. The dongle or development kit with the nRF52840 SoC is then connected to the Raspberry Pi, and it works as a connectivity chip running a Network Co-Processor. That allows for the OpenThread application on the SoC to communicate with network daemons on Unix-like operating systems through a protocol called Spinel [19].

The network co-processor hex from Nordic Semiconductor was flashed to a nRF52840 dongle and connected to a Raspberry Pi. After boot, the Border router was operable. Along on the disc image, there was also a MQTT-SN gateway made by Eclipse, called Paho MQTT-SN Gateway.[20]. The only necessity to set it up correct was to update the configuration file to contain the MQTT broker IP address with desired credentials.

6.1.3 Rebuilding OpenThread libraries

As the Thread SDK from Nordic Semiconductor runs static pre-compiled libraries from OpenThread, the need for changing system-wide variables and pre-processor defines has to be done by changing the OpenThread stack. Fortunately, OpenThread is open-source, as stated earlier. As OpenThread is built in a specific Linux environment and the static libraries being build from a specific commit, the easiest way was to deploy a Docker container and to clone the entire repository of the OpenThread source.

As OpenThread supports nRF52840 and vice versa, it should be straight forward to change the variables, rebuild the libraries, and copy them to the SDK for further use within the OpenThread application being developed in SEGGER. However, the static libraries outputted from the building turned out quite different from the ones in the SDK on beforehand. Though it enabled changes on the OpenThread source code.

6.2 Implementation

6.2.1 Setting up publisher and subscriber features

As MQTT is known from beforehand, a feasible approach to communicate between the robot and the workstation, as well as in between the robots is to have an publish address equal to the robots identity. It was choosen to implement the topics like below:

```
ntnu-slam\robot\<robot-id>\<type>
```

and similarly for the server:

```
ntnu-slam\server\<robot-id>\<type>
```

Whats inside the greater-than and less-than signs are changing variables depending on which device id and the message type. ”ntnu-slam” is a localization specifier, but could be called anything else, but only for practical reasons if the gateway should be connected to another broker which serves multiple applications to avoid collisions.

If the server is to subscribe to multiple networks and robot IDs, the following topic setup would satisfy to collect all messages sent on the system:

```
ntnu-slam\robot\#
```

Where both + and # are wildcards as described in 2.6.

In addition to these topics, a will topic is also registered which is a brilliant way to tell the server that a client device is disconnected. Since the server can subscribe to all activity on the broker and network, it could also detect a defined will message such as ”Disconnected” on an arbitrary topic.

In order to register to custom topics, dependent on the client id, a set of pre- and postfixes were set up to make it easy to add and remove topics as needed. The topic structure mqttsn_topic_t were used as the name is needed for registration and subscription, while only the topic id which is given by the gateway is used for publishing.

```c
typedef struct mqttsn_topic_t
{
    const uint8_t * p_topic_name;
    uint16_t        topic_id;
} mqttsn_topic_t;

static struct mqttsn_topic_t m_topics[NUM_TOPICS];

static char*        postfixes[] = {"/adv", "/update", "/cmd"};
static char*        prefixes[] = {"ntnu-slam/robot/", "ntnu-slam/robot/",
                                  "ntnu-slam/server/"};
```

The client id is configured to be randomly generated once a robot starts up as it should be unique. However, it is possible to store a client id once in persistent memory to load it on start up. Random id was a great feature when changes were done to the code every time it ran. The client id is embedded into the topic name between the pre and postfixes.

The publishing configurations are set to QoS 1 and a retransmission interval which is 8 seconds at default. No retain or dup are set, only will topic and message.

6.2.2 UART communication with robot

OpenThread ships with an interface for using UART and a command line interface. The command line interface is commonly used to set configuration parameters, read data and test connectivity while being connected to a computer. Thus, making new commands is fairly complicated and I was not able to do any changes to this interface. Instead I decided to remove the entire interface as the devices were quite easy to debug using J-link and SES's debug interface. As the command line interface also used one of the main UARTs I thought it was better to remove the interface and using UART as a way to send byte commands to the robot.

The first approach was to use the OpenThread implementation UART0, which included an API to use in the application. It was quite easy to configure, but after successfully sending a few bytes, I was not able to receive them. Even though the interrupt service routine was set for the UART0 RX register, nothing happened once bytes were sent. The CLI was then re-instituted to the application which later proved to be a brilliant decision.

The second approach was to implement UARTE1 alongside OpenThread's implementation of UART0. After a while, configuring the local SDK configuration file, it got working flawlessly. I also got the pin mapping of the UARTE1 to be the same as for the dongle that were already on the robot for simplicity. While moving the pin mapping I noticed a bug in the SDK, which took some time to find. The RX pin was not pulled high when the UARTE1 was initialized, which lead to a range of framing errors and assertions. A quick look in the register reference, the problem was solved.

The UARTE1 handler works by a interrupt being made from the app UART module with an event. If the event is that there are data in the RX register, the bytes are loaded into a fifo queue. The reasoning behind this is to have a circular buffer so that the first bytes inserted are the first to be removed.

```c
void uarte_receive_and_send(app_fifo_t * fifo,
                            uint32_t (*func)(uint8_t*, uint16_t)){
  uint32_t err_code;
  uint32_t index = 0;
  uint8_t data;
  uint16_t idx = 0;
  uint32_t fifo_length;

  app_fifo_read(fifo, NULL, &fifo_length);
  uint32_t remaining = fifo_length;
  uint8_t array[fifo_length];

  while(remaining > 0){
    app_fifo_peek(fifo, idx++, &data);
    --remaining;
    array[index++] = data;

    if(index == UART_RX_BUF_SIZE || data == COBS_DELIMITER){
        err_code = func(array, index);
        if(err_code == NRF_ERROR_NULL || err_code == NRF_ERROR_FORBIDDEN){
            break;
        }
        else if(err_code != NRF_ERROR_INVALID_STATE){
            app_fifo_read(fifo, array, &index);
            idx=0;
            APP_ERROR_CHECK(err_code);
        }
        index = 0;
    }
  }
}
```

The code above shows how a function for both sending to MQTT, UART, or any custom function calls which receives a buffer and its length. It reads the fifo until a delimiter is reach, then it pass the values to the desired function and removes the bytes from the fifo. This approach also makes the nRF52840 application to buffer at least 512 bytes of data before it can be published if the network connection fails for a while. That equals at a minimum 25 messages from the robot. A side note, the same approach was tried on the BLE Mesh implementation, but it appears to be some race conditions such that the fifo never filled up as supposed to.

In addition, in order for the robot to start operation, an interrupt on pin 2 on the Arduino board is necessary to be detected. To achieve that, once a gateway was found and it was successfully connected, one defined pin on the development kit was cleared. By doing this, we would avoid the robot sending messages before the network is ready, to avoid the fifo to be filled.

6.3 Changes on the robot

As a part of the scope of this thesis, one task was to make a more compact and easier way for the robot application to communicate with a server. By enabling this, I removed everything regarding communication on the robot to build another interface to the MQTT application running on the nRF52840 device. Also, I made it suitable to the new server made by my fellow student Torstein Grindvik; which I will show and mention in section 7.4.

The main difference with his server from the java server is that it receives messages by MQTT rather than Bluetooth. As well, instead of receiving all sensor data, angles and position of the robot, it only receives the position and where an obstacle is located relative to the robots position. For this we had to declare a common data structure for both the robot and the server.

```c
typedef struct {
  int16_t x;
  int16_t y;
} __attribute__((packed)) position_t;

typedef struct {
  position_t pos;
  position_t obstacles[4];
  uint8_t len;
} __attribute__((packed)) update_mqtt_message_t;
```

Since there are not necessarily four obstacles in either directions, so we have to figure out in which position are relative to the robot. This was done by converting the sensor data and angles to a point using basic geometry.

```c
void send_update_mqtt(int16_t xhat, int16_t yhat, int16_t heading_deg,
    int16_t towerAngle_deg, uint8_t * sensor_data){
  update_mqtt_message_t message;
  message.pos.x = xhat;
  message.pos.y = yhat;
  uint8_t ob_counter = 0;

  for(uint8_t i = 0; i < 4; ++i){
    position_t ob;
    if(sensor_data[i] > 13){
      ob = (position_t){
                  xhat+(sensor_data[i] *sin(heading_deg*DEG2RAD+
                  towerAngle_deg*DEG2RAD+
                  90*i*DEG2RAD))
                              ,
                  yhat+(sensor_data[i] * cos(heading_deg*DEG2RAD+
                  towerAngle_deg*DEG2RAD+
                  90*i*DEG2RAD))
                  };
      message.obstacles[ob_counter] = ob;
      ob_counter++;
    }
  }

  uint8_t data[sizeof(position_t)+sizeof(position_t)*ob_counter+1];
  memcpy(data, (uint8_t*) &message, sizeof(data));

  send_mqtt_message(data, sizeof(data));
}
```

The result can be a negative integer, so, therefore, the variables are set to be an integer and not an unsigned integer. At last the values, depending on the size, get sent to be COBS encoded, sent onto UART to the DK and further to the MQTT broker.

The resulting and new stack overview as described in figure 4.1 on page 33 is after the changed to as in figure 6.1

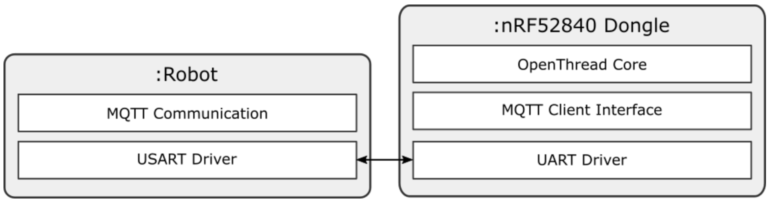


Figure 6.1: The new robot communication stack

Chapter 7

Test setup and experiments

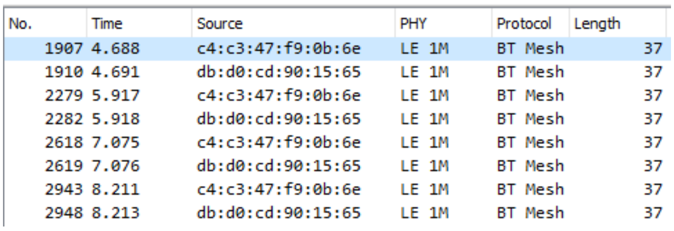
7.1 Thread Topology Monitor

Nordic Semiconductor offers a powerful tool for Thread network visualization, and that is called Thread Topology Monitor or TTM for short. That enables the user to see the network and its connected devices live, ping them and configure network parameters on the go. Figure 7.11 shows a set up. It proved well as it also shows the routes between the connect devices.

7.2 Data extraction for BLE Mesh

7.2.1 Wireshark

To get to capture BLE Mesh message ”mid-air” I've used Wireshark along with nRF Sniffer[21]. nRF sniffer is a tool for debugging BLE application and to get a real-time display of Bluetooth packets. Unfortunately, Wireshark does not have inbuilt support for decrypting BLE Mesh messages, so that is a limitation. However, Wireshark contributors commited a fix for this problem may 2019 [22].



No.	Time	Source	PHY	Protocol	Length
1907	4.688	c4:c3:47:f9:0b:6e	LE 1M	BT Mesh	37
1910	4.691	db:d0:cd:90:15:65	LE 1M	BT Mesh	37
2279	5.917	c4:c3:47:f9:0b:6e	LE 1M	BT Mesh	37
2282	5.918	db:d0:cd:90:15:65	LE 1M	BT Mesh	37
2618	7.075	c4:c3:47:f9:0b:6e	LE 1M	BT Mesh	37
2619	7.076	db:d0:cd:90:15:65	LE 1M	BT Mesh	37
2943	8.211	c4:c3:47:f9:0b:6e	LE 1M	BT Mesh	37
2948	8.213	db:d0:cd:90:15:65	LE 1M	BT Mesh	37

Figure 7.1: BLE Mesh Packet Captured

Though it is very helpful for checking routing and for debugging. Figure 7.1 shows what information we can extract from the sniffer. In this example, the source c4:c3:47:f9:0b:6e is a robot that sends an unacknowledged message though the minimal communication model to the server db:d0:cd:90:15:65. The reason also the server transmit is because it is relaying the message if there are other nodes to receive the message. A problem which I noticed is that there are being sent a crazy amount of Bluetooth messages in the environment of the workplace such that not all messages from the application will be catched though I can verify otherwise they have. Based on the samples from 7.1 an average of 295 Bluetooth messages were captured each second.

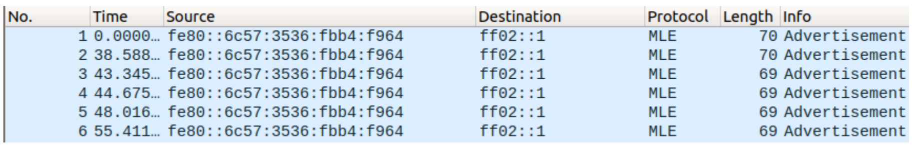
7.2.2 SEGGER RTT Viewer

SEGGER RTT is a graphical user interface which enables real-time data transfer between a host and a target application. It features a very high transfer speed and is supported by J-link and can do data logging of output sent by the target application. That enables a data transfer with timestamped messages to use later for performance analysis or simply debugging.

7.3 Wireshark with Thread and MQTT

In order to capture packets between the Thread devices, an external NCP device had to be set up and connected to an Ubuntu 17.04 or greater host running Wireshark. A NCP firmware was flashed onto a nRF52840-DK, and the neccessary permissions for Wireshark configured. Also, each protocol used, as in 2.4 had to be configured right and the network decryption key provided. The sniffer, or packet capturer, is provided by OpenThread and is called PySpinel [23].

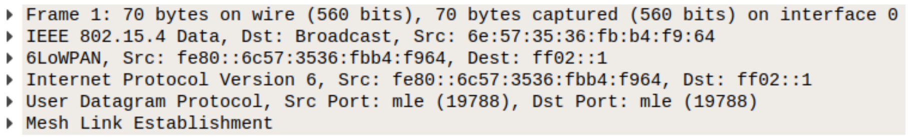
To start with, after starting the sniffer, the only messages we get are mesh link establishment advertisement packages as seen in 7.2. Recall from 2.5.1, the source is a link-local address, which belongs to the border router, and the destination address is a multi-cast address for all link-local FTD and MEDs.



No.	Time	Source	Destination	Protocol	Length	Info
1	0.0000…	fe80::6c57:3536:fbb4:f964	ff02::1	MLE	70	Advertisement
2	38.588…	fe80::6c57:3536:fbb4:f964	ff02::1	MLE	70	Advertisement
3	43.345…	fe80::6c57:3536:fbb4:f964	ff02::1	MLE	69	Advertisement
4	44.675…	fe80::6c57:3536:fbb4:f964	ff02::1	MLE	69	Advertisement
5	48.016…	fe80::6c57:3536:fbb4:f964	ff02::1	MLE	69	Advertisement
6	55.411…	fe80::6c57:3536:fbb4:f964	ff02::1	MLE	69	Advertisement

Figure 7.2: MLE Advertisement

We can also see the protocols derived from the message by analyzing the frame:



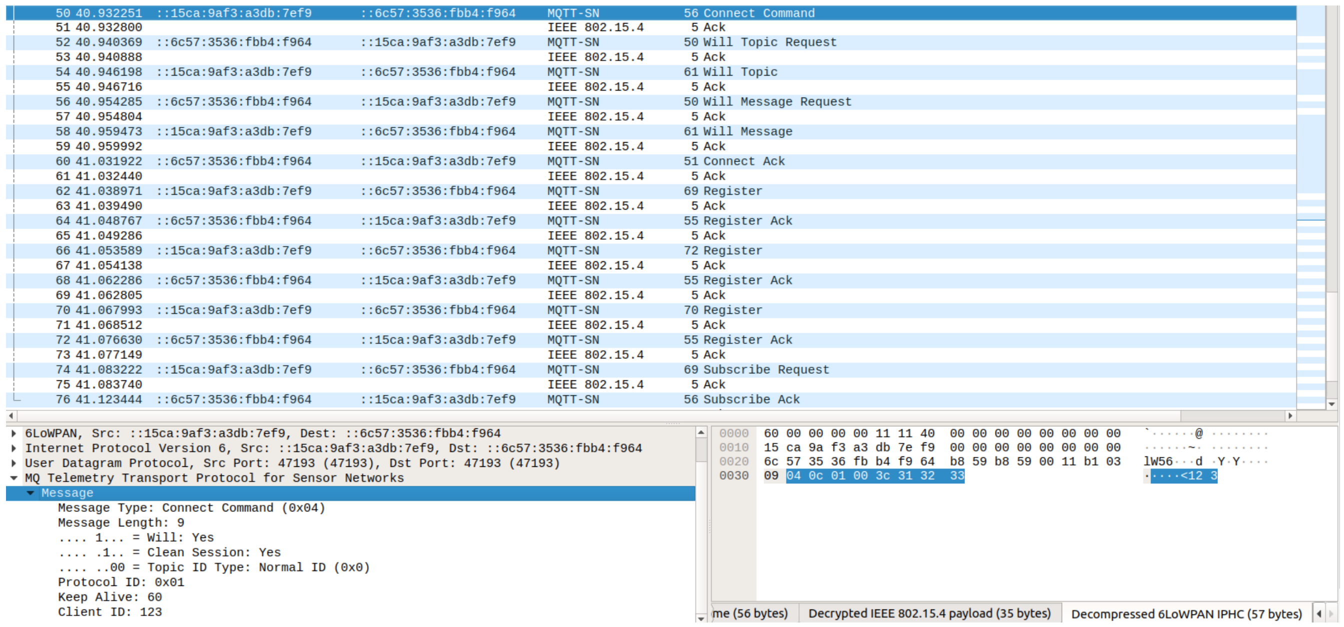
```
▸ Frame 1: 70 bytes on wire (560 bits), 70 bytes captured (560 bits) on interface 0
▸ IEEE 802.15.4 Data, Dst: Broadcast, Src: 6e:57:35:36:fb:b4:f9:64
▸ 6LoWPAN, Src: fe80::6c57:3536:fbb4:f964, Dest: ff02::1
▸ Internet Protocol Version 6, Src: fe80::6c57:3536:fbb4:f964, Dst: ff02::1
▸ User Datagram Protocol, Src Port: mle (19788), Dst Port: mle (19788)
▸ Mesh Link Establishment
```

Figure 7.3: Frame and protocol description

We can clearly see all the protocols used for the transmission. The physical layer represented by ”Frame 1”, describing all bytes and bits captured, the data link layer IEEE 802.15.4 which decodes the bytes to a frame with source, destination. The 6LoWPAN and IPv6 divides into packets and User Datagram Protocol (UDP) into segments. At the end, the MLE, contains the data for this transmission.

7.3.1 Gateway search and connection procedure

As from section 2.7.1 we had a brief look at how the gateway procedures are to work when a robot clients connects to the server and both register and subscribes to topics. Figure 7.4 shows all the data captured by Wireshark during this process between the nRF52840 dongle and the gateway located on the border router.



```
50 40.932251  ::15ca:9af3:a3db:7ef9   ::6c57:3536:fbb4:f964   MQTT-SN        56 Connect Command
51 40.932800                                                  IEEE 802.15.4   5 Ack
52 40.940369  ::6c57:3536:fbb4:f964   ::15ca:9af3:a3db:7ef9   MQTT-SN        50 Will Topic Request
53 40.940888                                                  IEEE 802.15.4   5 Ack
54 40.946198  ::15ca:9af3:a3db:7ef9   ::6c57:3536:fbb4:f964   MQTT-SN        61 Will Topic
55 40.946716                                                  IEEE 802.15.4   5 Ack
56 40.954285  ::6c57:3536:fbb4:f964   ::15ca:9af3:a3db:7ef9   MQTT-SN        50 Will Message Request
57 40.954804                                                  IEEE 802.15.4   5 Ack
58 40.959473  ::15ca:9af3:a3db:7ef9   ::6c57:3536:fbb4:f964   MQTT-SN        61 Will Message
59 40.959992                                                  IEEE 802.15.4   5 Ack
60 41.031922  ::6c57:3536:fbb4:f964   ::15ca:9af3:a3db:7ef9   MQTT-SN        51 Connect Ack
61 41.032440                                                  IEEE 802.15.4   5 Ack
62 41.038971  ::15ca:9af3:a3db:7ef9   ::6c57:3536:fbb4:f964   MQTT-SN        69 Register
63 41.039490                                                  IEEE 802.15.4   5 Ack
64 41.048767  ::6c57:3536:fbb4:f964   ::15ca:9af3:a3db:7ef9   MQTT-SN        55 Register Ack
65 41.049286                                                  IEEE 802.15.4   5 Ack
66 41.053589  ::15ca:9af3:a3db:7ef9   ::6c57:3536:fbb4:f964   MQTT-SN        72 Register
67 41.054138                                                  IEEE 802.15.4   5 Ack
68 41.062286  ::6c57:3536:fbb4:f964   ::15ca:9af3:a3db:7ef9   MQTT-SN        55 Register Ack
69 41.062805                                                  IEEE 802.15.4   5 Ack
70 41.067993  ::15ca:9af3:a3db:7ef9   ::6c57:3536:fbb4:f964   MQTT-SN        70 Register
71 41.068512                                                  IEEE 802.15.4   5 Ack
72 41.076630  ::6c57:3536:fbb4:f964   ::15ca:9af3:a3db:7ef9   MQTT-SN        55 Register Ack
73 41.077149                                                  IEEE 802.15.4   5 Ack
74 41.083222  ::15ca:9af3:a3db:7ef9   ::6c57:3536:fbb4:f964   MQTT-SN        69 Subscribe Request
75 41.083740                                                  IEEE 802.15.4   5 Ack
76 41.123444  ::6c57:3536:fbb4:f964   ::15ca:9af3:a3db:7ef9   MQTT-SN        56 Subscribe Ack
```

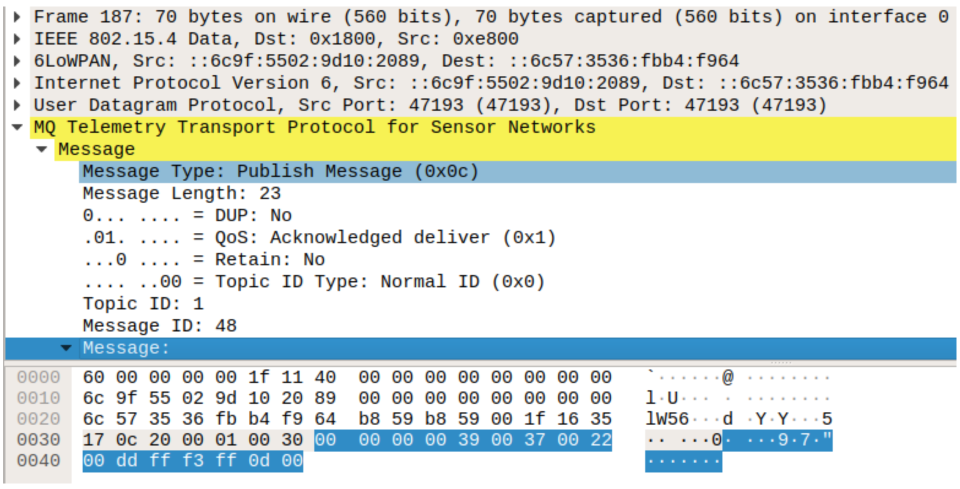
```
▸ 6LoWPAN, Src: ::15ca:9af3:a3db:7ef9, Dest: ::6c57:3536:fbb4:f964
▸ Internet Protocol Version 6, Src: ::15ca:9af3:a3db:7ef9, Dst: ::6c57:3536:fbb4:f964
▸ User Datagram Protocol, Src Port: 47193 (47193), Dst Port: 47193 (47193)
▾ MQ Telemetry Transport Protocol for Sensor Networks
  ▾ Message
      Message Type: Connect Command (0x04)
      Message Length: 9
      .... 1... = Will: Yes
      .... .1.. = Clean Session: Yes
      .... ..00 = Topic ID Type: Normal ID (0x0)
      Protocol ID: 0x01
      Keep Alive: 60
      Client ID: 123
```

```
0000  60 00 00 00 00 11 11 40  00 00 00 00 00 00 00 00   `......@ ........
0010  15 ca 9a f3 a3 db 7e f9  00 00 00 00 00 00 00 00   ......~. ........
0020  6c 57 35 36 fb b4 f9 64  b8 59 b8 59 00 11 b1 03   lW56...d .Y.Y....
0030  09 04 0c 01 00 3c 31 32  33                        .....<12 3
```

me (56 bytes)
Decrypted IEEE 802.15.4 payload (35 bytes)
Decompressed 6LoWPAN IPHC (57 bytes)

Figure 7.4: Connect and register procedure

7.3.2 Robot message publish example



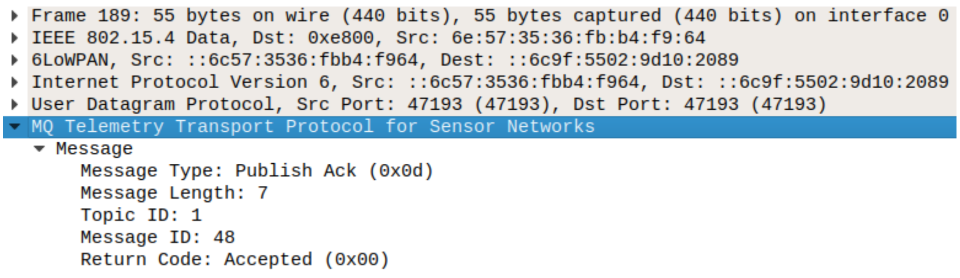
```
▸ Frame 187: 70 bytes on wire (560 bits), 70 bytes captured (560 bits) on interface 0
▸ IEEE 802.15.4 Data, Dst: 0x1800, Src: 0xe800
▸ 6LoWPAN, Src: ::6c9f:5502:9d10:2089, Dest: ::6c57:3536:fbb4:f964
▸ Internet Protocol Version 6, Src: ::6c9f:5502:9d10:2089, Dst: ::6c57:3536:fbb4:f964
▸ User Datagram Protocol, Src Port: 47193 (47193), Dst Port: 47193 (47193)
▾ MQ Telemetry Transport Protocol for Sensor Networks
  ▾ Message
      Message Type: Publish Message (0x0c)
      Message Length: 23
      0... .... = DUP: No
      .01. .... = QoS: Acknowledged deliver (0x1)
      ...0 .... = Retain: No
      .... ..00 = Topic ID Type: Normal ID (0x0)
      Topic ID: 1
      Message ID: 48
    ▾ Message:
0000  60 00 00 00 00 1f 11 40  00 00 00 00 00 00 00 00   `······@ ········
0010  6c 9f 55 02 9d 10 20 89  00 00 00 00 00 00 00 00   l·U··· · ········
0020  6c 57 35 36 fb b4 f9 64  b8 59 b8 59 00 1f 16 35   lW56···d ·Y·Y···5
0030  17 0c 20 00 01 00 30 00  00 00 00 39 00 37 00 22   ·· ···0· ···9·7·"
0040  00 dd ff f3 ff 0d 00                               ·······
```

Figure 7.5: Robot publish data

Figure 7.5 shows a captured package from the robot while running and scanning the area around. If we look at the MQTT message in yellow label, we can see its properties. In blue, further down, we can see the bytes of the message which is the decompressed 6LoWPAN IPv6 Header Compression. The bytes marked in blue at the lower end is the actual message payload sent. If we were to decode this to the structs defined in 6.2.1 we can represent the data in the following table where the bytes in the fourth row is in hexadecimal representation:

		Obstacles					
Robot position		1		2		3	
x	y	x	y	x	y	x	y
00 00	00 00	39 00	37 00	22 00	dd ff	f3 ff	0d 00
To decimal							
0	0	57	55	34	-22	-12	13

As we use QoS level 1, we would expect an publish acknowledgement which we get after 40 ms, shown in 7.6. Observe that the message ID and the topic id is the same as in the published message.



```
▸ Frame 189: 55 bytes on wire (440 bits), 55 bytes captured (440 bits) on interface 0
▸ IEEE 802.15.4 Data, Dst: 0xe800, Src: 6e:57:35:36:fb:b4:f9:64
▸ 6LoWPAN, Src: ::6c57:3536:fbb4:f964, Dest: ::6c9f:5502:9d10:2089
▸ Internet Protocol Version 6, Src: ::6c57:3536:fbb4:f964, Dst: ::6c9f:5502:9d10:2089
▸ User Datagram Protocol, Src Port: 47193 (47193), Dst Port: 47193 (47193)
▾ MQ Telemetry Transport Protocol for Sensor Networks
  ▾ Message
      Message Type: Publish Ack (0x0d)
      Message Length: 7
      Topic ID: 1
      Message ID: 48
      Return Code: Accepted (0x00)
```

Figure 7.6: Robot publish acknowledgement

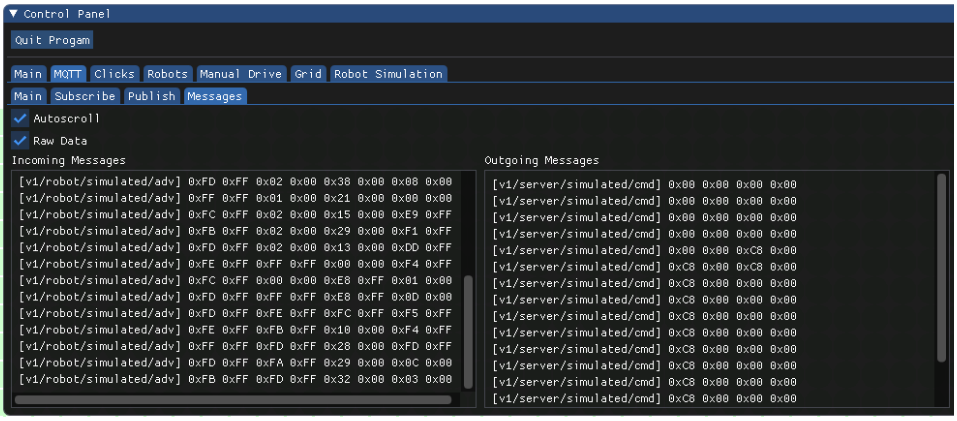
7.3.3 Wireshark Parser

A Python project to make a Wireshark parser was made. It supports both JSON and CSV file format. Which is JavaScript Object Notation and Comma-Separated Value format respectively. As Wireshark support various formats of desired data, it became a great source for investigation of both network behaviour and to debug. For example, timestamps and occurrences were captured and the data represented by use of graphs from Matplotlib which is a Python wrapper for Matlab-styled plotting features.

Most of the results presented in chapter 8 regarding the Thread network are made with this Python Wireshark Parser.

7.4 Robot visualization MQTT server

My fellow student Torstein Grindvik made during this thesis a server in C++ which subscribes to robot messages and plot the data as it get published. Figure 7.7 shows the main control panel of the server where data received on the desired topic is received. Figure 7.8 beneath shows the actual visualization where points are plotted based on the data received over the desired topic. This enables easier debugging and data visualization on how the new robot system works.



✓ Autoscroll

✓ Raw Data

Incoming Messages

```
[v1/robot/simulated/adv] 0xFD 0xFF 0x02 0x00 0x38 0x00 0x08 0x00
[v1/robot/simulated/adv] 0xFF 0xFF 0x01 0x00 0x21 0x00 0x00 0x00
[v1/robot/simulated/adv] 0xFC 0xFF 0x02 0x00 0x15 0x00 0xE9 0xFF
[v1/robot/simulated/adv] 0xFB 0xFF 0x02 0x00 0x29 0x00 0xF1 0xFF
[v1/robot/simulated/adv] 0xFD 0xFF 0x02 0x00 0x13 0x00 0xDD 0xFF
[v1/robot/simulated/adv] 0xFE 0xFF 0xFF 0xFF 0x00 0x00 0xF4 0xFF
[v1/robot/simulated/adv] 0xFC 0xFF 0x00 0x00 0xE8 0xFF 0x01 0x00
[v1/robot/simulated/adv] 0xFD 0xFF 0xFF 0xFF 0xE8 0xFF 0x0D 0x00
[v1/robot/simulated/adv] 0xFD 0xFF 0xFE 0xFF 0xFC 0xFF 0xF5 0xFF
[v1/robot/simulated/adv] 0xFE 0xFF 0xFB 0xFF 0x10 0x00 0xF4 0xFF
[v1/robot/simulated/adv] 0xFF 0xFF 0xFD 0xFF 0x28 0x00 0xFD 0xFF
[v1/robot/simulated/adv] 0xFD 0xFF 0xFA 0xFF 0x29 0x00 0x0C 0x00
[v1/robot/simulated/adv] 0xFB 0xFF 0xFD 0xFF 0x32 0x00 0x03 0x00
```

Outgoing Messages

```
[v1/server/simulated/cmd] 0x00 0x00 0x00 0x00
[v1/server/simulated/cmd] 0x00 0x00 0x00 0x00
[v1/server/simulated/cmd] 0x00 0x00 0x00 0x00
[v1/server/simulated/cmd] 0x00 0x00 0x00 0x00
[v1/server/simulated/cmd] 0x00 0x00 0xC8 0x00
[v1/server/simulated/cmd] 0xC8 0x00 0xC8 0x00
[v1/server/simulated/cmd] 0xC8 0x00 0x00 0x00
[v1/server/simulated/cmd] 0xC8 0x00 0x00 0x00
[v1/server/simulated/cmd] 0xC8 0x00 0x00 0x00
[v1/server/simulated/cmd] 0xC8 0x00 0x00 0x00
[v1/server/simulated/cmd] 0xC8 0x00 0x00 0x00
[v1/server/simulated/cmd] 0xC8 0x00 0x00 0x00
[v1/server/simulated/cmd] 0xC8 0x00 0x00 0x00
[v1/server/simulated/cmd] 0xC8 0x00 0x00 0x00
```

Figure 7.7: The MQTT based server control panel

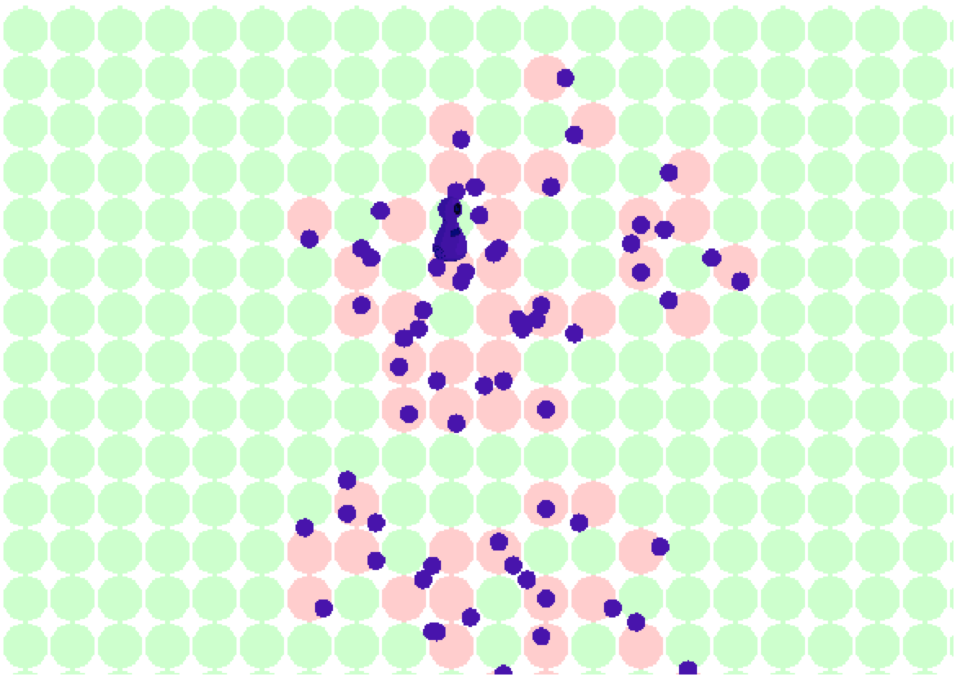


Figure 7.8: The MQTT based server visualization pane

7.5 Range and hopping mechanism

I did some tests regarding the core of mesh; the ability for message hopping and forwarding. To do this, I retrieved in total three development kits which I uploaded the provisioner/workstation server code onto one, and robot software on two. I then found a place where the only connection between one of the robot DK's and the workstation DK was through another robot DK. See figure 7.9 for placement. I then started a UART service equal to that of the robot which sent messages onto the robot DK every 200ms. The goal was to check both if it really worked and how well it worked based on packets received and lost.

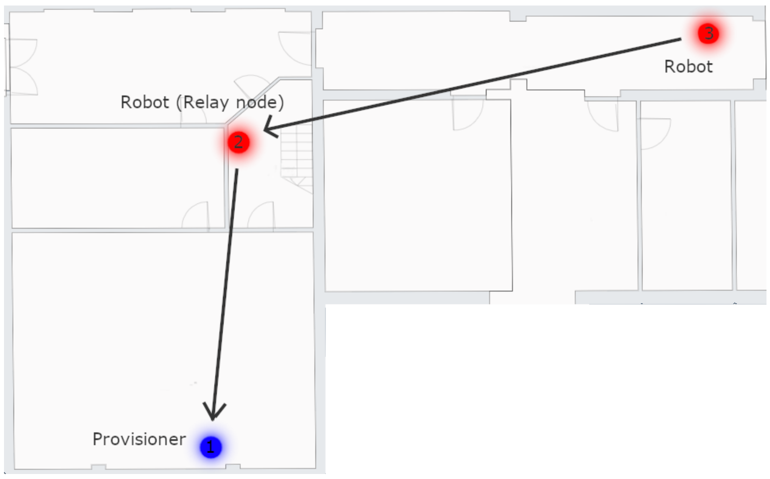


Figure 7.9: Room layout and DK locations

The lower-most blue dot symbolizes the position of the workstation server DK whereas the two red dots, number 2 and 3, are the robot DKs. The thick walls on the layout are made out of thick concrete and the thinner ones are presumably plaster.

I ensured that the middle robot DK point 2, had connection to the workstation server and the other robot DK, number 3, and that the rightmost robot DK, number 3, had no connection to the workstation server, blue, to ensure that all transmitted messages had to go through the other robot DK at point 2.

The same setup was also used in testing of the Thread network application's hopping mechanisms, but as an addition, Thread Topology Monitor and PySpinel with a thread sniffer were used at point two to get an overview on how the thread network were set up regarding roles and how the routing took place. That was a necessary tool to know whether all the messages would be routed through the topology monitor host or not.

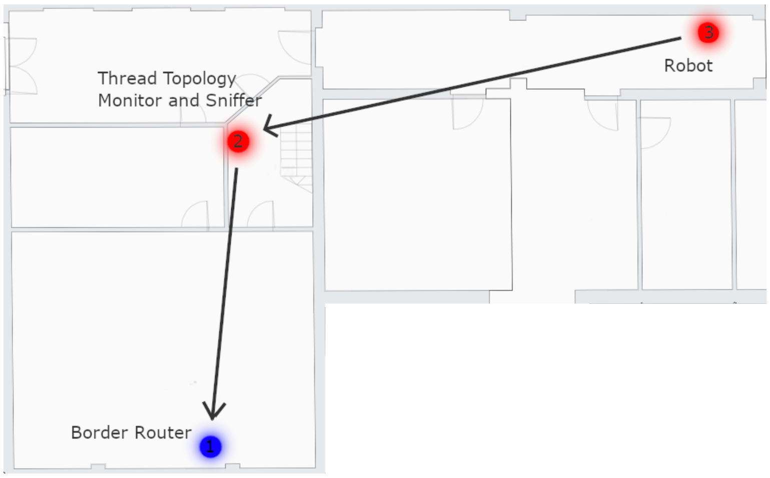


Figure 7.10: Room layout for thread hop and routing testing

Once set up in the locations shown in 7.10 TTM showed the following view of the network:

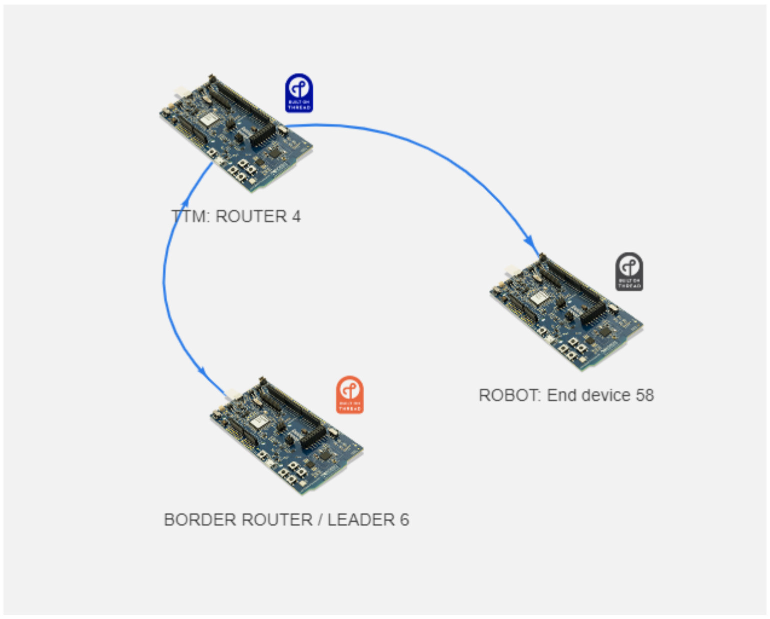


Figure 7.11: Thread Topology Monitor visualization of nodes at test location

What we see here is the devices connected. The names have been changed to illustrate better what is going on. The Border Router has the role as leader, the TTM as a router and the robot as an end device having only one connection, directly to the router.

7.6 Robot setup

The robot was set up with a nRF52840-DK onboard. The main reason to this is to have the ability to do debugging while it is running if something unforeseen happens and to do adjustment underway.

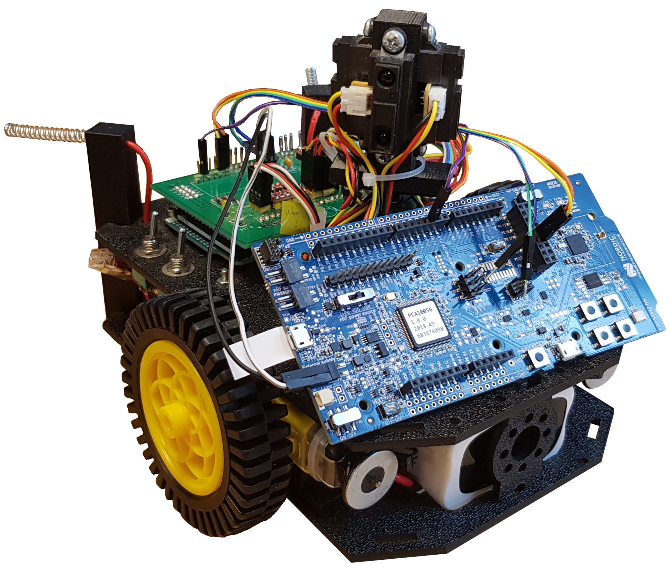


Figure 7.12: Test setup for the robot

Chapter 8

Results

8.1 Bluetooth Low Energy Mesh Network

Reliable messages

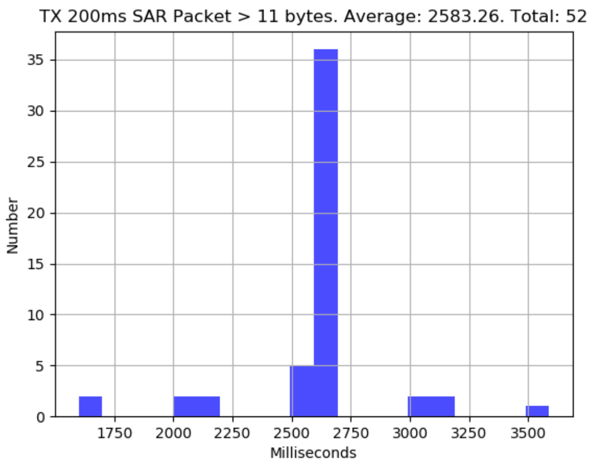


Figure 8.1: Mesh 200ms periodic reliable publishing

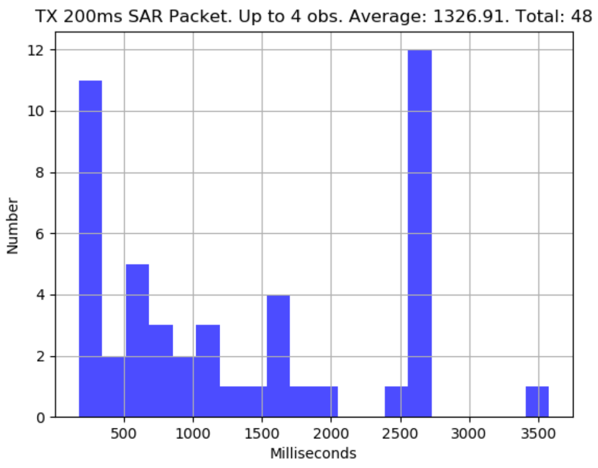


Figure 8.2: Mesh 200ms periodic reliable publishing new approach

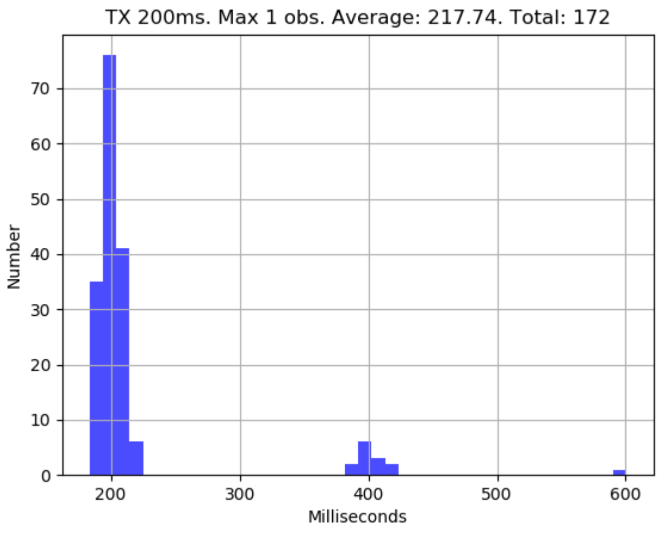


Figure 8.3: Mesh 200ms periodic reliable publishing max one obstacle

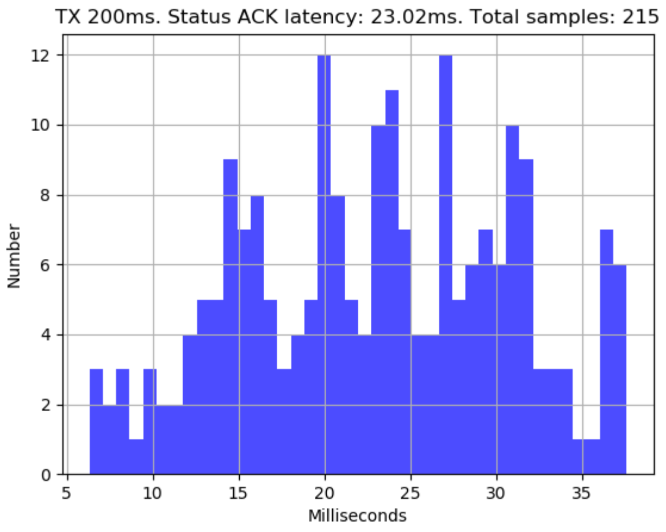


Figure 8.4: Latency status acknowledge

Unacknowledged messages

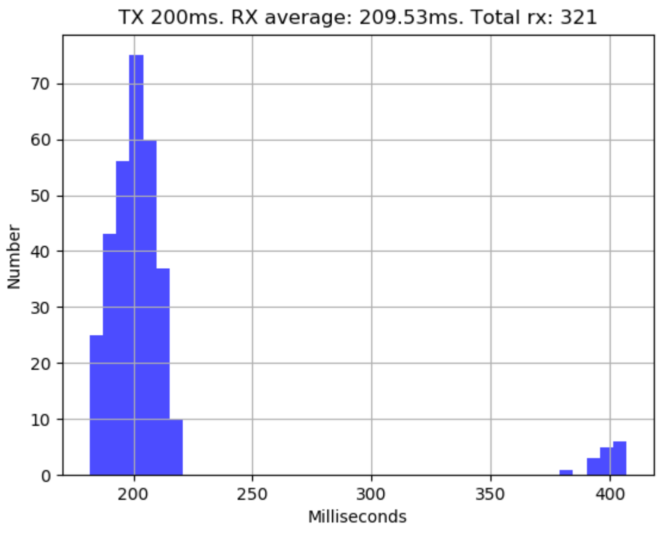


Figure 8.5: Unreliable message RX server side

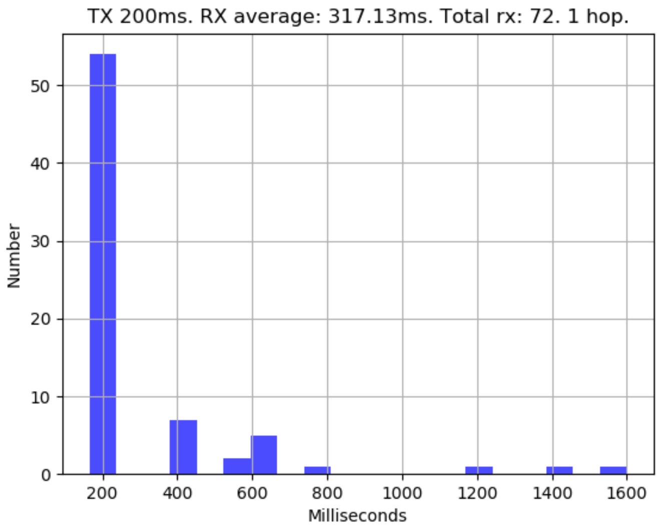


Figure 8.6: Hopping packets RX interval unreliable

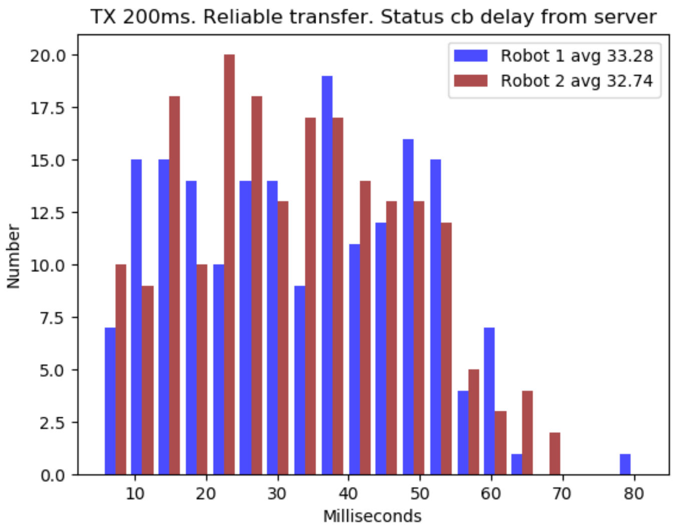


Figure 8.7: Bar plot two sending robots. Interval between sent message en received status ack from the server model

8.2 Thread network with OpenThread

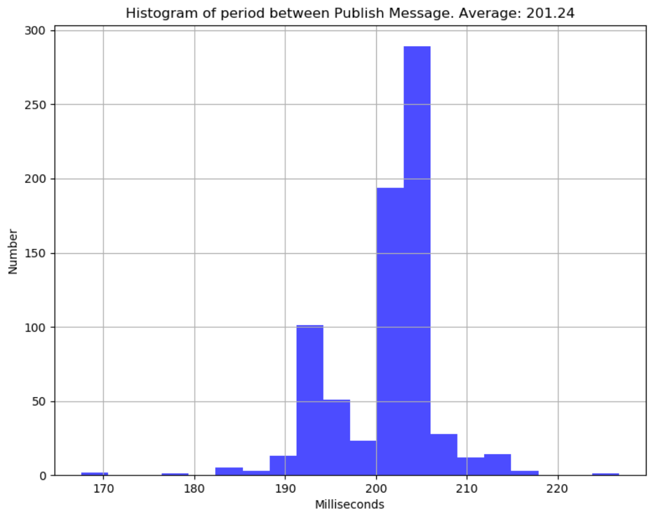


Figure 8.8: Histogram of publish messages QoS 1

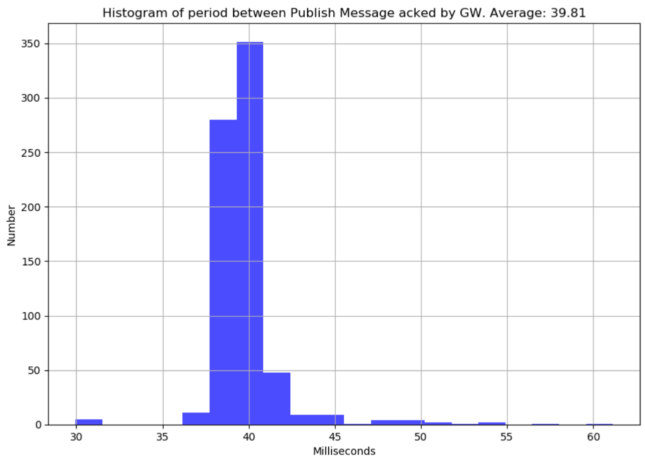


Figure 8.9: Histogram of PUBACK messages. Time from the message was published QoS 1

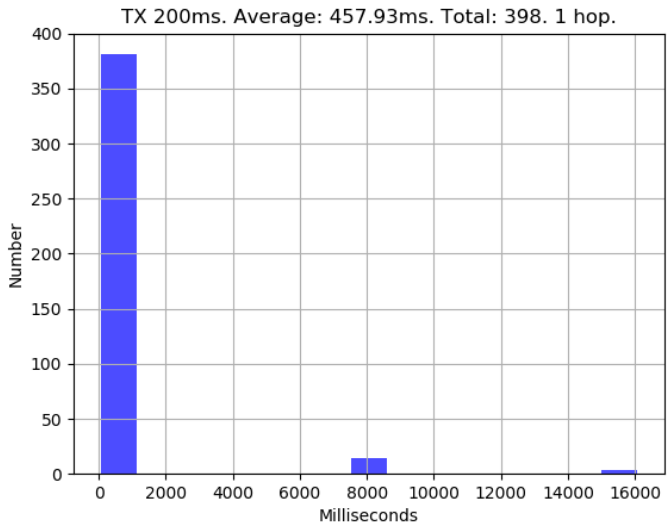


Figure 8.10: Histogram of PUBACK messages with 1 hop. Time from the message was published. Default retransmission interval of 8000ms

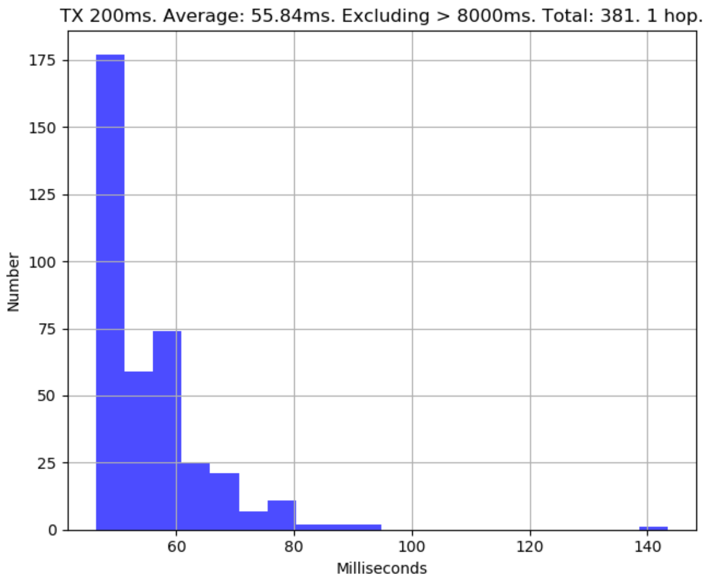


Figure 8.11: Histogram of PUBACK messages but excluding republishing. Time from the message was published

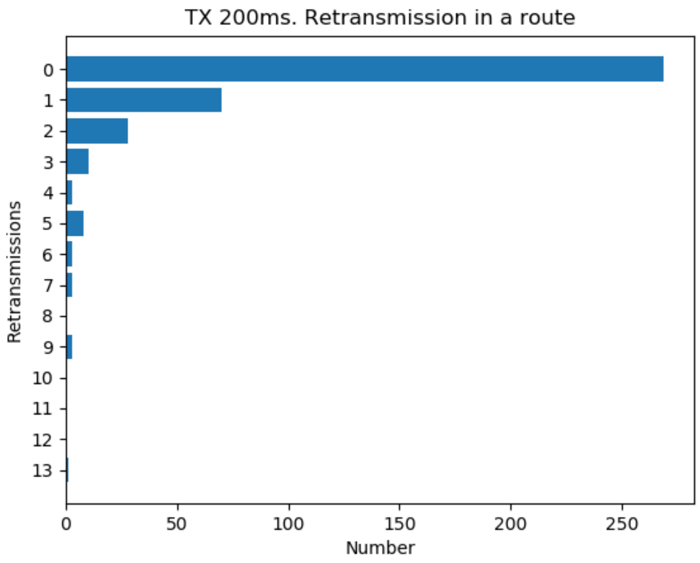


Figure 8.12: Number of retransmissions in route due to lack of 802.15.4 or MQTT acknowledgement

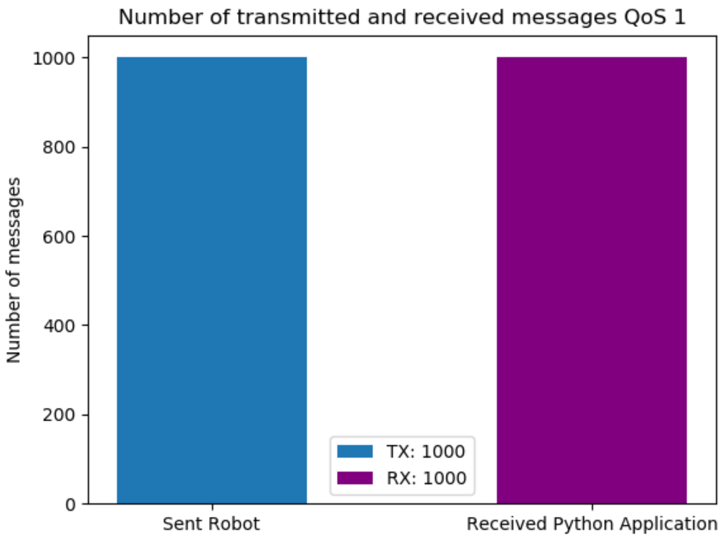


Figure 8.13: Number of messages sent vs received at host computer via broker

8.3 Robot

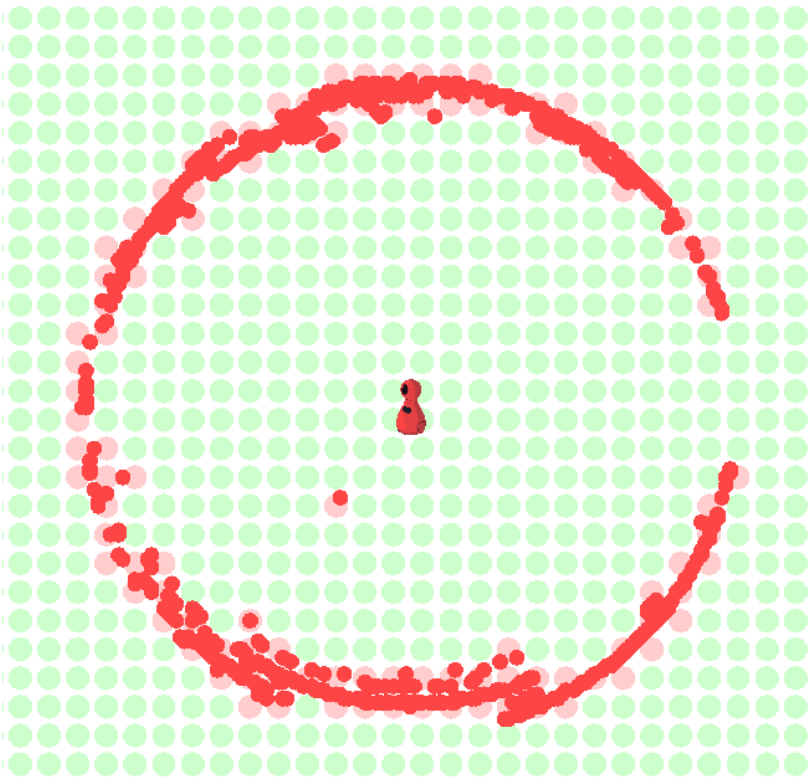


Figure 8.14: Wall detection with MQTT interface

Chapter 9

Discussion and further work

9.1 Performance robot messages

The results in figure 8.1 are made up by a series of timestamps generated when the provisioner received a mesh message in the application layer. What we see in figure 8.1 is that most messages are received after an average of 2583 milliseconds, or 2.6 seconds, which is way far from what we want to achieve and expected of about 200 milliseconds.

I tried with the approach as in the OpenThread implementation with sending the position and up to four obstacles depending on the sensor values. The result turned out to be more efficient, but there are still unpredictable spikes around 2600 milliseconds.

I managed to track down the different messages causing delay or being sent quickly. It turned out that when the message is only 9 bytes which implies only one detected obstacle, that message and the following message get sent immediately. It turned out that if a message has to be segmented, the following messages get blocked. That was not the case for the project I conducted in the autumn; then it was no problem publishing rapidly and while testing the functionality during the implementation, no suitable messages had been tested. However, I found out that Nordic Semiconductor in their most recent SDK had implemented a part of the Mesh Profile Specification, section 3.6.4.1 that states:

"The upper transport layer shall not transmit a new segmented Upper Transport PDU (protocol data unit) to a given destination until the previous Upper Transport PDU to that destination has been either completed or canceled."

Based on the results so far, this implies that the time it takes for the Upper Transport PDU to complete a segmented message is about 2.6 seconds. So, instead of sending up to obstacles, I decided to test out a solution to send only *one* obstacle at a time along with the position of the robot. Figure 8.3 shows the result after enabling only one obstacle to be sent, ensuring the message size to be only 9 bytes. The result shows a more satisfying response every 200ms. Although we can see that some messages are sent after 400 and 600ms which is because the outgoing message gets cancelled, as 3.6.4.1 states.

With Thread, however, based on the results, it shows promising results compared to the issue with BLE Mesh. From 8.8 and 8.9 we can see that we achieve no unexpected values during the time it ran.

9.2 Reliability

The Thread network really shows good results based on the tests conducted. I wrote a Python script on my computer that connected to the broker and subscribed to the robot. The result is shown in 8.13 After 1000 messages were sent, 1000 were also received. This should be expected when sending messages with QoS 1. However, the standard specifies that with QoS 1, the message will be delivered *at least once*, so I was expecting to see some duplicates.

In the case of BLE Mesh, figure 8.5 and 8.3, with unreliable and reliable publishing respectively, shows that there are some messages that gets either lost or cancelled. Though it was hard to get an accurate number of how many got lost. If we look at the messages received at 400 milliseconds and 600 milliseconds in both histograms, a rough estimation is 5% message loss for unacknowledged messages and 8%-10% for the acknowledged messages when cancellation and less then 9 bytes are sent. I find the result a bit counter intuitive.

9.3 Scalability

As of today, the project has only three somewhat operational robots, but in the future, there may be added even more. I got to test the BLE Mesh implementation with two devices running and sending a 9 byte message every 200 milliseconds. Figure 8.7 shows the delay between a message being sent and for it to be acknowledged. In comparison with 8.4, there is an indication that two robots increases the latency by an average of 10 milliseconds. It would have been useful to have even more robots or DKs available to see if it would increase even more, or to a point where the delay gets too high for the robots to cope with. Either no status is received before next message is sent or that the server would not pass the message to the application. As a note, I did not get satisfactory data to neither conclude nor discuss scalability of Thread the same way as with BLE Mesh.

9.4 Message hopping

The result in 8.6 shows 72 messages being received by the server model, but if we take a look at the number of times messages are delayed, that implies how many were lost. In total 42 unreliable messages with 1 hop were lost. That is approximately 30% loss for unreliable messages with one hop. That is a very high value and I definitely question how the set-up was. The main problem is that if the transmitting device was placed at the range limit, such that it was difficult even for the relay node to catch the messages. We can not know for sure which path lead to the most loss, which will remain a uncertainty and source of error.

Figure 8.10 shows the interval of acknowledge publish, which reached the robot. What we see is that some messages are received after 8 and 16 seconds which is the default retransmission interval defined in the MQTT client. Figure 8.11 shows the values below the retransmission interval. On the other hand, figure 8.12 shows how many times a message was retransmitted in the route due to lack of either 802.15.4 or MQTT acknowledgment. This implies that many messages was lost, but the network managed to recover itself. I did not get an accurate number on how many that were lost completely, but with only two messages retransmitted after 16 seconds, there is reason to believe that the number is low.

9.5 The new server and MQTT

As mentioned in section 4.3 I did some recalibration of the IR sensors. The main objective for this was to ensure that the data sent through the new robot interface to the new server was in order and represented correctly. Figure 8.14 shows the data points received by the server sent from the robots while in a testing circle with diameter 160cm. The upper threshold of the IR sensors is 80 cm, so it's hard to get the measurements accurate and consistent. However, the result implies significant opportunities and possibilities for the new server to be used in the future continuation of the LEGO robot project.

9.6 MQTT Broker

For further work and appliances of this system, one change to be done is to have the MQTT broker moved to a closer physical location of the Thread network to avoid latency. Though no consequences of latency was found directly by the results as the gateway handles the correspondance with the remote broker. It could be that messages pile up between the broker and gateway with a larger network, especially with more frequent messages.

9.7 Mapping optimization system within the mesh network

As the communication has been set up to flow not only between a server and a robot device but also between robot devices, it may be feasible to add further functionality regarding mapping and localization. As at least the nRF52-robots have a significant amount of resources and computational power at spare, a Multi-Robot Simultaneous Localization and Mapping system could be the way to go for future projects. This solution can presumably be achievable by adding more models supporting, for example, path suggestions and plans to avoid multiple robots mapping the same area.

Besides, in the past years, there has been a release of multiple alternatives, for example, regarding distance sensors. The IR sensors on the robots today are somewhat outdated by looking at the other alternatives. Having four sensors on a rotating tower is a prolonged and unreliable process as the analog values float on further distances.

9.8 Enhance BLE Mesh throughput

To accommodate the problem with small payload for BLE Mesh, Nordic Semiconductor, has an experimental Instaburst TX module which provides a higher throughput for BLE Mesh at the expense of breaking specification compliance. Another solution to the problem may be to implement local segmentation and reassembly functionality at both the robots and the receiving application so that larger messages can be handled. Last alternative solution is to enable the robot to send one obstacle at the time, but to achieve previous speed, the task sending sensor data may be called more frequently.

9.9 Both systems running on a nRF52840-based robot

The project has a robot with a nRF52832 SoC embedded on a custom PCB to do the same tasks as this robot. That enables BLE Mesh but if the SoC could be upgraded to a nRF52840 SoC, it could have utilized OpenThread and 802.15.4 also. Common for both systems, we could then eliminated the serial interface and optimize the message exchange utterly.

The most exciting point regarding BLE Mesh I would say is to set it up to run concurrently with the robot's tasks and to let there be a shared set of states which is available for both the network, the position estimator and the controller. By enabling this, we can achieve a better approach to an actual implementation towards states described in the Mesh Profile Specification[7].

9.10 Enhance the USB CDC ACM class module implementation

By utilizing the USB CDC ACM class module fully, a more advanced command line interface which can enable configuration and debugging of both the BLE Mesh and Thread systems in real time without having to close a debugging session, recompile your program or start again. Also, as it is not configured fully to receive bytes, that would enable full integration to the older java server if that is to be used in the future continuation of the LEGO project.

Bibliography

[1] Kristian Fjelde Pedersen. *BLE Mesh Communication*. Technical report, Department of Engineering Cybernetics, NTNU, 2018.

[2] Kristian Lien. Embedded utvikling på en fjernstyrt kartleggingsrobot. Master's thesis, NTNU, Trondheim, 2017.

[3] Wikipedia Contributors. *Bluetooth*, 2018. https://en.wikipedia.org/w/index.php?title=Bluetooth&oldid=874093193 [Accessed: 20.03.2019].

[4] Bluetooth Special Interest Group. *State of Bluetooth in 2018 - and beyond*, 2018. https://www.bluetooth.com/blog/the-state-of-bluetooth-in-2018-and-beyond/ [Accessed: 05.04.2019].

[5] Bluetooth Special Interest Group. *Bluetooth Mesh Networking FAQ*, 2018. https://www.bluetooth.com/bluetooth-technology/topology-options/le-mesh/mesh-faq [Accessed: 05.04.2019].

[6] Bluetooth Special Interest Group. *How Bluetooth Mesh Puts the 'Large' in Large-Scale Wireless Device Networks*, 2018. https://www.bluetooth.com/blog/mesh-in-large-scale-networks/ [Accessed: 25.05.2019].

[7] Mesh Working Group. Mesh Profile Bluetooth Specification v1.0, 2017.

[8] Mesh Working Group. Mesh Models Specification v1.0, 2017.

[9] member.relations@bluetooth.com Bluetooth SIG. The bluetooth sig announces a new smart home sub-group. 2018.

[10] Internet Engineering Task Force. *Internet Protocol version 6*, 2017. https://tools.ietf.org/html/rfc8200 [Accessed: 29.04.2019].

[11] Silicon Labs. Mesh Link Establishment Standard, 2015. https://tools.ietf.org/html/draft-ietf-6lo-mesh-link-establishment-00 [Accessed: 24.05.2019].

[12] Silicon Labs. *Mesh Link Establishment Protocol*, 2018. https://tools.ietf.org/id/draft-kelsey-intarea-mesh-link-establishment-05.html [Accessed: 29.04.2019].

[13] Aaron Balchunas. *Routing Information Protocol*, 2012. https://www.routeralley.com/guides/rip.pdf [Accessed: 05.05.2019].

[14] OASIS. *OASIS Message Queuing Telemetry Transport (MQTT) TC*, 2018. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=mqtt [Accessed: 29.04.2019].

[15] OASIS MQTT Technical Group. MQTT v3.1.1 Standard, 2014. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html [Accessed: 29.04.2019].

[16] Simen Nilsen. Unknown Title. Master's thesis, NTNU, Trondheim, 2018.

[17] Nordic Semiconductor. *EasyDMA feature*, 2019. https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.nrf52832.ps.v1.1%2Feasydma.html [Accessed: 15.04.2019].

[18] Nordic Semiconductor. *Simple On Off Model*, 2019. https://infocenter.nordicsemi.com/topic/com.nordic.infocenter.meshsdk.v3.1.0/group__SIMPLE__ON__OFF__CLIENT.html [Last accessed: 25.05.2019].

[19] OpenThread. *OpenThread Network Co-Processor and Spinel*. https://openthread.io/guides/ncp [Accessed: 19.05.2019].

[20] Eclipse. *MQTT-SN Transparent / Aggrigating Gateway*. https://github.com/eclipse/paho.mqtt-sn.embedded-c/tree/master/MQTTSNGateway [Accessed: 20.05.2019].

[21] Nordic Semiconductor. *nRF Sniffer*. https://www.nordicsemi.com/?sc_itemid=%7B655FA723-4404-4FBE-8062-7C5C5DCDF36E%7D [Accessed: 19.05.2019].

[22] Piotr Winiarczyk. *BTMESH: Add access layer decryption*. https://code.wireshark.org/review/#/c/33093/ [Accessed: 30.05.2019].

[23] OpenThread. *OpenThread Pyspinel Github*, 2018. https://github.com/openthread/pyspinel [Accessed: 19.05.2019].

Appendix A

Installation instructions Mesh

For making this project run you need the following at hand:

- 2 x nRF52840 development kit PCA10056
- 2 x Micro USB cables and USB slots on your computer
- Nordic Semiconductor SDK v15.2.0
- Nordic Semiconductor SDK for Mesh v3.1.0
- SEGGER Embedded Studio v4.15
- J-link drivers from SEGGER

If you are reading this thesis online, the bullet points above have links to the correct download pages (as of may 2019). Extract the SDKs and follow the README-files to set them up correctly. After that you can retrieve the zip of the source files. Extract the ”robot_mesh” folder to the ”examples” directory of the mesh SDK, and extract the folders in ”models” to the SDK's ”models” folder.

1. Click into the *provisioner* folder in *robot_mesh* and open ”provisioner_nrf52840_xxAA_s140__6_1_0.emProject”. Segger will open a session with all files neccesary.
2. Once open, click the folder ”Application” in the Project Explorer and open main.c
3. Click ”Target” in the top menu followed by ”Connect J-link”
4. Once connected, click ”Erase all” to erase the entire flash.
5. Click ”Debug” to start a debug session enabling you to see what the application does and how it behaves.

Follow the same steps but for the robot folder, and on another development kit.

Alternatively, you can use nRF Connect to upload the erase and upload the hex files directly. The hex files are located at /robot_mesh/provisioner/build/provisioner_nrf52840 _xxAA_s140_6_1_0_Debug. Same for the robot hex.

If to be runned with the robot, place the robot DK and connect the TX2 and RX2 pins of the robot Arduino Mega to pin 20 and 22 on the DK to enable communication. The interrupt pin PE4 should be connected to P1.05 on the DK. Also, connect power from the robot to the external power supply pins on the DK. Start the robot and the communication will be enabled. Make sure to connect the provisioner to the host computer.

Installation instructions OpenThread

For making this project run on a robot you need the following at hand:

- 1 x nRF52840 Dongle
- 1 x ”Lego Robot”
- Nordic Semiconductor SDK v15.2.0
- Nordic Semiconductor SDK for Thread and Zigbee v2.0.0
- SEGGER Embedded Studio v4.15
- J-link drivers from SEGGER

- Thread Border Router. Check Nordic Semiconductor infocenter for latest updates and guide.
- 1 x nRF52840 developmentkit PCA10056 (Optional)

Extract the SDKs and follow the README-files to set them up correctly. After that you can retrieve the zip of the source files. Extract the ”mqttsn_client_publisher_ot_uart” folder into the ”examples/thread” folder.

1. Click into ”PCA10059/blank/ses”.
2. Open ”thread_mqtt_sn_uart_pca10056.emProject”. SES will then start a session with the necessary files.
3. Once open, click the folder ”Application” in the Project Explorer and open main.c
4. Click ”Target” in the top menu followed by ”Connect J-link”
5. Once connected, click ”Erase all” to erase the entire flash.

6. Click ”Debug” to start a debug session enabling you to see what the application does and how it behaves.

Notice that if a border router or gateway is not set up, the application will keep on trying to connect forever. No messages can be sent before connection is established.

File Hierarchy

Mesh Application

```
/c
└─ nRF5_SDK_for_Mesh_v3.1.0_src
   ├─ examples
   │  └─ robot_mesh
   │     ├─ include
   │     ├─ robot
   │     └─ provisioner
   └─ models
      ├─ min_com
      │  ├─ include
      │  └─ src
      ├─ pos_cmd
      │  ├─ include
      │  └─ src
      ├─ sensor_update
      │  ├─ include
      │  └─ src
      └─ on_off
         ├─ include
         └─ src
```

Thread Application

```
/c
└─ nRF5_SDK_for_Thread_and_Zigbee_2.0.0_<hash>
   └─ Examples
      └─ thread
         └─ mqttsn_client_publisher_ot_uart
```

Appendix B

Description of attachments

B.1 Images

Images used in this report.

B.2 Source code

All of the software developed in this project. See Appendix A for instructions on demonstration.

B.3 Previous reports

Previous reports and software fetched from earlier reports and theses.

B.4 Datasheets and product specifications

All relevant datasheets and specifications for the hardware used in this thesis.

Pedersen, Kristian Fjelde

Mesh Networking for IoT