



Effiziente Datenzuordnung und Visualisierung für kabellose Sensor Netzwerke

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Technische Informatik

eingereicht von

Daniel Harringer

Matrikelnummer 11775835

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner

Mitwirkung: Dipl.-Ing. Daniel Ramsauer

Wien, 1. Juli 2021

Daniel Harringer

Wolfgang Kastner



Visualization and efficient data mapping of wireless sensor networks

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Computer Engineering

by

Daniel Harringer

Registration Number 11775835

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner

Assistance: Dipl.-Ing. Daniel Ramsauer

Vienna, 1st July, 2021

Daniel Harringer

Wolfgang Kastner

Erklärung zur Verfassung der Arbeit

Daniel Harringer

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Juli 2021

Daniel Harringer

Danksagung

Allen voran möchte ich meinen Eltern danken, die mich in meiner bisherigen Studienzeit sowohl tatkräftige als auch finanziell immer unterstützt haben. Darüber hinaus möchte ich ihnen danken mich immer wieder motiviert haben diese Arbeit fertig zustellen.

Ein besonderer Dank gilt auch meinem Betreuer Daniel Ramsauer für seine umfassende Unterstützung im Zuge dieser Arbeit.

Acknowledgements

First of all, I would like to thank my parents, who have always supported me both actively and financially during my studies. In addition, I would like to thank them for always motivating me to finish this thesis.

Special thanks go to my supervisor Daniel Ramsauer for his comprehensive support during the course of this work.

Kurzfassung

Das Internet of Things (IoT) ist ein Netzwerk bestehend aus diversen elektronischen Geräten, deren Aufgabe es ist, verschiedenste Daten zu sammeln und über das Internet oder einen anderen - auf dem Internet Stack basierenden - Protokoll auszutauschen. Den größten Anteil an Geräten im IoT machen kabellose Sensoren aus, welche ein sogenanntes Wireless Sensor Network (WSN) formen und weitflächig eingesetzt werden, um Daten zu sammeln und diese an eine zentrale Stelle oder einen zentralen Server zu senden. Aufgrund der großen Anzahl an verwendeten Sensoren ist es schwierig, einen Überblick über die Verteilung und die Konfiguration der einzelnen Sensoren zu bekommen und zu behalten. Die Sensoren im WSN werden größtenteils durch eine Batterie betrieben. Um die Batterielaufzeit zu erhöhen, werden energie-intensive Aufgaben, beispielsweise Kommunikationsaufgaben, auf ein Minimum reduziert. Trotzdem müssen die generierten Daten vom Sensor an die zentrale Stelle geschickt werden. Um dies möglichst energieeffizient zu ermöglichen, werden beispielsweise nur Messdaten und keine Meta-Informationen versendet, wobei dadurch eine weitere Verarbeitung durch andere Komponenten erschwert wird. Die Ziele dieser Arbeit sind es, die vernachlässigten Meta-Daten auf dem Weg von dem kabellosen Sensor zu einem Sammelpunkt zu aggregieren, um die Verarbeitung durch weitere Komponenten zu erleichtern und das WSN in einem Modell abzubilden, welches einen Überblick über die Sensoren und deren Konfiguration geben soll. Um dieses Ziel zu erreichen, wurde ein simples WSN aufgebaut und im Bezug auf den zuvor definierten Anforderungen evaluiert.

Abstract

The IoT is a network of electronic devices, which are collecting and sharing information using the Internet or lightweight protocols, mostly building upon the Internet communication stack. Most devices in the IoT are wireless sensors, which form so called WSNs and they are widely used in the context of IoT for collecting data and transmitting them to a sink. Since these networks consist of hundreds of sensors, it is hard to keep track. The wireless sensors are often battery powered and therefore restrict their communication activities to expand their lifespan. Wireless sensors produce a high amount of data, which is transmitted to a common sink but due to the limited lifespan the transmitted data often does not provide any meta-data, which complicates further processing steps. The goal of this thesis is to aggregate neglected meta-data on the way from the sensor node to the sink, in order to simplify the computational logic on the sink and to construct a model of the sensor network that could help to keep track of the sensor nodes and their parametrization. In order to achieve this goal, a simple example WSN was set up and evaluated regarding the defined requirements. The results and findings are presented within this thesis.

Contents

| | |
|---|-------------|
| Kurzfassung | xi |
| Abstract | xiii |
| Contents | xv |
| 1 Introduction | 1 |
| 2 Background | 3 |
| 2.1 Related Work | 3 |
| 2.2 Internet of Things | 6 |
| 2.3 Internet Protocol version 6 | 6 |
| 2.4 MQTT | 11 |
| 2.5 OpenThread | 15 |
| 2.6 Ontologies | 21 |
| 3 Methodology | 25 |
| 4 Implementation | 27 |
| 4.1 Sensor nodes | 28 |
| 4.2 Border Router | 29 |
| 4.3 Toolchain Setup | 30 |
| 4.4 Brick Model | 34 |
| 4.5 Proof of Concept | 37 |
| 5 Results | 43 |
| 5.1 Possible points for data aggregation | 43 |
| 5.2 Visualization sensor data using a graph environment | 44 |
| 6 Discussion | 51 |
| 6.1 Analysing points for data aggregation | 51 |
| 6.2 Evaluation of the presented WSN models | 52 |
| 6.3 Brick extension for Wireless Sensor Networks | 54 |
| 6.4 Discussing result regarding research questions | 54 |
| | xv |

| | |
|------------------------|-----------|
| 7 Conclusion | 57 |
| List of Figures | 59 |
| List of Tables | 61 |
| Listings | 63 |
| Bibliography | 65 |

Introduction

The IoT is a network of electronic devices, which is collecting and sharing information using the Internet or lightweight protocols, mostly building upon the Internet communication stack. Most devices in the IoT are wireless sensors, which form so called WSNs. They are widely used in the context of IoT for data collection and transmission to the sink. WSN consists of hundreds of sensors and produce a huge amount of data for service applications on higher layers, for example cloud services [1]. In most cases, the data gets sent without information about the type, the source or the usage of this data, meaning higher service levels have to figure out this information before they start processing it. For most service applications, the data has a specific format so the sink is able to process it in the right way, which does not only limit the usage of the WSN and the extendability of services, but additionally makes it hard for humans to figure out where the data is from. This may lead to redundant placement of sensor nodes, resulting in a higher maintenance effort. In order to minimize or even eliminate the redundant sensor placement and make the WSN useable for more and also extendable services, the generated data gains additional information while getting transported from the sensor to the application server [2]. This information might be necessary when it comes to environmental parameters, the unit (i.e. temperature in °C), the physical location of the data's source (i.e. Meeting Room 2) or adding a unique identifier used within the whole system if such data is required. Another considerable aspect of data aggregation is the location within the environment where additional data is added. For example, which component is suited best for adding the spatial information of the temperature measurement. This thesis focuses on how to add such meta-data, which data is important to add and additionally, where is the best location to add this kind of data.

A WSN is very difficult to survey due to the high number of components and its possible physical expansion. In order to keep track of all the data, where it comes from, where its target is or what kind of data a specific node sends to the sink, a suitable visualization is needed. Furthermore, the visualization might not only show the data types, sources and

sinks, but also the entire network, which includes the components (i.e. sensor nodes or cloud services) and the relations between each component. In addition, the model should also contain parameter settings of different components (i.e. communication parameters). Therefore, another focus of this thesis lies on possible visualizations of the WSN and sensor meta-data.

In order to address the mentioned challenges, the following research questions have been formed:

RQ 1: Which location in the environment is most suitable for data aggregation in an ongoing process?

RQ 2: What are possible solutions for visualizing aggregated sensor data into a graph environment?

Background

2.1 Related Work

A WSN is a set of hundreds or thousands of identical low-cost sensors with limited computation and communication capabilities and in most cases these sensors are battery powered [1]. Since WSNs are large scaled and commonly used to generate data but not to process them, the data is transmitted to a common sink. In order to achieve this mostly one or more **Gateways** are needed, which have more processing power and higher communication capabilities. These **Gateways** are connected to a group of WSN nodes in order to manage and monitor them, and further, to transmit the generated sensor data to a central server or the cloud. The WSN, the **Gateways** and the central server or cloud can be seen as a three-tier architecture, see Figure 2.1 [3].

In order to achieve a long lifetime of sensor nodes in the network, the communication tasks should be kept as short as possible, so that they transfer only a minimal set of data. An energy efficient technique to minimise the communication tasks and aggregate data in order to reduce the message payload is presented in [4].

The authors assume that the sensor nodes within their WSN have a limited lifetime. Nevertheless energy optimizing algorithms can also be applied within WSNs which have unlimited lifetime, e.g. to extend the battery replacement intervals. A comparison of different data aggregation techniques depending on network topology and clustering methods for aggregating data packets is done in [5].

In a WSN, massive data are generated, which need to be stored and processed somewhere. However, most of the generated data is redundant, because of the physical placement or the slowly changing environmental parameters. This data can be merged or eliminated, as it does not provide any additional information to the central server or the cloud, which consequently can reduce the payload, time and energy used for communication.

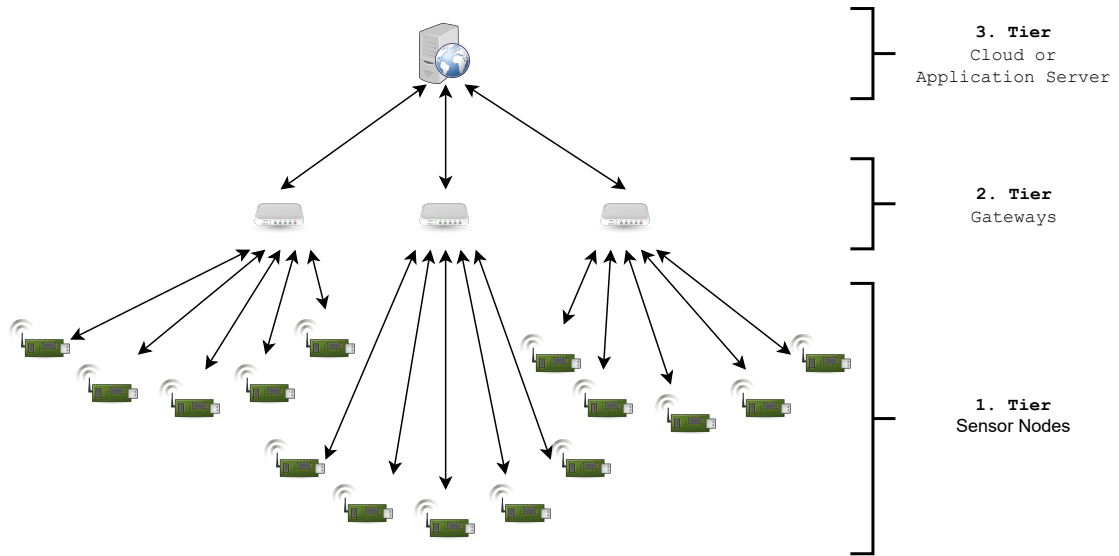


Figure 2.1: A 3-tier WSN architecture, based on and adapted from [3]

One way to combine data is described by the authors of [2]. The data is merged when overlapping paths on the route to the destination exist. If a sensor node gets a package for forwarding, and this package has the same destination as its own package, it merges them into one and sends it to the sink. This technique reduces package loss and the energy consumption (compared to direct forwarding of packages), but the latency increases due to the merging and certain message forwarding timeouts. These timeouts are used to prevent the sensor node from waiting for more data until it can send its own package.

Data that is generated in a WSN can be processed with paradigms of Big Data computation [6], but this approach still needs to eliminate redundant data or add additional information for further processing or categorising. The authors of [7] use the model of a three-tier architecture. The sensors, which compose the first layer, are generating application specific data and form a network based on the application type. Since these sensors are aware of application changes by checking the data packages coming from the second or third layer, they add additional information to their data in order to help any of those two layer to process this data.

As mentioned before, sensors deliver application specific data. Assuming that there is a second application that also needs the data of this sensor, but uses another communication protocol or network topology, it would be necessary to deploy an additional sensor node at the same position, in order to get the same data for the second application. This would not only be a physical redundancy and may also interfere with the communication process of the first network. The authors of [8] address this problem and use a rather common technique in computer science. They overlay a virtualized WSN on top the physical one, so the redundant deployment of the sensors will be eliminated and the sensor can be used for further applications.

Some further issues and challenges of data collecting and transferring them through the network, focusing on energy consumption and inhomogeneity of a WSN especially, are discussed in [9].

System on Chip

The IoT consists of billions of devices equipped with processing, memory and communication capabilities. These devices need to be inexpensive and should be able to work with performance, power and area constraints. This is possible with the System-on-Chip (SoC) architecture, which merges one or more application processors, memory blocks and a range of peripherals for communication and I/O operation on a single integrated chip. The architecture defines the system-level building block for all components and the interconnection between them. These blocks are often reusable logic units also called Intellectual Property Cores (IP-Cores), which have a defined functionality, for example processing signals. They are developed, maintained and licensed by a party and should simplify the design of a SoC.

Figure 2.2 illustrates some basic elements of an SoC system. It includes a number of processors which are connected to one or more memory blocks. SoCs also have analogue and custom circuitry for computing sensor data, analog-to-digital conversion or to support wireless transmission. A SoC implemented in smart phones, for example features circuitry audio input and output capabilities, Internet access functionality, multimedia facilities for video communication, document processing and entertainment for games [35].

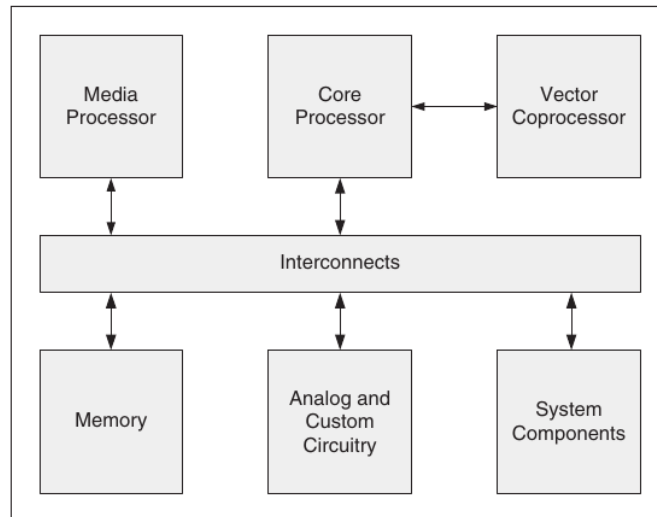


Figure 2.2: A basic SoC system model, from [35]

If it is impossible to fit all components onto one chip, some of the peripherals and the chip are placed on a PCB and are termed system on board.

The difference between a SoC and a conventional general-purpose computer with memory on board is the specific design. A general-purpose computer can run nearly any program because it interprets the instructions at runtime, which makes it more flexible and adaptable for many applications, even for running them at the same time. A SoC is designed for a known application, this means the element can be selected and sized to fit the applications as required.

2.2 Internet of Things

The IoT is a network of electronic devices, including both digital and mechanical ones, which are collecting and sharing information. These 'smart' devices, also called 'things', can be everything, e.g. an industrial machine, a sensor that tracks information about the human body or even a TV. For most parts, IoT-devices are not supposed to be interconnected or connected to the Internet, as opposed to appliances, wearable or healthcare devices. The IoT has the ability to collect and analyse data automatically, share it with connected devices and perform the right action in real-time without any human interaction.

The IoT consists of three main parts: sensors, microcontrollers and one or more service platforms. The sensors are installed into the physical world to acquire data and events about the environment, whereas microcontrollers are responsible for sharing information provided by the sensor with other microcontrollers and services. The service platforms are used to analyse the collected data, process it and set actions according to defined rules. These platforms are also responsible for improving the user experience by enabling user based system rules [10].

The IoT uses the Internet Protocol (IP) as fundamental protocol to connect all devices together and thereby allowing them to communicate with each other. Based on this, many other protocols like OpenThread or MQTT, were developed to improve the IoT in order to make it more powerful, more secure and more flexible for end users and application developers [11].

2.3 Internet Protocol version 6

Due to an increasing number of devices on the Internet and the need of a unique address for each device, the limited address range of the Internet Protocol version 4 (IPv4) is running out of space. Therefore, the Internet Engineering Task Force (IETF) started to design the Internet Protocol version 6 (IPv6) as a direct successor of IPv4. IPv6, just like IPv4, is a communication protocol for computer networks with the task to find a path from one communications partner to another, even through different networks. The new version of the IP does not only extend the existing address space of IPv4, also it implements new features such as more IPv6 addresses per host, automatic configuration of addresses and faster routing [12].

IPv6 Address structure

In IPv6 the address consists of 128 bits (16 bytes) which enables more than $3.4 * 10^{38}$ possible combinations. An address is encoded in HEX - format and is split up into blocks of two bytes, separated by an ':'. In Table 2.1, an example IPv6 - address is given. The first 64 bits are called the **Network Identifier**, the last 64 bits are called **Host Identifier** or **Interface Identifier**. The **Host Identifier** needs to be unique in a network to address a host but can be used for multiple interfaces. It might be the factory assigned EUI-64 Identifier or generated by the MAC - Address, called the modified EUI-64 Identifier [13].

| Network Identifier | | Host Identifier |
|---------------------|---|---------------------|
| 2001:0db8:0000:0000 | : | 0209:0000:0000:ec1f |

Table 2.1: Example IPv6 Address, from [14]

In order to make the addresses readable two modifications are allowed, the leading '0's can be neglected and exactly one block of '0' or sequence of '0'-blocks can be replaced by '::'. If more than one '0'-block is replaced, the address can't be rebuild exactly. The result of applying this rules to the given example IPv6 address can be seen in Table 2.2. The IPv6 supports, as its predecessor, prefix notation.

| Network Identifier | | Host Identifier |
|--------------------|---|--------------------|
| 2001:db8: | : | 209:0000:0000:ec1f |
| 2001:db8:0000:0000 | : | 209::ec1f |

Table 2.2: Shortened IPv6 Addresses

Types of IPv6 Addresses

In IPv6 three categories of IPv6 addresses exist. The **Unicast address** is a single interface, this means that packages which are addressed to a **Unicast address** are delivered to a single interface. **Anycast addresses** identify one or more interfaces. Servers which offer the same services use the same **Unicast address** and packages sent to this address are routed to the nearest server. This category of addresses is used for load-balancing, known as 'one-to-nearest' address. **Multicast addresses** in IPv6 are used for the same purpose as in IPv4, namely delivering packages to many interfaces. This address is also known as 'one-to-many' addresses. A major difference between IPv6 and IPv4 is the replacement of the Broadcast address by **Anycast-** and **Multicast addresses**. [13]

The IPv6 standard introduced some new terminology such as **link**, **interface** and **scope**. A **link** is a set of network interfaces that is bounded by routers and use the same **Network identifier**. It can be compared to an IPv4 subnet or network segment. An **interface** is the attachment of a node to a link.

Unicast addresses split up into three subtypes; the Global Unicast-, the Unique Local- and Link Local addresses. Global Unicast addresses are public addresses which are used to access the Internet. Currently only **2000::/3** addresses are able to be routed on the Internet. Unique Local addresses have the same functionality as local addresses in the IPv4 and have a **fc00::/8** or **fd00::/8** prefix. These addresses can only be assigned and addressed by devices in a private network. The Link Local address is generated automatically or manually configured by the host and it always starts with **fe80::/10**. After assigning an address to an interface, it immediately starts to communicate on the ISO/OSI model layer 3. Link Local addresses are only available in that network segment the host is connected to. Routers will not forward any packages addressed to a Link Local address.

A Multicast address is used to communicate with a dynamic group of hosts and starts with **ff00::/8**. The bits 9 to 12 are flags to specify the type of the Multicast address. The 12th bit indicates if it's a permanently-assigned Multicast address, assigned by the Internet numbering authority, or a non-permanently-assigned address. The next four bits limit the scope of the Multicast address. The most common scope values are:

| Value | Scope |
|-------|--------------------------|
| 1 | Node-Local scope |
| 2 | Link-Local scope |
| 4 | Admin-Local scope |
| 8 | Organization-Local scope |
| E | Global scope |

Table 2.3: Common scope values for IPv6 Multicast addresses, from [15]

The remaining address bits form the Multicast group identifier. A graphical overview of the IPv6 scopes can be found in Figure 2.3. Further informations on IPv6 Multicast addresses and scopes can be found in [15].

Anycast addresses do not differ from Unicast addresses since they are located in the same address space. Assigning a Unicast address to more than one interface turns it an Anycast address. A package addressed to an Anycast address will always be forwarded to the nearest host of the assigned group. It is not allowed to use an Anycast address as source address of an IPv6 package.

IPv6 Header

Every IPv6 package is made out of a header and the payload, the data which has to be transported from source to sink.

In IPv6 the header format is fixed (see Figure 2.4), it always has a length of 40 bytes and only contains the information which is needed to route the packages through the

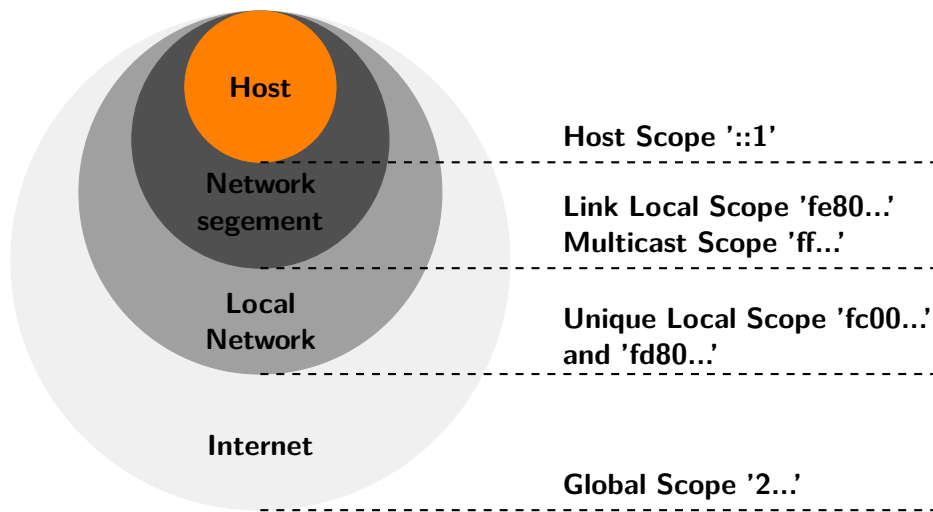


Figure 2.3: IPv6 Address scopes, based on and adapted from [16]

network. The fixed format of the header allows an optimized computation of the packages. This speeds up the forwarding process of messages and makes it easy to build network hardware. The definitions of the header fields can be seen in the Table 2.4

| | | | | | | | | | |
|---------------------|---|---------------|----|-------------|----|-----------|----|----|--|
| 0 | 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 | |
| Version | | Traffic Class | | Flow Label | | | | | |
| Payload Length | | | | Next Header | | Hop Limit | | | |
| Source Address | | | | | | | | | |
| Destination Address | | | | | | | | | |

Figure 2.4: IPv6 Header format, based on and adapted from [14]

In order to make the protocol more flexible, so called extensions headers are introduced to the new standard. This allows to implement new features even after the release, without changing the specification and implementation. Each header points to the next header whereby a so-called **header chain** is formed. A special function represents the **no next header** frame, it indicates the end of the chain. The transport protocol header, mostly the

2. BACKGROUND

| Field | Bits | Description |
|---------------------|------|---|
| Version | 4 | Internet Protocol version number. |
| Traffic Class | 8 | Package priority. |
| Flow Label | 20 | Labels the package for special services e.g. 'real-time' service or non-default quality of service. |
| Payload length | 16 | Length message excluding the header. |
| Next Header | 8 | Identifies the type of the header following immediately after this header. |
| Hop Limit | 8 | Equivalent to the IPv4 time-to-live field. |
| Source Address | 128 | Address of the originator of the packet. |
| Destination Address | 128 | Address of the intended recipient of the packet. |

Table 2.4: Meaning of IPv6 header fields

Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP) header, is placed before the payload, see Figure 2.5. Every extension header needs to be computed in the same order it was added by the source.

With concatenating headers, it is necessary to keep in mind, that the Maximum Transmission Unit (MTU) of the network link always stays the same. This means, if there are multiple headers in a message, the overall payload size of the package is reduced. In order to send bigger messages, a jumbo frame is necessary, which is only possible in a private network; or split up the message into several messages. The MTU describes the maximum package size of a protocol of the network layer, in the ISO/OSI model, which can be transferred in a single transaction without fragmenting the message.

The most frequently used extension headers are listed in the Table 2.5.

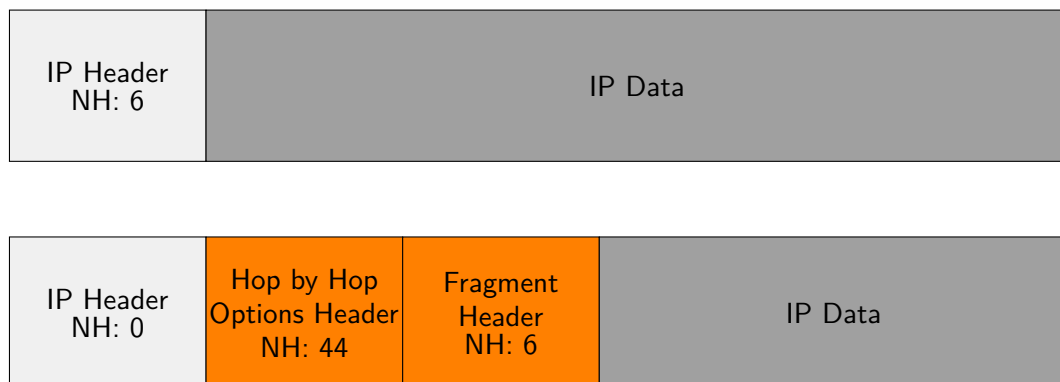


Figure 2.5: Example IPv6 Header and IPv6 Header chain, based on and adapted from [14]

| Name | Integer | Description |
|---------------------|---------|---|
| Hop-by-Hop Options | 0 | Optional Informations for all routers which route the package. |
| Routing | 43 | List of intermediate nodes on the route that has to be visited. |
| Fragment | 44 | Information about the package if its fragmented. |
| No next Header | 59 | Last header in the chain. |
| Destination Options | 60 | Options for the destination host. |

Table 2.5: Most frequently used extension headers of IPv6

Network automation

Due to the automatic generation of the **Host Identifier**, the **Link-Local** and **Multicast-Link-Local** addresses, a network device can find routers automatically. If the device joins a network, it sends a **Router Solicitation** packet searching for routers in the network. The Router sends a **Router Advertisement** packet periodically to notify other network devices or to answer a **Router Solicitation** packet. Both packets are Internet Control Message Protocol version 6 (ICMPv6) packets, which are sent over layer 3. The **Router Advertisement**, advertises the router in the network, the **Network Identifier** and the **Default Gateway**. Therefore, the joined network device can configure its IPv6 address and is able to communicate through out the network.

2.4 MQTT

The Message Queuing Telemetry Transport (MQTT) Protocol is a simple and lightweight communication protocol which is mainly used for machine-to-machine (M2M) communication and designed to fit the requirements of IoT. It was developed by Dr. Andy Stanford-Clark of IBM, and Arlen Nipper of Arcom in 1999 and was used as a proprietary protocol until 2010 when MQTT version 3.1 was officially released under free license. Shortly after the release, the Organisation for the Advancement of Structured Information Standard (OASIS), standardized MQTT and it became part of the ISO standard in 2016 (ISO/IEC 20922:2016) [17].

MQTT is based on a publisher/subscriber model and assures the transmission of messages from clients to a server over the TCP. There are two types of devices in a typically MQTT Network, multiple MQTT Clients and an MQTT Broker also known as MQTT Server. Any device or program, which is connected to the network and exchanges application messages, is called an MQTT Client. This client can be either publisher, subscriber or both. A publisher announces application messages and a subscriber applies for application messages. The MQTT Server is a device or program which interconnects all clients and accepts or transmits the application messages to all connected clients [18]. A client publishes its application messages to the server on a predefined or generated topic. The

broker collects and organizes the data and is responsible for forwarding messages to clients, which are subscriber of this topic.

Topics

Every data that is transferred in MQTT networks is specified by a **Topic**. Topics refer to a UTF-8 string and contain one or more **Topic** levels, each separated by the **Topic** level separator. A valid **Topic** name is case-sensitive and must contain at least one character, which also includes the **Topic** level separator alone.

Examples for valid topics:

- house/firstfloor/kitchen/temperature
- house/firstfloor/livingroom/temperature
- house/firstfloor/livingroom/lightswitch
- house/groundfloor/entrancehall/temperature
- house/groundfloor/entrancehall/doorlock

MQTT allows the usage of wildcards in **Topic** names for subscription only. This allows clients to subscribe to one or more **Topics** at once. The standard differs between single-level and multi-level wildcards. The '+' character represents the single-level wildcard, which can be used to replace one level in the **Topic**.

For example, a subscription to house/firstfloor/+/temperature would be exactly the same as, subscription to these **Topics**:

- house/firstfloor/kitchen/temperature
- house/firstfloor/livingroom/temperature

The multi-level wildcard character is represented by the '#' and is used to subscribe to every **Topic** level lower than the replaced one including this level. The '#' character has to be the last symbol in the **Topic** name, otherwise it would be an invalid **Topic**.

If the client subscribes to house/firstfloor/#, it would receive messages which are published under the following **Topics**:

- house/firstfloor/kitchen/temperature
- house/firstfloor/livingroom/temperature
- house/firstfloor/livingroom/lightswitch

Quality of Service

The MQTT standard offers three different Quality of Service (QoS) levels [19]. QoS level 0 delivers messages **at most once**, which means the sender never waits for any acknowledgement by the receiver. Every message with the QoS level 1 is acknowledged by the receiver, if this is not the case it will, be sent again. So the protocol assures message delivery **at least once**. Since QoS level 1 occasionally produces duplicated undesired messages, therefore the QoS level 2 was designed. This level satisfies the delivery of a message **exactly once** [18].

Transmission Sequence

Before clients and server can exchange application messages, they need to connect with each other and come to an agreement concerning the connection parameters.

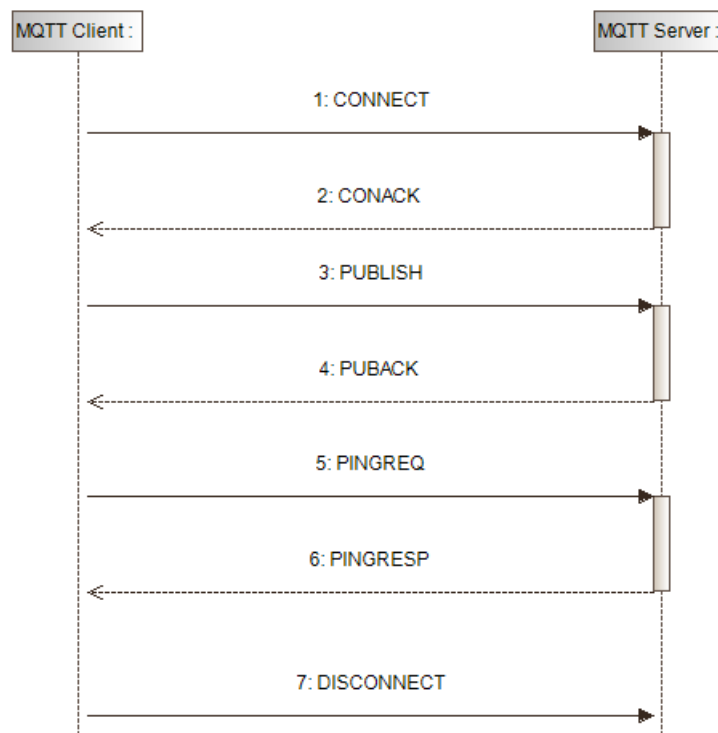


Figure 2.6: MQTT connection, based on diagram 1 and 2 and adapted from [18]

The client, that wants to connect to the server, sends a **CONNECT**-packet containing flags, protocol level and other fields. The server then returns the status of the connection with a **CONNACK**-packet.

If the connection establishment was successful, the client starts publishing application messages, as shown in Figure 2.6 with the QoS level 1, or subscribes to one or more Topics. Publishing data with the QoS level 1 are acknowledged by the server with a

PUBACK-packet. In order to subscribe to a **Topic**, the client sends a **SUBSCRIBE**-packet with the **Topic** name in the UTF-8 encoding. The server acknowledges the subscription and starts forwarding the application messages, which are published under this topic name.

After a certain timeout the connection is automatically terminated. In order to restore and reinform the server that a client is still alive, it sends a **PINGREQ**-packet to the server and waits for an acknowledgement packet.

If a client device wants to terminate the connection, it sends a **DISCONNECT**-packet to the server. The server repudiates the orders and drops all incoming messages concerning this client [18].

2.4.1 MQTT-SN

The most commonly used devices in the IoT are (wireless) sensor nodes because of their low costs, easy subsequent installation and low power consumption. Typical characteristics of these nodes are a lack of processing power and a very limited storage. The drawbacks above are the reason Message Queuing Telemetry Transport for Sensor Networks (MQTT-SN) [20] is especially tailored for this type of sensor nodes.

MQTT-SN is designed to work in a similar way as MQTT, but with some major differences concerning message payload and connections between devices in the network. In order to reduce the message payload in MQTT-SN, the **Topic** name is replaced by simple **Topic** identifier or just kept shorter as in MQTT. Since a typical wireless sensor is battery powered, the MQTT-SN protocol uses UDP to transfer application messages instead of the TCP. This is because TCP assures a permanent connection which is very power consuming and hardly ever needed by wireless sensors. Also MQTT-SN implements support for so-called sleeping clients in order to save battery power when they are not processing data. The **Gateway** (see Section Gateway) keeps track of the sleeping state of clients and buffers the messages for later delivery when they wake up.

In MQTT-SN, also QoS level 3, commonly known as -1, is defined. It works similar to the QoS level 0, which does not need any acknowledgement from the receiver, but with one main difference to level 0 regarding the omission of connection establishment. This means MQTT-SN devices which publish data with QoS -1 send their message without bothering about connecting to a broker or **Gateway**.

Gateway

In order to use MQTT-SN clients with MQTT servers the MQTT-SN standard defines one additional device, a so-called **Gateway**. These **Gateways** are used to translate the application messages from one protocol into another. MQTT-SN distinguishes between three types of **Gateway** architectures: the Transparent (a), Hybrid(b) or Aggregated (c) architecture, as shown in Figure 2.7. In the Transparent architecture every **Gateway** is connected to one node, so that every MQTT-SN connection has a corresponding MQTT

connection. The Aggregated Gateway handles more nodes at the same time and they share the same MQTT connection. A Hybrid Gateway architecture allows MQTT-SN nodes to connect to multiple Gateways at once. In general the number of Gateways is lower than the number of nodes [21].

Most of the time, every MQTT server software has a built-in functionality to provide different types of Gateways for MQTT-SN connections. Some nodes might be placed too far away from the server so that they are not able to connect, due to the fact that Gateway devices are used as Forwarder to the server or to another Gateways.

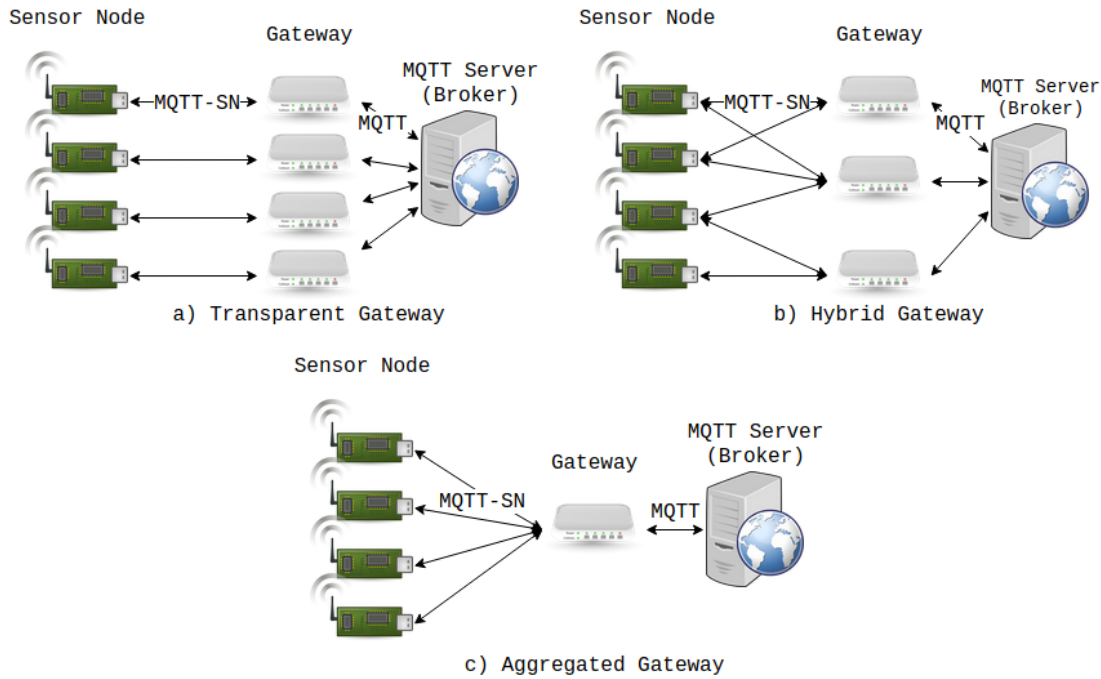


Figure 2.7: Various MQTT-SN Architectures for PUBLISH to server, based on and adapted from [21]

2.5 OpenThread

Thread is a recently standardized low-power network protocol for the IoT, driven by an industry consortium, lead by Google/Nest, which is called the Thread Group [22]. Thread allows device-to-device and device-to-cloud communication and aims towards improving the concept of the IoT. The Thread open network standard, is based on IPv6 (see Section 2.3) and IPv6 over low-power wireless personal area network (6LoWPAN) standard [23]. OpenThread (OT) is an open-source implementation of Thread, which is released by Google in order to speed up the development of products for building automations [24].

OT communicates on the ISM, 2.4 GHz band and supports mesh topology of connections (see Mesh Network). In contrast to a star topology, the mesh topology allows individual interconnectivity between network nodes and eliminates the central point of communication, which is responsible for message passing and routing. A mesh network can recover itself after an unexpected disconnection of a network node.

Thread offers high level of security features, as since communication over the network is encrypted and every device is authenticated before joining the network. The ISO/OSI model, IEEE 802.15.4 describes the physical and MAC layer of the Thread stack. In order to transport data through the network, UDP is used. The combination of all three; the UDP, the IPv6 and the 6LoWPAN, defines the Thread standard, as shown in Figure 2.8 [25].

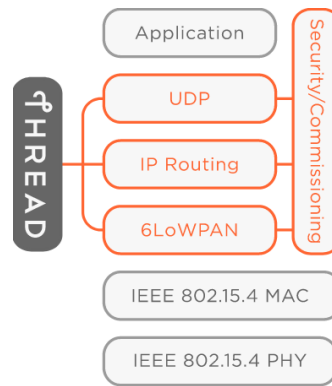


Figure 2.8: Thread protocol stack, from [23]

Mesh Network

A mesh network is a local network topology, where every device is dynamical connected to as many other network nodes as possible, without the need of central devices, such as routers or switches. There are two kinds of mesh topologies, namely full mesh topology (a) and partial mesh topology (b) (see Figure 2.9). In a full mesh network, each node is connected to every other node in the network, otherwise would be called partial mesh network. Mesh networks can organise and configure themselves, this means each node has routing ability and decides which way the packages take from source to sink, depending on the available connection between nodes. Based upon this characteristics, a dynamically distribution of network traffic among all nodes takes place, in case nodes in the network fails. This process is called self-healing [26].

OpenThread Device types

OT mainly distinguish between two types of devices in a Thread network, the Full Thread Device (FTD), which implements the whole functionality of the Thread protocol; and

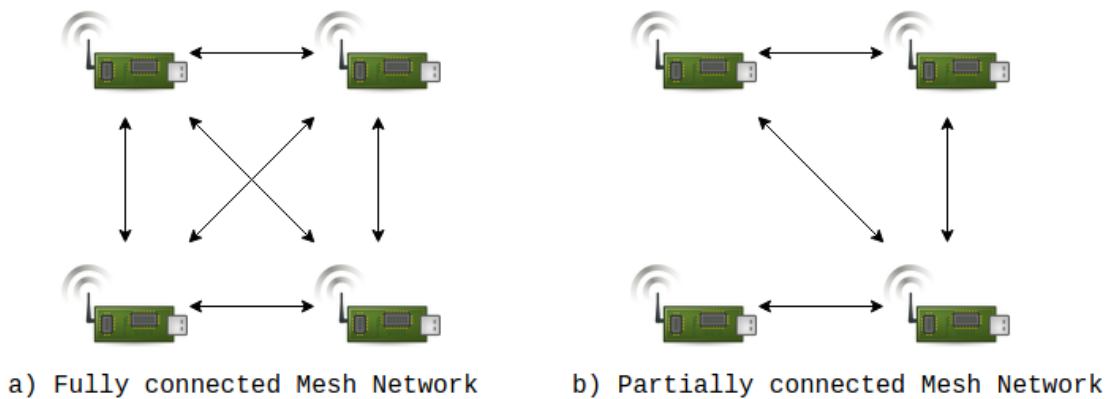


Figure 2.9: Full- and Partially connected Mesh Networks

the Minimal Thread Device (MTD), which implements just the necessary parts. FTD and MTD can be split up in further subcategories as shown in Figure 2.10.

A FTD can operate as router, Router Eligible End Device (REED) or Full End Device (FED). FTDs, except FED, are subscribers of all-router Multicast addresses, are always listening and forwarding traffic from other devices. Also they keep track of the IPv6 address mappings and maintain them if they do not correspond to the actual network. A special type of FTDs are REEDs. These device types can upgrade themselves to routers, if an end device wants to join the network, but does not have a routing device to connect nearby. Instead, the end device will connect to the REED node, which will downgrade itself if no end device is connected to it.

MTDs, which only operate as end devices, do not forward any packages for other network components and are not subscribers of any multicast traffic. An MTD exchanges all messages with the router directly connected to it, and is the only communication partner for that particular node. The router only forwards a package to an MTD, if it is addressed to the node. The subcategories for MTDs are the Minimal End Device (MED) and the Sleepy End Device (SED). Transceivers of devices of type MED are always activated and do not need to poll messages from their parents, because it will always forward messages to the child. In contrast to this, the SEDs are normally in sleep mode, there by saving battery power, and wake up periodically to exchange messages with their parents and perform their tasks.

Network structure

Thread networks have a unique two byte personal area network (PAN) ID, eight byte Extended-PAN ID and a human-readable network name. Every Thread network consist of one leader, up to 32 routers and up to 511 end devices per router, a small Thread network is shown in Figure 2.11.

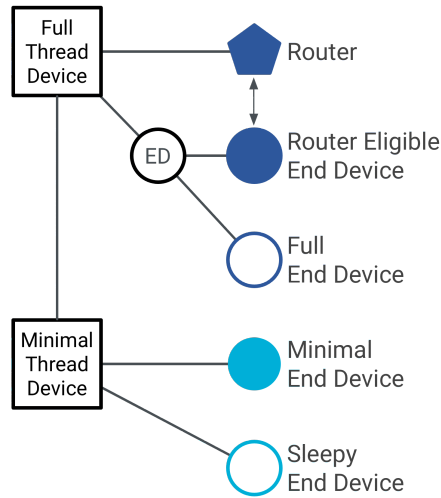


Figure 2.10: OpenThread Devices types, from [24]

A Thread leader is the decision maker in a Thread network. The first router which connects to a Thread network becomes the leader of the network and as soon as it disconnects from the network, all remaining routers dynamically elect a new leader. The leader is responsible for managing and advertising the network infrastructure such as router IDs, 6LoWPAN contexts and collects Border Router (see Border Router) information [27].

Router devices and REEDs can form a Thread network and can allow other devices to join. In order to form a network, the router selects the least busy channel and chooses an unused PAN ID. After forming the Thread network, the device starts advertising to notify other devices to join. If a device wants to join a Thread network, it configures its network parameters according to the network via Thread Commissioning. After commissioning by the leader, the new device is connected to the next reachable router or REED.

A Thread network can be split into multiple partitions, but will still be one Thread network. This can happen if a group of network nodes loses connection to the rest of the network. Each partition works as a normal Thread network, with the same network credentials for all devices. There is no wireless connection between any partition in a Thread network yet and if a partition reconnects to an other, they automatically merge into one.

Border Router

A Border Router establishes connectivity between a Thread network and a non-Thread network, such as Wi-Fi or Ethernet. This ensures the transfer of data between these networks or in the cloud and makes Thread devices directly addressable. The Border

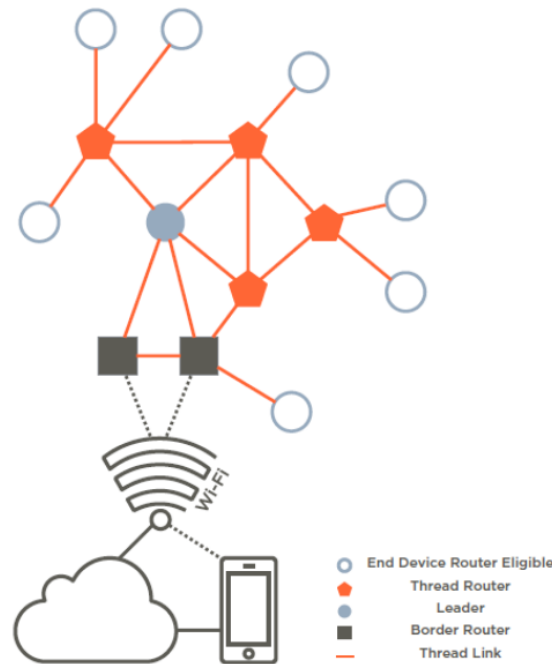


Figure 2.11: OpenThread network structure, from [27]

Router also allows for external commissioning and controlling of the network status. Thread allows multiple Border Router in an network.

IPv6 Addressing

OT uses the IPv6 Standard for addressing and network communication. In Thread there are three scopes for Unicast addresses: the Link-Local, Mesh-Local and Global. The Link-Local addresses reach every Interface within a single radio signal transmission and has the prefix of **fe80::/16**. The Mesh-Local addresses prefixes are **fdxx::/8** and can be reached by every Interface in a Thread network. Global scope addresses can be reached from outside a Thread network.

In order to identify devices in a Thread network, they get a 16 bit unique address, the Routing Locator (RLOC), it consists of a Child ID and a Router ID. The RLOC and a fixed 48 bit address (for instance **0000:00ff:fe00:RLOC16**) form the Interface Identifier of the Thread node and are based on the location in network topology. A Thread node, which connects to a router as a child, will get a Child ID from the router and inherit its Router ID. A router cannot be the child of any other Thread network devices and therefore gets the Child ID 0.

For example a router has the Router ID 1, a child connects and gets the Child ID 1 so the RLOC will be **0x401** (see Table 2.6)

| RLOC16 | | | | | | | | | | | |
|-----------|---|---|---|---|---|---|----------|---|---|---|---|
| Router ID | | | | | | R | Child ID | | | | |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |

Table 2.6: Example RLOC Address, from [24]

Combining Host identifier, including the RLOC, and the configured Mesh-Local prefix generates the IPv6 address for the network device, see Table 2.7. This can be applied to all nodes of a Thread network, see Figure 2.12 for a greater number of devices in a network.

| Mesh-Local prefix | Host Identifier | RLOC16 |
|-------------------|-----------------|--------|
| fde5:8dba:82e1:1 | ::ff:fe00: | 401 |

Table 2.7: Example Device IPv6 Address including RLOC16, from [24]

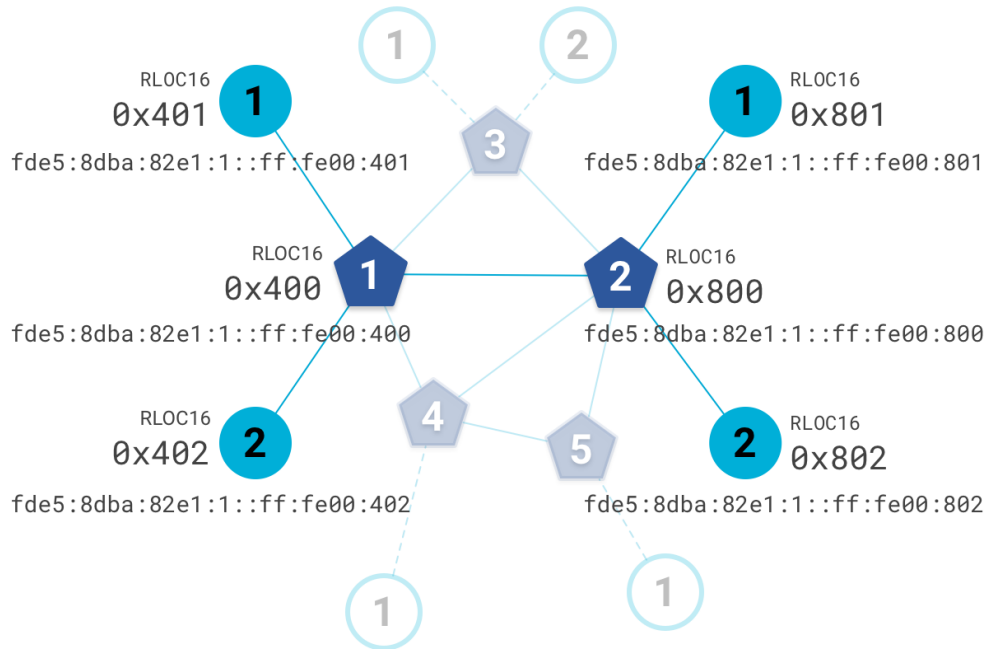


Figure 2.12: OpenThread RLOC16 and Mesh-Local address examples, from [24]

In order to inform multiple devices at the same time, Thread uses Multicast addresses. If they also include SEDs, they are pending on the unicast Mesh-Local prefix, it can therefore vary between different Thread Networks. Reserved Multicast addresses, their scopes and group of reached devices can be seen in Table 2.8

Thread supports Anycast. This type of address is used to route traffic through the

| IPv6 Address | Scope | Addressed group |
|--------------|------------|-------------------|
| ff02::1 | Link-Local | All FTDs and MEDs |
| ff02::2 | Link-Local | All FTDs |
| ff03::1 | Mesh-Local | All FTDs and MEDs |
| ff03::2 | Mesh-Local | All FTDs |

Table 2.8: Reserved Multicast addresses in a Thread Network, from [24]

network to a specific interface if the RLOC of the destination is not known by the sender. This can happen if the network topology changes. This is called the Anycast Locator (ALOC), which identifies the location of multiple interfaces within a Thread network [24]. As by RLOC the last 16 bit of the ALOC represent a specific type or group which should be located. Some of them are listed in Table 2.9.

| ALOC suffix | Type |
|-----------------|-------------------------|
| 0xfc00 | Leader |
| 0xfc01 - 0xfc0f | DHCPv6 Agent |
| 0xfc10 - 0xfc2f | Service |
| 0xfc30 - 0xfc37 | Commissioner |
| 0xfc40 - 0xfc4e | Neighbor Discover Agent |
| 0xfc38 - 0xfc3f | Reserved |
| 0xfc4f - 0xfcff | Reserved |

Table 2.9: Predefined Anycast locator address, from [24]

2.6 Ontologies

In order to provide information about a building's equipment and components, a suitable approach is to use ontologies. Ontologies describe the knowledge of a specific area of expertise with the assistance of formal ordered conceptualities and their relations to each other. They are mainly used to share the knowledge between different applications and services, automatic inferences, their representations and visualization of knowledge. Most components of ontologies are individuals, classes and relations [28] [29] [30].

The individuals, sometimes also called objects, are the basic components of ontologies and include real objects such as a **developer** (person), a **climate system** or an **office room**. Classes or concepts are collections of objects with similar characteristics, comparable to classes in any object orientated programming language. A class might be a facility system, which contains all components such as **heating**, **climate systems** and **entrance systems** for buildings. Another example for a class is a building; individuals of this class are for instance an office room, a conference room and a dressing room. The last component to describe an ontology is the relation.

Relations are used to link objects and define the correlation between themselves and outline properties. In association to building equipment and components **has**, **is part of** and **controls** are examples for relations that can be used to link the objects.

An Ontology is defined by triples, one triple is a set of three entities that describe a statement about semantic data in the form of **subject-predicate-object**. A triple offers the possibility to represent knowledge in a way comprehensive to machines. Some examples of triples are presented in Figure 2.13, for instance, **conference room** is the **subject**, **has a** is the **predicate** of the triple and **entrance system** is the **object** of the described relationship.

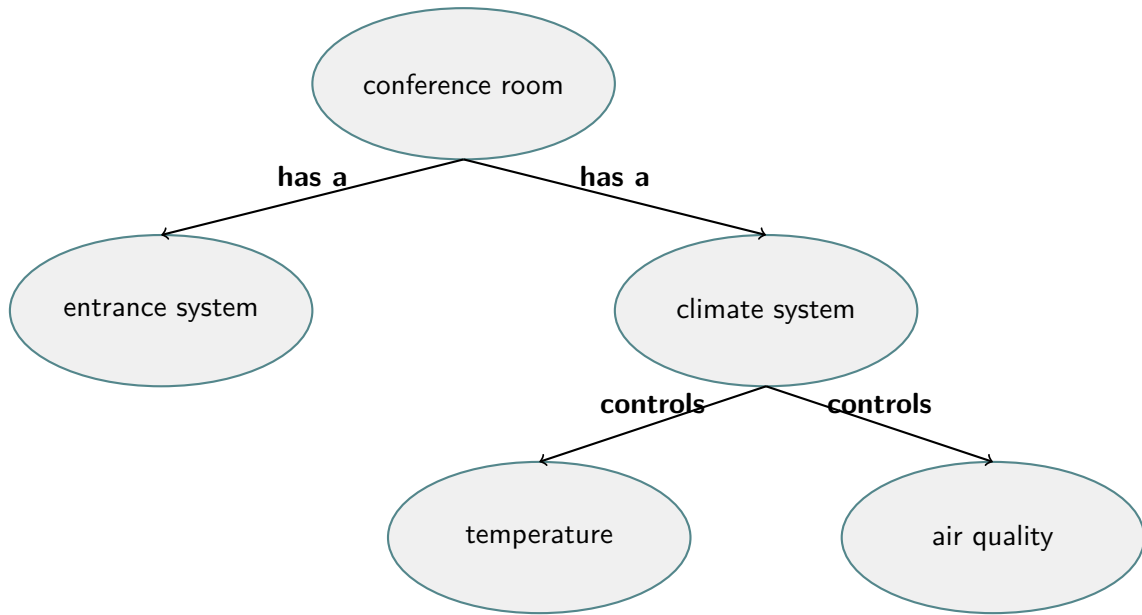


Figure 2.13: Foundational Ontology

A simple ontology, as illustrated in Figure 2.13, is called a basic or foundational ontology. This type allows to formalize general terms of physical objects, properties or disciplines. In order to formalize complex structures, like a company building, a far more domain specific, so called domain ontology, is used to get a detailed description of the facilities and their aspects of the whole building.

Web Ontology Language (OWL)

Ontologies have recently attracted interest due to the semantic web (*initiative*). The semantic web is an idea to make the World Wide Web readable to machines. Therefore, it is necessary to tag the web documents with meta data, which specifies the content of these documents. This metadata is primarily used by search engines and similar applications, with the advantage of higher efficiency and the ability to combine the requested information and generate implicit knowledge. There are several modelling

languages, like the Extensible Markup Language (XML), the Resource Description Framework (RDF) and RDF-S (-Schema), but they are not suitable to express the complex relationships between objects, because of their limited capability of expression. In order to achieve this functionality, a common language to describe the web content is needed. The World Wide Web Consortium (W3C) standardises a knowledge presentation language, the so called Web Ontology Language (OWL). OWL is based on formal logic, which allows inference and generating implicit knowledge. A web document represented by an OWL-ontology is called an OWL-document. Towards making OWL more flexible for different use cases, the W3C introduced three sub-languages OWL Full, OWL DL and OWL Lite with various expression possibilities and scalability. Each sub type of OWL has its advantages and disadvantages. OWL Full allows the usage of all defined OWL-elements and RDF-S elements without limitation; this leads to complex OWL-documents. Due to these difficulties the sub versions OWL DL and OWL Lite were defined.

These two types are limited by elements of speech, rules and other restrictions; OWL Lite only implements the basic elements. Nowadays OWL DL is the most used type of the three variants, because it can describe documents in a more complex way than the Lite version, but it is still computable in finite time compared to the full variant of OWL [31] [32].

Brick

Brick is an open-source effort to provide a standardized description schema for buildings [33]. It introduces standard entities and relationships that are used by applications within that domains. The defined entities and relationships in Brick are described using the semantic web technology, as in the RDF Framework, to create a flexible data model and to integrate this model into existing tools and databases [33]. A model in Brick refers to a RDF data model, which expresses knowledge as a graph with subject-predicate-object tuples, known as triples. Due to the graph representation of the model, it is simple and fast to search entities within the graph, even if they become very large [34].

Methodology

During the creation of this thesis the research method applied follows the design and creation research strategy and uses academic literature research for gathering deep knowledge in relevant topics [36]. The design and creation strategy, in terms of computing research, focuses on developing new IT products, so called artefacts [37]. The artefacts might be constructions, models, methods and instantiations [38] but in most cases are a combination of these artefacts to creates new knowledge. Artefacts are often represented by computer-based products, the design and creation strategy distinguish itself from product development by concentrating on analysis, explanation, argument, justification and critical evaluation of the results.

The design and creation method does not focus on the artefact itself, but on using the artefact as a technique to create new knowledge or on the method to develop an artefact to create knowledge [39]. The applied method is a problem-solving technique and builds upon the principles of system development [40]. This process typically covers five steps: awareness, suggestion, development, evaluation and conclusion. It should be mentioned that these steps are performed in a cyclic way. In most cases, a lot of work goes into investing time in the development step and verifying the system.

This research project focuses on modelling a WSN and using this model to dynamically configure the sensor nodes and add information to the produced data according to this configuration.

This work has been created as a part of the HumBAS research project [41].

Implementation

The WSN is split up in Sensor nodes and the Border Router. The **Sensor nodes** are able to communicate with each other and are connected to the Border Router. Furthermore, some Environmental Sensors are connected to the nodes and are generating data by observing their environment. The **Sensor nodes** toolchain setup and additional functionality of OT is implemented as described in Toolchain Setup.

The Border Router act as **Gateway** between the **Sensor nodes** and the Internet. It allows nodes to join the network, if they have the same Network ID and communication channel and manage them after joining. Moreover, the Border Router provides an MQTT-SN Gateway which forwards the generated data to the application MQTT Broker. Figure 4.1 presents an abstract overview of the involved components and their interconnection.

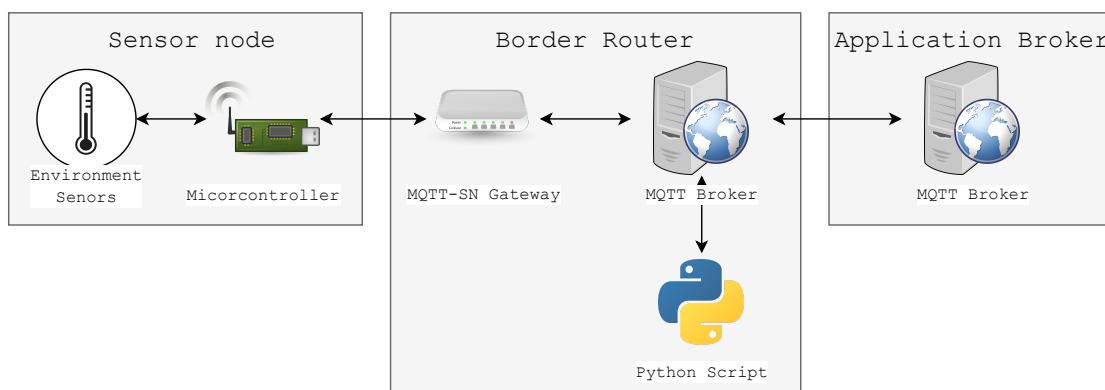


Figure 4.1: Abstract overview of the components and their interconnection

4.1 Sensor nodes

The Sensor node consists of several parts, a Micro Development Kit (MDK) ¹ for IoT Applications, a Base Dock ² and various environmental sensors. The Development Kit uses the Nordic nrf52840 SoC ³ as core processor. It features advanced Bluetooth 5 and the IEEE 802.15.4 standard, which is the basis for other low power wireless connectivity solutions including Zigbee, 6LoWPAN, OT and many more.

The Base Dock comes with four connectors to attach sensors and other peripherals and is powered by an AA battery which also supplies the MDK.

4.1.1 Environmental Sensors

The most important environmental parameters to monitor are ambient temperature, humidity, barometric pressure, air quality and ambient light. All these parameters can be measured by using two sensors, the BME680 and VEML6030.

The BME680 ⁴ is an environmental sensor from Bosch Sensortec, that can measure ambient temperature, relative humidity, barometric pressure and a broad range of gases such as volatile organic compounds (VOC). The sensor is designed for mobile applications and therefore has small footprint and consumes little battery. The latter makes it suitable for battery powered WSN applications.

The Bosch Sensor Application Programming Interface (API) ⁵ is used to communicate with the sensor over the Two Wire Interface (TWI) and to interpret the raw data coming from this interface.

The VEML3060 is a ambient light sensor from Vishay Semiconductors. It features a 16-Bit Analog to Digital-Converter (ADC), can be connected to the TWI and an Interrupt Pin for application specific interaction. Furthermore, it can also measure white portion in the ambient light.

During the course of this work, an API for configuration and simple communication has been developed. The API offers generic read and write functions, which provide all necessary data to communicate with the sensor the microcontroller specific TWI access needs to be implemented. It allows to configure the sensor according to personal needs or use the pre-defined settings, furthermore, the raw data, generated by measurements, is internally converted into meaningful values.

¹<https://wiki.makerdiary.com/nrf52840-mdk/>

²<https://wiki.makerdiary.com/base-dock/>

³<https://www.nordicsemi.com/Products/Low-power-short-range-wireless/nRF52840>

⁴<https://www.bosch-sensortec.com/products/environmental-sensors/gas-sensors-bme680/>

⁵https://github.com/BoschSensortec/BME680_driver (Version 3.5.10)

4.2 Border Router

The Border Router is a part of the OT network and it allows the network to communicate with traditional networks or the Internet. A Raspberry Pi 3+ Model B is used as platform for the Border Router. OT offers an officially supported repository with the necessary code to compile and install it on the Pi. For this work, the pre-build Border Router image (RaspPIoT Border Router Demo, Version 4.1.0-1.alpha) for the Raspberry Pi offered by Nordic Semiconductors is used. After power-up the Border Router activates a graphical web interface to setup a OT network. The **Thread Channel** and **PAN ID** must match those of the sensors for successful communication.

4.2.1 Network Co-Processor

The Network Co-Processor (NCP) acts as connectivity chip for any device which does not support IEEE 802.15.4 or is not able to execute compiled binaries of OT. It allows its host device to communicate with other network members just like a native Thread device. In this work, a **nRF52840 Dongle** from Nordic Semiconductors is programmed with the Thread NCP/RCP Example of the Software Development Kit (SDK) (nRF5 SDK for Thread and Zigbee Version 4.0) and plugged into one USB port of the Border Router.

4.2.2 MQTT-SN Gateway

In this work, application data is exchanged using the lightweight protocol MQTT-SN. In order to evaluate this data, it has to be sent to an MQTT Broker, therefore an MQTT-SN Gateway is used. When using the provided image from Nordic Semiconductors, the **paho.mqtt-sn.embedded-c** Gateway has already been installed.

Since OT transmits data over UDP version 6, the following parameters are relevant for communication. The **GatewayUDP6Broadcast** is used for **Gateway** search and has the default address **FF03::1** and the **GatewayUDP6Port** by default is **47193**. In order to connect to the correct MQTT Broker, the **BrokerName** should be set to **localhost**, because of security issues see section MQTT Broker.

4.2.3 MQTT Broker

An everyday topic in computer science and application development is security. Therefore, the communication between gateway and the application MQTT Broker is secured using Transport Layer Security (Version 1.2) (TLS (V1.2)). Unfortunately, the **paho.mqtt-sn.embedded-c** does not support TLS (V1.2). Because of this, a second MQTT Broker is setup on the Raspberry and bridged to the application MQTT Broker.

Python Script for Data transformation

The MQTT-SN API provided by the SDK allows to send data byte wise only. This means data has to be split up in single bytes and sent over the network to the **Gateway**.

In order to make the data from **Sensor nodes** adaptable for other data on the Broker and prepare them for further processing, a Python script is used for transformation. This script subscribes to all **Topics** coming for the sensor network, converts it to floating point representation and publishes it using the same **Topic** or another predefined one.

4.3 Toolchain Setup

For programming the **Sensor nodes**, some additional programs and the recompiled OT libraries are required.

4.3.1 Toolchain

The toolchain requires the GNU Arm Embedded Toolchain, GNU make, pyOCD, a SDK for the MDK, the SDK for the Nordic SoC and the Nordic Command Line Tools.

The GNU Arm Embedded Toolchain and GNU make are used for compiling and linking the source code.

pyOCD is an open source Python implementation of the On-Chip Debugger (OCD), which is used for programming and debugging Arm Cortex-M microcontrollers and supports multiple types of USB debug probes. To install pyOCD, the Python package-management system can be used or it can be installed using the repository on GitHub ⁶, this has been used within this work.

Due to the custom Design of the MDK, the manufacturer offers some examples, pin mappings and linker files in an additional repository ⁷.

For the main functionality of the microcontroller, the official SDK (nRF5 SDK for Thread and Zigbee Version 4.0, including support for SoftDevice 140) from Nordic needs to be downloaded and copied in the subfolder `nrf_sdks` of the repository. Afterwards, the GNU toolchain root and version within the file `<SDK>/components/toolchain/gcc` need to be adopted according to the installation above. There are different file extensions for different operating systems.

The last step is to download Nordics Command Line Tools ⁸ in order to combine the hex file of the SoC and the SoftDevice, which is responsible for Bluetooth applications.

4.3.2 Compiling OpenThread Libraries

The libraries are compiled under Linux. The Nordic DevZone provides guides and further information on compiling using Windows or general questions.

⁶<https://github.com/pyocd/pyOCD> (Version 0.24)
⁷<https://github.com/makerdiary/nrf52840-mdk> (Commit 23d86f9)
⁸<https://www.nordicsemi.com/Software-and-tools/Development-Tools/nRF-Command-Line-Tools/Download> (Linux 64-Bit Version 10.10.0)

Although the cloned repository offers compiled OT libraries, it is good practise to compile OT again to get the newest version. The repository which contains the source code can be downloaded from the OT GitHub page ⁹. After cloning the repository, the OT environment needs to be set up. This can be done by executing the bootstrap script in the root folder. OT offers a Command Line Interface (CLI) which runs over the Universal Synchronous/Asynchronous Receiver/Transmitter, allowing direct interaction with the Thread network and its settings. Furthermore, the CLI offers possibilities to send debug messages from the microcontroller to the host using the Universal Synchronous/Asynchronous Receiver/Transmitter. In order to enable the CLI on the Sensor node, the following lines of code (see Listing 4.1), the pins for communication need to be changed in the file `openthread/examples/platforms/nrf52840/platform-config.h`. For further information, visit the Makerdiary website linked above.

```

1  /**
2   * UART Hardware Flow Control.
3   *   NRF_UART_HWFC_ENABLED - HW Flow control enabled.
4   *   NRF_UART_HWFC_DISABLED - HW Flow control disabled.
5   */
6  #ifndef UART_HWFC
7  #define UART_HWFC NRF_UART_HWFC_DISABLED
8  #endif
9
10 /**
11  * UART TX Pin.
12  */
13 #ifndef UART_PIN_TX
14 #define UART_PIN_TX 20
15 #endif
16
17 /**
18  * UART RX Pin.
19  */
20 #ifndef UART_PIN_RX
21 #define UART_PIN_RX 19
22 #endif

```

Listing 4.1: Change pins of USART

Most of the peripherals can be controlled using OT as API. In order to control them, the OT source code needs to be adapted in the following way. First, function declarations need to be added somewhere in the file `examples/platforms/openthread-system.h` (see Listing 4.2).

```

1  /* Declare functions for LED module. */
2  void otSysLedInit(void);
3  void otSysLedSet(uint8_t aLed, bool aOn);
4  void otSysLedToggle(uint8_t aLed);

```

Listing 4.2: Declarations for LED functionality

⁹<https://github.com/openthread/openthread> (commit 8a1992e)

Furthermore, a file named `gpio.c` should be created and moved to the `examples/platform-s/nrf528xx/src` directory. The content of the created file can be seen in Listing 4.3. Listing 4.3 contains the pin definitions of the user programmable LEDs of the MDK and headers for low-level functions of the nRF52840. In addition, there are functions to initialize, set/reset and toggle the LEDs.

```
1  /**
2   * @file This file implements the system abstraction for GPIO and GPIOTE.
3   *
4   */
5   #define GPIO_LOGIC_HI 0
6   #define GPIO_LOGIC_LOW 1
7
8   #define LED_GPIO_PORT 0x50000300UL
9   #define LED_1_PIN 22 // pin of green Led
10  #define LED_2_PIN 23 // pin of red Led
11  #define LED_3_PIN 24 // pin of blue Led
12
13  /* Header for the functions defined here */
14  #include "openthread-system.h"
15
16  #include <string.h>
17
18  /* Header to access an OpenThread instance */
19  #include <openthread/instance.h>
20
21  /* Headers for lower-level nRF52840 functions */
22  #include "platform-nrf5.h"
23  #include "hal/nrf_gpio.h"
24  #include "hal/nrf_gpiote.h"
25  #include "nrfx/drivers/include/nrfx_gpiote.h"
26
27  /**
28   * @brief Function for configuring: PIN_OUT pin for output
29   */
30  void otSysLedInit(void) {
31
32      /* Configure GPIO mode: output */
33      nrf_gpio_cfg_output(LED_1_PIN);
34      nrf_gpio_cfg_output(LED_2_PIN);
35      nrf_gpio_cfg_output(LED_3_PIN);
36
37      /* Clear all output first */
38      nrf_gpio_pin_write(LED_1_PIN, GPIO_LOGIC_LOW);
39      nrf_gpio_pin_write(LED_2_PIN, GPIO_LOGIC_LOW);
40      nrf_gpio_pin_write(LED_3_PIN, GPIO_LOGIC_LOW);
41  }
42
43  /**
44   * @brief Function to set the mode of an LED.
45   */
46  void otSysLedSet(uint8_t aLed, bool aOn) {
47
```



```

48     switch (aLed){
49         case 1:
50             nrf_gpio_pin_write(LED_1_PIN, (aOn == GPIO_LOGIC_HI));
51             break;
52         case 2:
53             nrf_gpio_pin_write(LED_2_PIN, (aOn == GPIO_LOGIC_HI));
54             break;
55         case 3:
56             nrf_gpio_pin_write(LED_3_PIN, (aOn == GPIO_LOGIC_HI));
57             break;
58     }
59 }
60
61 /**
62  * @brief Function to toggle the mode of an LED.
63  */
64 void otSysLedToggle(uint8_t aLed){
65
66     switch (aLed){
67         case 1:
68             nrf_gpio_pin_toggle(LED_1_PIN);
69             break;
70         case 2:
71             nrf_gpio_pin_toggle(LED_2_PIN);
72             break;
73         case 3:
74             nrf_gpio_pin_toggle(LED_3_PIN);
75             break;
76     }
77 }

```

Listing 4.3: Content of file gpio.c

After creating an API for the LEDs and changing the Universal Synchronous/Asynchronous Receiver/Transmitter pins, the compile command of Listing 4.4 in the root folder of the repository needs to be run in order to apply the changes and create new libraries.

```

1  make -f examples/Makefile-nrf52840 BORDER_AGENT=1 BORDER_ROUTER=1 COAP=1
    COMMISSIONER=1 DISABLE_BUILTIN_MBEDTLS=1 DNS_CLIENT=1 DIAGNOSTIC=1
    EXTERNAL_HEAP=1 JOINER=1 LINK_RAW=1 MAC_FILTER=1 MTD_NETDIAG=1
    SERVICE=1 UDP_FORWARD=1 ECDSA=1 Sntp_CLIENT=1 COAPS=1 DHCP6_SERVER=1
    DHCP6_CLIENT=1

```

Listing 4.4: Command to compile OpenThread Library

Afterwards, the created binaries and header files need to be copied in the folders of the SDK, which is done by:

- Copying the libraries from: `openthread/output/nrf52840/lib` to the SDK folder: `/external/openthread/lib/nrf52840/gcc`.

- Copying the openthread/include folder to /external/openthread.
- Copying the platform specific platform-fem.h and platform-softdevice.h files from: /external/project/openthread/examples/platforms/nrf528xx/src to /external/openthread-include/platform.
- Copying openthread-system.h file from: /external/project/openthread/examples/platforms to the /external/openthread/include/platform folder.

4.4 Brick Model

Brick is used to model the WSN. Based on this model, configuration files for **Sensor nodes** and MQTT-SN Gateways are generated.

4.4.1 Creation process of a Model

A Brick model is represented as graph, this means physical components, virtual objects and parameter settings are represented as vertices and relations between them are illustrated as edges.

Brick models are written using the RDF framework, writing a model is very prone to errors as typos can occur. In order to avoid these errors, a Python script is used to create the model, which allows to define reusable functions for creation nodes, **Gateways** or other objects. In Listing 4.5, a simple example of a **Sensor node** creation function can be seen. For creating a **Sensor node** two functions are necessary, first to create the node itself and then modelling the configuration and linking it to the node. These functions are written to keep the creation process as simple as possible, the function body creates the well known ontology triples. For example, the nodes **Microcontroller** and **Universally Unique Identifier (UUID)** and their relation **has_UUID**.

After the creation of a **Sensor node**, the function `create_wsn_node` is called. The first parameter is the variable name addressing the node, which is needed for the configuration. Afterwards, the UUID for node and **Gateway** are handed over to the function. The **Gateway** UUID is only used for presentation purpose only for showing, which subnet the node can be found. The remaining boolean parameters determine which sensors are connected to the node and on which MQTT-SN **Topics** the node will be published.

After creating a **Sensor node** configuration starts, therefore the function `create_wsn_node_configuration` is used. First, the functions needs to know to which node the configuration belongs to. Afterwards, the function takes parameter to configure the nodes behaviour and the networks security credentials.

Theses parameters are:

- Sensor readout duration (in minutes)
- OpenThread Network security credentials

- Thread Network PAN ID
- Thread Communication Channel
- MQTT-SN Communication Port

```

1 # ----- create node, configuration and sensors -----
2 # Node 1
3 create_wsn_node("Office1_Node", "Node_101", mqtt_sn_gateway1, True, True,
4               True, True, True)
5 create_wsn_node_configuration("Office1_Node", 1, thread_pan_id,
6                             thread_channel, mqtt_sn_port)

```

Listing 4.5: Code to create node and configuration

The code of Listing 4.5 creates the model which is shown in Figure 4.2, some of its nodes and edges are hidden to simplify the representation. Exemplarily, nodes within the red and blue rectangles are all modelled as sensors, and the MQTT-SN Topic and unit of the measured value can be seen in the red rectangle. Nodes surrounded in green are configuration items, and the UUID is modelled in the orange box, that connects to the center node, which is the root node in this example representing the Sensor node itself.

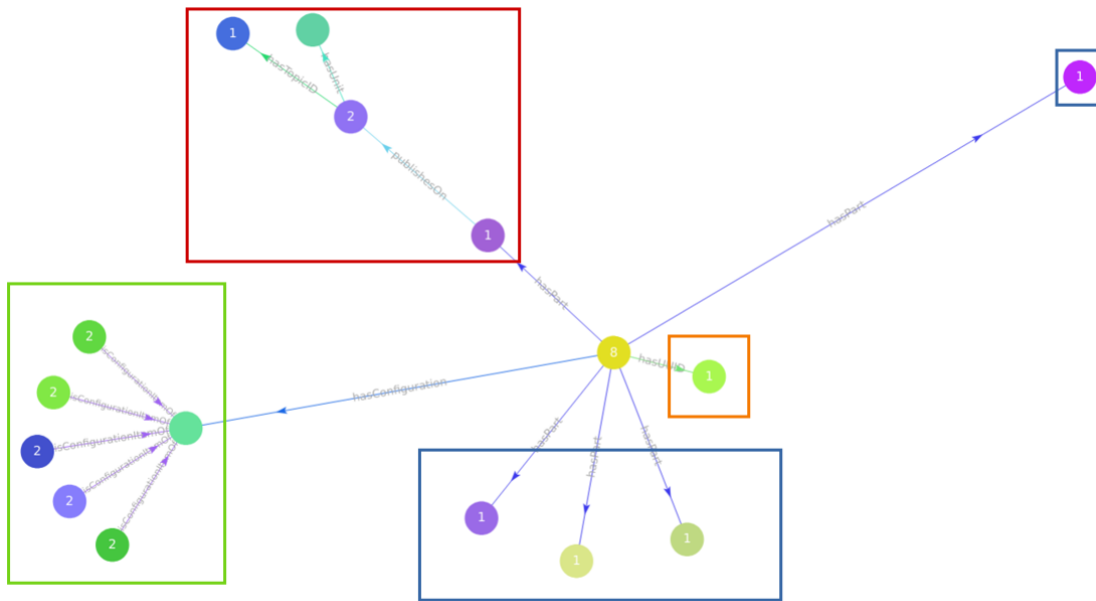


Figure 4.2: Brick model of a Sensor node

4.4.2 Creating a configuration file

The model is mainly used to keep track of all configuration parameters, therefore also configuration files for all components are generated based on the model. To query for settings the query language for RDF, SPARQL is used.

An example query can be seen in Listing 4.6, which shows how to find the UUID of a node. In this example, the UUID of `Office1_node` is found and stored in the variable `node_uuid_q`.

```
1 node_uuid_q = graph.query(  
2     """SELECT ?uuid ?value WHERE {  
3         office:""" + node_name + """ humbas:hasUUID ?uuid.  
4         ?uuid humbas:hasValue ?value  
5     }""")
```

Listing 4.6: Query UUID of Office1 Node

For **Sensor nodes** a single C header file is generated. This file contains all symbolic constants to access the LEDs, communication parameters and the MQTT-SN topic configurations. Listing 4.7 shows the generated configuration of the model defined in Listing 4.5. A similar file is generated for the MQTT-SN Gateway.

```
1  /* Configuration file for Office1_Node */  
2  
3  /* Device Name */  
4  static char DeviceName[] = "Node_101";  
5  
6  /* Symbolic LED Constants */  
7  #define GREEN 1  
8  #define RED 2  
9  #define BLUE 3  
10  
11 /* Sensor Parameters */  
12 /* Thread Communication Channel */  
13 #define THREAD_CHANNEL 15  
14 /* Readout Duration of the Sensors */  
15 #define READ_SENSOR_DURATION APP_TIMER_TICKS(60000)  
16 /* Maximal Topics */  
17 #define MAX_TOPICS 6  
18 /* Thread Network ID */  
19 #define THREAD_PANID 43981  
20 /* MQTT-SN Communication Port */  
21 #define MQTTSN_PORT 47193  
22  
23 /* Sensors connected to the Node */  
24 /* Sensor BME680 */  
25 #define BME680  
26 /* Sensor VEML6030 */  
27 #define VEML6030  
28  
29 /* Symbolic Topic Constants */  
30 #define BATTERY 0  
31 #define TEMPERATURE 1  
32 #define HUMIDITY 2  
33 #define PRESSURE 3  
34 #define CO2 4  
35 #define AMBIENTLIGHT 5  
36
```

```

37  /* Struct with Topic Names */
38  static char topic_names[][25] = {
39      {"wsn/Node_101/batteylevel"},
40      {"wsn/Node_101/temperature"},
41      {"wsn/Node_101/humidity"},
42      {"wsn/Node_101/pressure"},
43      {"wsn/Node_101/co2"},
44      {"wsn/Node_101/ambientLight"},
45  };
46
47  /* Struct to register Topics at the Gateway */
48  static mqttsn_topic_t topics[6] = {
49      {
50          .p_topic_name = (unsigned char *) topic_names[BATTERY],
51          .topic_id = 0,
52      },
53      {
54          .p_topic_name = (unsigned char *) topic_names[TEMPERATURE],
55          .topic_id = 1,
56      },
57      {
58          .p_topic_name = (unsigned char *) topic_names[HUMIDITY],
59          .topic_id = 2,
60      },
61      {
62          .p_topic_name = (unsigned char *) topic_names[PRESSURE],
63          .topic_id = 3,
64      },
65      {
66          .p_topic_name = (unsigned char *) topic_names[CO2],
67          .topic_id = 4,
68      },
69      {
70          .p_topic_name = (unsigned char *) topic_names[AMBIENTLIGHT],
71          .topic_id = 5,
72      }
73  };

```

Listing 4.7: Configuration of Office1 Node

4.5 Proof of Concept

In order to proof the systems concept, a single **Sensor node** and one **Gateway** are used from the WSN. The proof of concept foresees that a **Sensor node** publishes temperature data over the WSN by processing it on an Application Server, which hosts an MQTT Broker. In the following part, a short description of the **Sensor node** software with additional screenshots of the output during operation is given. Furthermore, screenshots of the clients verifying published message arrival on the MQTT Broker are included. The Figure 4.3 illustrates the sample WSN and the information flow from the temperature sensor to the Application MQTT Broker, which is not part of the WSN.

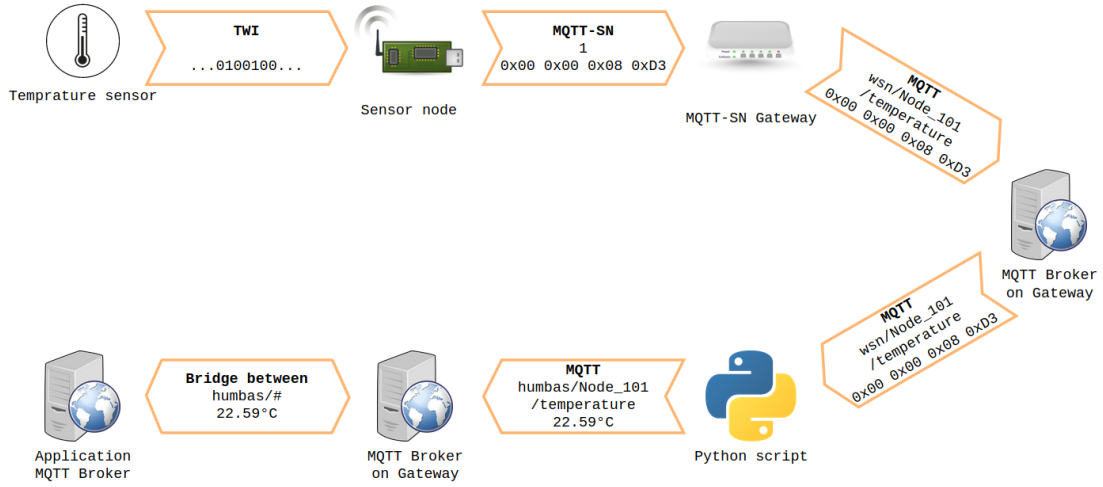
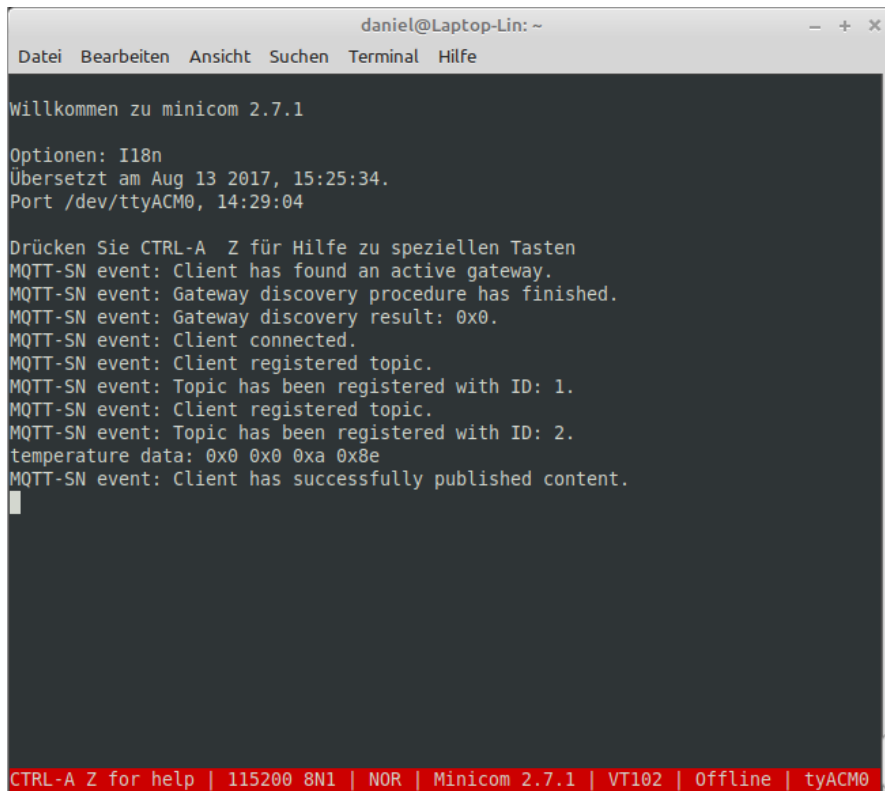


Figure 4.3: Flow of information from sensor to Application MQTT Broker

During the initialization a **Sensor node** scans the environment for an already existing OT network. If there exists a network, the node will request to join this network, and after a successful join process, all connected sensors will be configured. Then, the node searches for an **MQTT-SN Gateway** within the network and tries to connect to the nearest one. This process is determined by the minimum number of hops it takes to reach the **Gateway**. After signing up, the node registers all configured **MQTT-SN Topics**. If the previous two steps are executed successfully, the **Sensor node** will continuously read data from the sensors and publish it under predefined topics. The output of a **Sensor node** of the described procedure can be seen in Figure 4.4.

The Figure 4.5 illustrates the communication process between the components. Messages using MQTT are printed in blue and MQTT-SN messages are coloured in orange. First the **Python Script** connects the **MQTT Broker** on the **Gateway** (messages 1 and 2) and subscribes, for instance, the topic **SensorNode1/temperature** (messages 7 and 8). Further, the **Sensor node** sends an **MQTT-SN Connect** message to the **Gateway**, which translates it into an **MQTT** message and forwards it to the **Broker** (messages 3 to 6). If the **Sensor node** has read the register data of the temperature sensor, it publishes the data on the Topic **SensorNode1/temperature** (messages 9 to 11 and 13). The **MQTT Broker** processes this message and publishes it to all clients that are subscribed to this Topic (message 12). In this case, the **Python script** receives the temperature data, converts it from HEX-representation to decimal-representation, changes the Topic and publishes it back to the **Broker** (messages 14 and 15). The bridge communication between **Broker on the Gateway** and the **Application Broker** is not shown, since every Topic that matches the bridge configuration is mirrored to the respective other **Broker**.

The **Sensor nodes** publish the data to the **Gateway**, where the Topic is mapped from an **MQTT-SN** to a full **MQTT** Topic and then forwarded to the **MQTT Broker**, locally



```
daniel@Laptop-Lin: ~
Datei Bearbeiten Ansicht Suchen Terminal Hilfe

Willkommen zu minicom 2.7.1

Optionen: I18n
Übersetzt am Aug 13 2017, 15:25:34.
Port /dev/ttyACM0, 14:29:04

Drücken Sie CTRL-A Z für Hilfe zu speziellen Tasten
MQTT-SN event: Client has found an active gateway.
MQTT-SN event: Gateway discovery procedure has finished.
MQTT-SN event: Gateway discovery result: 0x0.
MQTT-SN event: Client connected.
MQTT-SN event: Client registered topic.
MQTT-SN event: Topic has been registered with ID: 1.
MQTT-SN event: Client registered topic.
MQTT-SN event: Topic has been registered with ID: 2.
temperature data: 0x0 0x0 0xa 0x8e
MQTT-SN event: Client has successfully published content.

CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7.1 | VT102 | Offline | tyACM0
```

Figure 4.4: Output of Sensor node during initialization and operation

running on the Gateway. The received MQTT Message can be seen in Figure 4.6.

Since the data is decoded in HEX-format a Python script is used to convert it into floating point representation. Furthermore it changes the top level name from `wsn/` to `humbas/` and publishes the message back to the MQTT Broker on the Gateway. The converted topic arrived on the Broker can be seen in Figure 4.7.

Lastly, all topics starting with `humbas/` are available on the Application Broker, because of the bridging between this and the MQTT Broker running on the Gateway. Figure 4.8 shows the topic `humbas/Node_101/temperature` published to the Application Broker.

4. IMPLEMENTATION

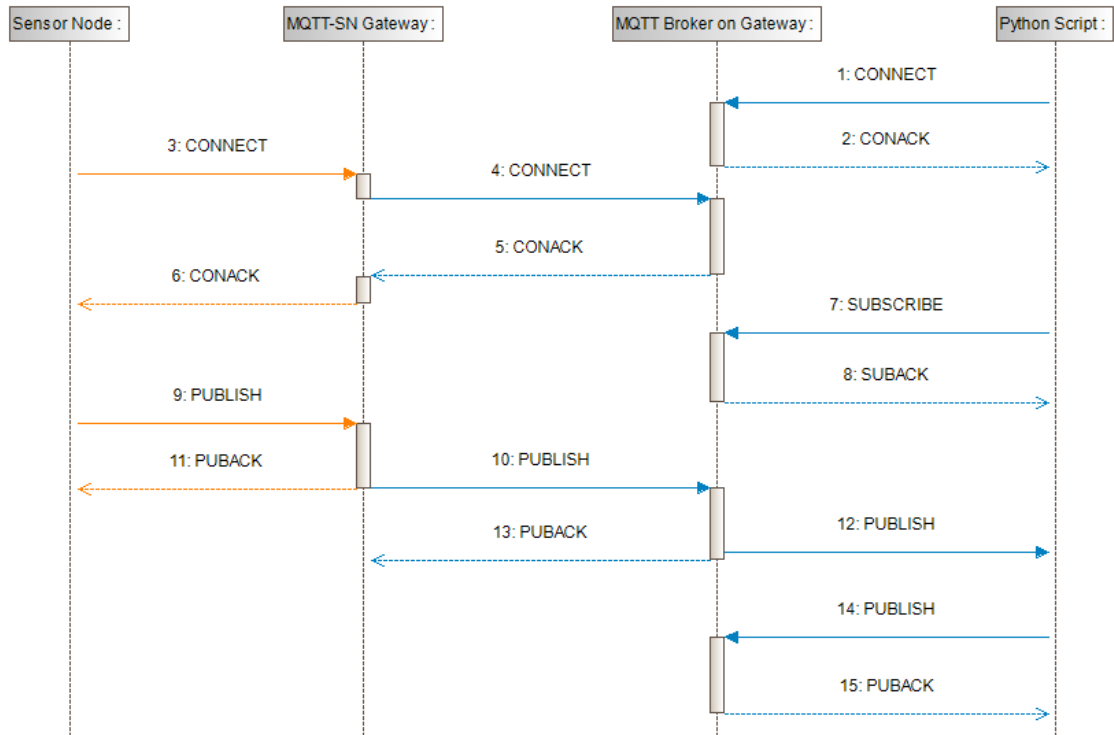


Figure 4.5: Sequence diagram of communication in the WSN

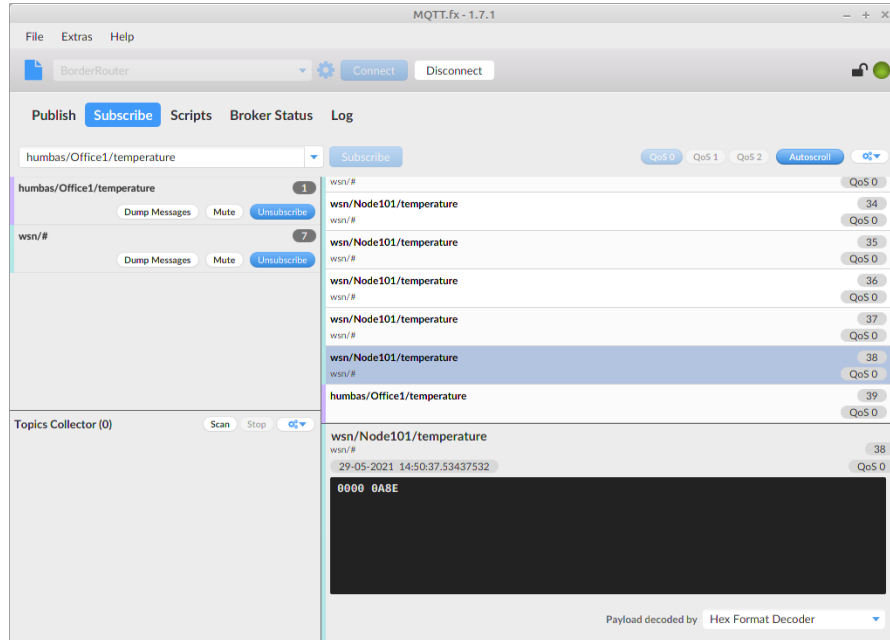


Figure 4.6: MQTT Broker on the Gateway; published temperature from the Sensor node

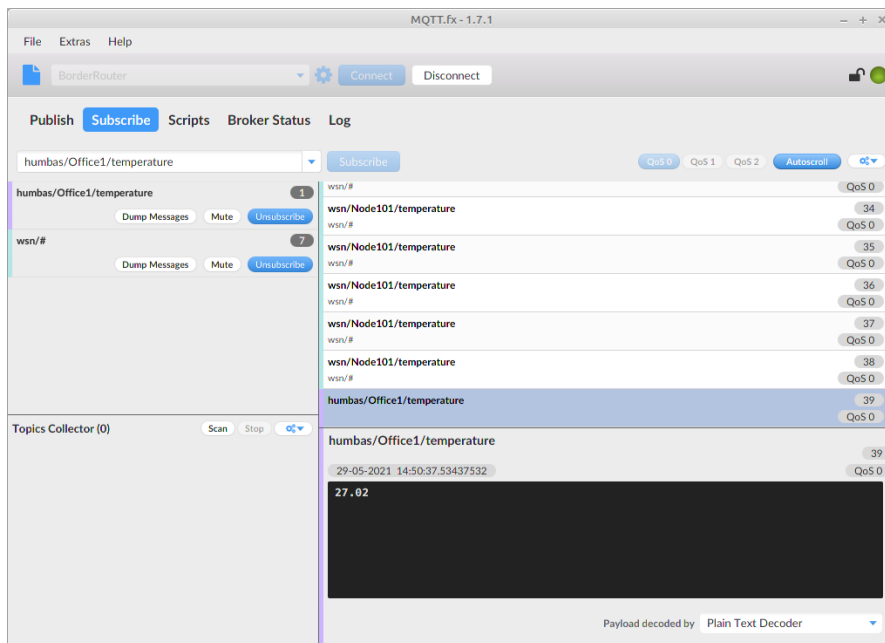


Figure 4.7: MQTT Broker on the Gateway; published temperature from the Python script

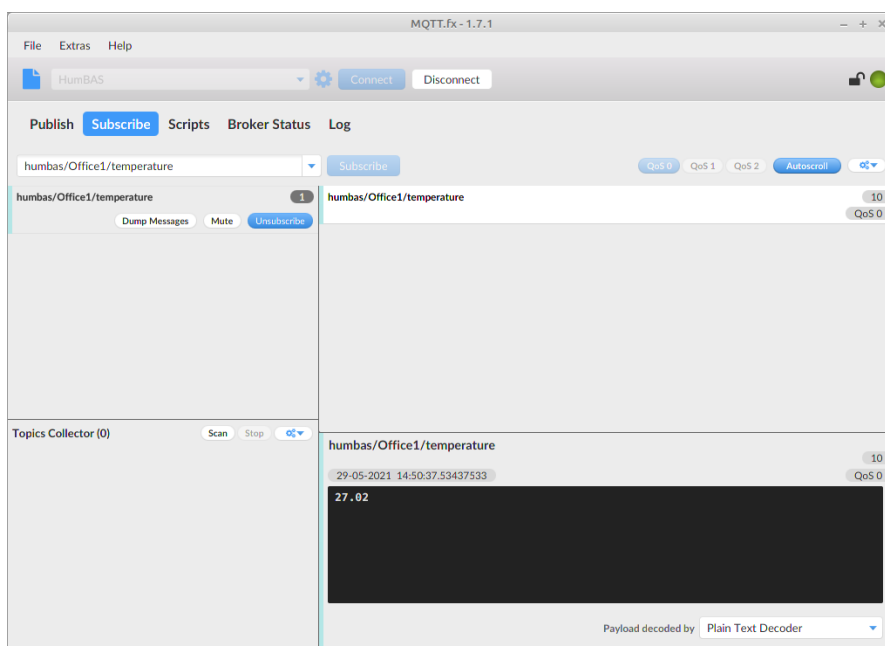


Figure 4.8: Application MQTT Broker; bridge temperature topic from MQTT Broker on the Gateway

Results

This chapter presents the results to answer the formulated research questions in Chapter 1. Based on the first research question and the results of the literature survey presented in Section 2.1, a table has been created that lists possible points for data aggregation. Regarding the second research question, Section 5.2 proposes different modelling concepts for a WSN with Gateway and MQTT Broker.

5.1 Possible points for data aggregation

Most sensor nodes in a WSN have a limited lifespan because they run out of power. In some cases, the power provider, e.g. the battery, can be exchanged and the node's lifespan is increased. However, to keep the lifetime as long as possible the sensor nodes use lightweight protocols to transmit their data and include necessary data only in the messages sent, for instance, a temperature sensor would transmit the current temperature value. For some applications, this might be enough information for further processing, but others might need more information in the upper hierarchical layers, therefore data needs to be aggregated. This aggregation should not impact the lifespan of the component, also it should not generate much computational overhead for the component adding the data.

Table 5.1 shows possible components of an example application infrastructure, including a WSN, an **Application Broker**, **Application Software** and a **Client Software**, using the produced data coming in from the **Sensor nodes**. The header of the table describes the data that should be added, whereas the symbol **X** marks components of the infrastructure at which this data is available. The symbol **O** marks components that might be able to determine or request this data. For example, the **Application Broker** can request the location of measurement from the **Gateway**. It then adds it to the measurement-data. A – symbol in the entry means the component cannot determine or request the needed data for aggregation.

| | Unit predefined/not predefined | Location |
|----------------------|--------------------------------|----------|
| Sensor node | X/X | X |
| Gateway | X/O | X |
| Application Broker | X/O | O |
| Application Software | X/O | O |
| Client Software | X/- | - |

Table 5.1: Possible points in a WSN for data aggregation of Unit and Location

The transformation of the message’s content and its Topic while travelling from the source to the sink can be seen in Table 5.2. As first step, the data comes in binary format and without any Topic from the sensor. The **Sensor node** converts it into HEX-format and transmits it to the **Gateway**, using MQTT-SN and the Topic identifier 1, which corresponds to **wsn/Node101/temperature**. The **Gateway** converts the HEX-formatted temperature value into decimal numbers and changes the Topic from **wsn/** to **humbas/** also, it exchanges the node’s identifier with the corresponding location in the building. For example, if **Sensor node 101** is placed in **Office 1**, then the identifier would be **Node101** and is exchanged with the location **Office1**. Afterwards, the **Gateway** publishes the temperature on the **Application Broker**, which provides the data for the **Application Software** or the **Client Software**.

| Points in the WSN | Message content | Message Topic |
|----------------------|--------------------|-----------------------------------|
| Sensor | ...10011... | - |
| Sensor node | 0000 0A8E | 1 |
| Gateway | 27.02 | wsn/Node101/temperature |
| Application Broker | 27.02 °C | humbas/Office1/temperature |
| Application Software | 27.02 °C | humbas/Office1/temperature |
| Client Software | 27.02 °C | humbas/Office1/temperature |

Table 5.2: Transformation of message content and Topic from source to sink

5.2 Visualization sensor data using a graph environment

The whole system should be modelled so that it represents all necessary meta-data, relations and configurations in a structured way. It should provide an overview of all the components and their interaction. The model should also keep track of all application relevant parameters, for example, the Topic IDs for the MQTT-SN Topics and the configuration parameters, for example, the Broker IP Address. If one of these parameters changes, it is easy to figure out which component needs to be changed.

The following models illustrate feasible representations of the system, but are limited to a single microcontroller instance measuring temperature, pressure, humidity and publishing it to an MQTT Broker.

The node uses MQTT-SN to send the sensor data to the MQTT-Broker. Therefore, Topics are needed, since all Topics are managed by a Sensor nodes are combined under the top class Topics. This separates the representation of application data, for example, the temperature from node meta-data. A Topic in MQTT-SN can be published by using a unique integer value, because of this every Topic has an ID. In order to avoid errors by computing the sensor data, the unit of the data is appended to the Topic. The node's meta-data includes all system parameters, for example, the Gateway Port and the UUID. Node names of the following models which are ending with a *-symbol, are required to match for a correct working WSN.

Model 1

In Figure 5.1, the first approach for modelling the WSN can be seen. The model shows an MQTT Broker, its Broker IP Address and its communication partner, an MQTT-SN Gateway. The MQTT-SN Gateway's successors are the Gateway Port and the Broker IP Address, which are both system parameters the Gateway needs to know for correct operation, and a Microcontroller. The Sensor nodes, which are the main part of the WSN, consist of a Microcontroller and Sensors which are connected to it. This is also represented in the model. Furthermore, all system- and application-related parameters are also direct successors of the Microcontroller.

Model 2

In this model, all changeable configuration parameters and the Topics, for example of the microcontroller, are combined by an extra node called Configuration. All Sensors are directly connected to the Microcontroller node. Due to this extra node, software-related parts are summarized and connected to the node Microcontroller as one edge in the graph and hardware parts are still directly connected, because of their physical link to the Microcontroller. In case of hardware parts, the connection means this component is exchanging data, which is later stored or processed by a higher layer or an external not modelled component. For a graphical representation, see Figure 5.2.

Model 3

In the third model, the MQTT-SN Gateway and the MQTT Broker setup stay the same as in the models presented earlier. In this model, the UUID of the Microcontroller within the network is directly linked to Microcontroller node in the graph. The Configuration sub-graph is structured in the system- and application parameters and the MQTT-SN Topics are bundled by a top level named Topics. Exemplary, a Configuration sub-graph means a graph that contains the nodes Configuration, Gateway Port, Location and UUID. Topics combines all MQTT-SN Topics the Microcontroller publishes data to.

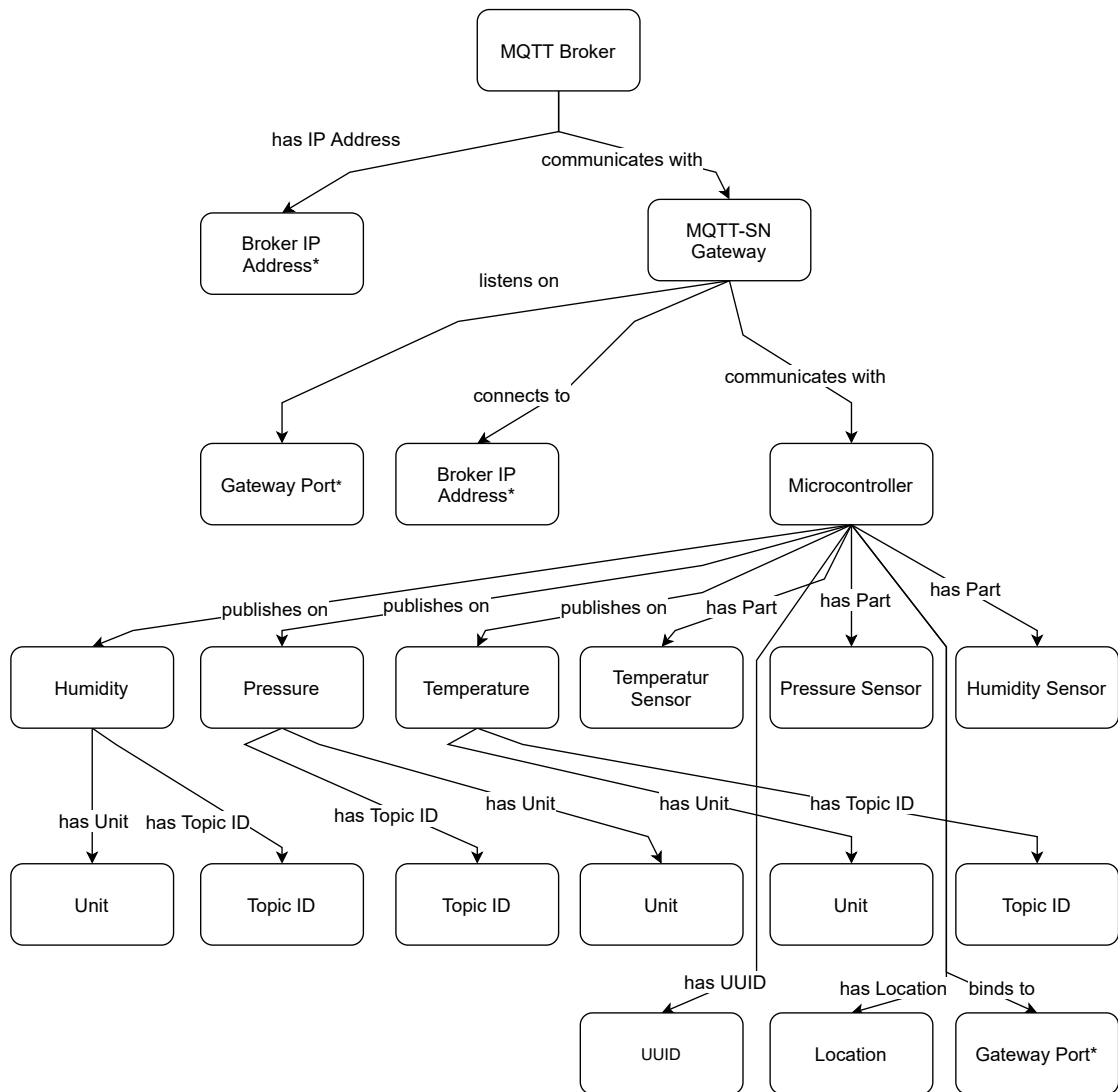


Figure 5.1: Wireless Sensor Network Model 1

Model 4

The fourth model can be seen in Figure 5.4 and shows a Configuration sub-graph, which concentrates all system relevant parameters at a single point in the model. The Topics are now children of the corresponding Sensors. This means the Temperature Sensor has an edge to the Topic Temperature where it published collected data. The setup of MQTT-SN Gateway and MQTT Broker remains the same as in the models presented previously.

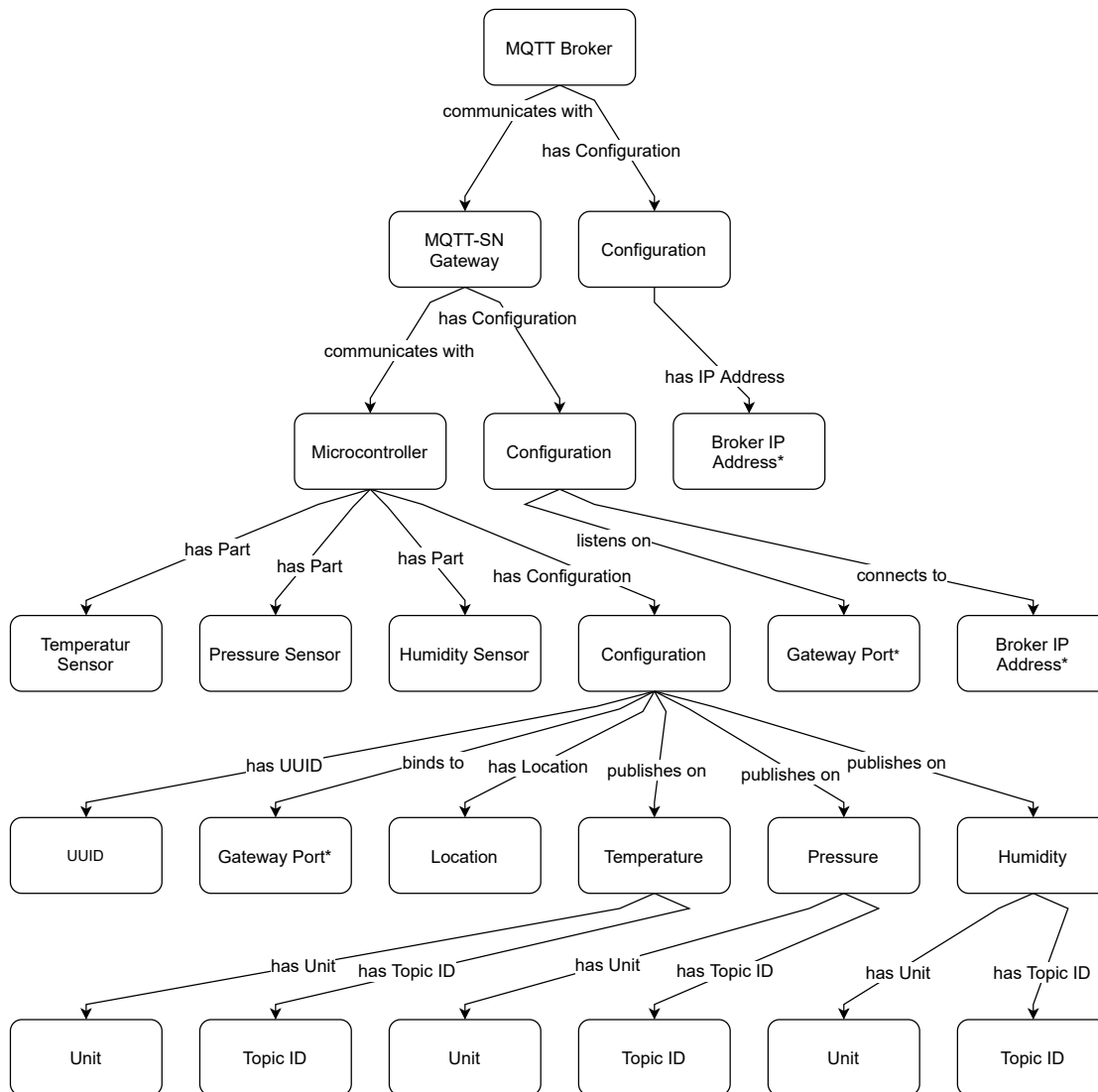


Figure 5.2: Wireless Sensor Network Model 2

Final Model

This model has been created after an analysis of the drawbacks and advantages of the presented models previous. Model 3 and Model 4 are combined, in order to create this modelling approach, which best meets the presented requirements. Chapter 6 presents a detailed description of the benefits and disadvantages of each model and states why this fifth model is needed.

The fifth model, also known as the final model, is shown in Figure 5.5. As a top instance an MQTT Broker is modelled, it has two children nodes, one of them being the related Configuration sub-graph and the other being an MQTT-SN Gateway, which is a

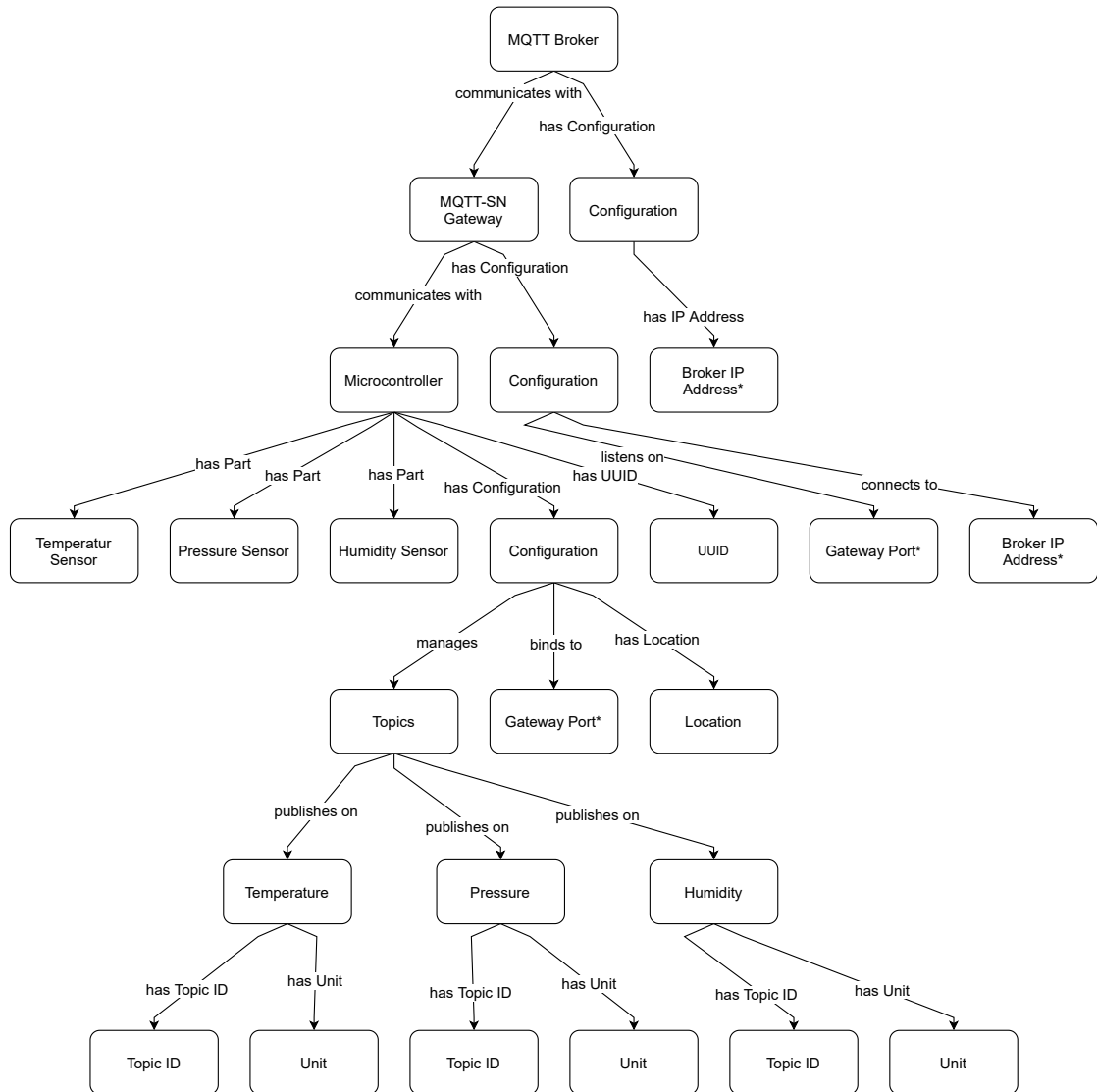


Figure 5.3: Wireless Sensor Network Model 3

communication partner of the broker. The number of children the MQTT-SN Gateway has highly depends on the total number **Sensor nodes**. Inside the model, the **UUID** is a direct successor of the node **Microcontroller**, and further, the **Configuration** sub-graph contains all remaining system parameters. The **Sensors**, which are components of the **Sensor node**, are directly connected to the node **Microcontroller** and the **Topics** are successors of the sensors.



Figure 5.4: Wireless Sensor Network Model 4

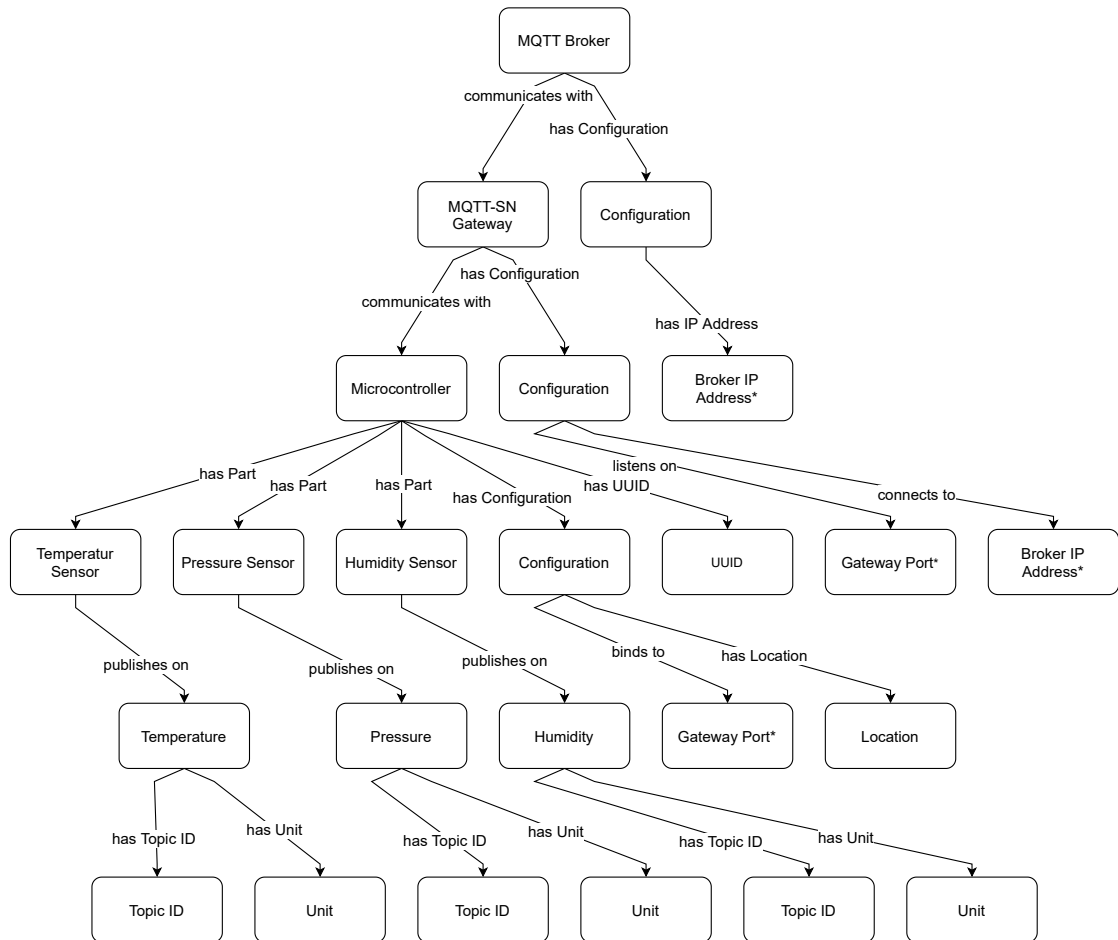


Figure 5.5: Wireless Sensor Network Model 5

Discussion

This chapter discusses the results presented in Chapter 5. An analysis to determine the most suitable point for data aggregation in a big application infrastructure, including a WSN, in Section 6.1 will be given first. In Section 6.2, the presented modelling approaches are evaluated regarding the overview, readability and reuseability of defined sub-graphs. Finally, the discussion in Section 6.4 answers the research question formulated in chapter 1.

6.1 Analysing points for data aggregation

Table 5.1 shows possible locations for data aggregation in an example application infrastructure. In order to keep the computation overhead as low as possible the aggregation should be done as early as possible. In addition, the aggregation should be performed at a point in the infrastructure where there is enough computing power and battery power to keep the lifespan as long as possible. Furthermore, the communication for requested data should not flood the network, otherwise actual measurement-data can no longer be sent.

The aggregation of the physical unit of measurement data is important information to avoid calculation errors by falsely interpreted data. If the unit is predefined for the application, it can be added at every point or, since every component implicitly knows the unit of the measurement data, not be added. However, it is assumed that the unit is not predefined, since it cannot be guaranteed that every sensor node uses the same type of sensor. For example, the temperature sensor on **Node 1** measures it in °C and the sensor on **Node 2** measures temperature in °F. Therefore, other members of the network can not be sure about the unit every sensor sends. The only component that knows the unit without requesting it from a lower level, is the **Sensor node**. Every other component in the application infrastructure, except the **Client Software**, is able to determine the unit of the temperature by sending a request to the lower levels. This

raises the number of communication message exchanges needed, which further increases the power consumption of components that are involved. The **Client Software** is not able to request it because the data might have been evaluated already or has been merged by the **Application Software**. Due to the merging, aggregating of the physical unit at the **Sensor node** is mostly suitable option, if the unit is not predefined within the application, otherwise it would not be added at all.

When measuring environmental parameters, it is also important to know at which physical location the data is measured, for example, the temperature in a conference room is different from the temperature in a server room. The **Sensor node** knows its location either by explicit configuring it in the configuration file or it implicitly is given by the UUID of the node within the WSN. In both cases, the node can aggregate the data itself and then send the message, but this can increase the package size dramatically for long **Location** names. The **Location** of measurement can be added on the **Gateway**, if it has knowledge of the locations and the corresponding sensor names. This can either be achieved by using a predefined lookup table on the **Gateway** or the **Sensor node** transmits this information at the connection establishment process and the **Gateway** stores it. The **Application Broker** and **Software** can request the locations, but due to the high amount of incoming data the overhead for aggregating data becomes quite huge and might slow down the main application. Again, the **Client Software** is not able to request the exact **Location**, since the data has been processed or merged by the **Application Software**. Therefore, the most suitable point in the environment to add information about the **Location**, is the **Gateway**. Since the amount of incoming data is manageable, it has enough computation power and it does not need to request the data before aggregation.

6.2 Evaluation of the presented WSN models

In order to model the WSN into a graph environment with previous defines properties in chapter 5, four different models and a fifth one which is a combination of all four, is presented. This section outlines the benefits and drawbacks of each model and argues why the fifth model best fits the demand.

Model 1

The first model, presented in Figure 5.1, is a simple representation of the WSN and shows the direct impact of the communication partner, configurable system parameters as well as application relevant information.

However, the model does not group any nodes in sub-graphs, which means it is hard to determine, if the observed graph node is a system parameter, e.g. the **Gateway Port**, an application relevant setting, e.g. the **Topic ID** or a hardware component which generates or exchanges information, for example, a temperature **Sensor**.

Model 2

Figure 5.2 shows the second possible modelling approach for the WSN. A big advantage of this representation is the separation of hardware parts and software objects, which leads to a better overview and makes the model easier to read. Since every configuration parameter is a child node of **Configuration**, there is only one point of view to check if this controller in the WSN has the correct configuration.

However, the model does not distinguish between system parameters, for example, **Gateway Port**, and application information, for instance, the **Topic ID**, because they are all directly connected to the node named **Configuration**. Furthermore, the **UUID** is a child of **Configuration**, this strictly limits the reusability of the whole sub-graph, because it could not be used for more than one **Microcontroller** node since the **UUID** makes it unique. As has been mentioned above, this modelling approach allows to model a **Sensor** without a **Topic**.

Model 3

In the third model, which can be seen in Figure 5.3, the drawback of the second one is compensated, due to a relation between **UUID** the node **Microcontroller**. For this reason, the sub-graph with **Configuration** node as root can be used for several **Microcontrollers** with an identical configuration, which also reduces the number of nodes in the graph. Another advantage of this model is the separation of application data and system settings, which is achieved using a top level node **Topics**.

However, the link between **UUID** and the root **Microcontroller** node eliminates the strict separation between software parameters and hardware components, which makes the model a little bit harder to survey. The model allows to define a **Sensor** without a **Topic** it publishes to or vice versa.

Model 4

The last proposed model approach can be seen in Figure 5.4. This model bundles all system parameters under a top level node **Configuration**. Due to this property, the model has only one point where all configurable system parameters can be found. A big benefit of the fourth model is the link between **Sensor** and **Topic**, which ensures that there is no sensor without a topic that can it publish to and there is no unused topic where no data is published on.

Since the **Topic** is a child node of the **Sensor** nodes, the nodes **Topic ID** and **Unit** are generated for every temperature sensor, which makes them redundant. Furthermore, the **UUID** is a child node of **Configuration**. The **Configuration** sub-graph is a unique structure for every modelled **Microcontroller**, which also generates many redundant nodes and makes the model obscure.

Final Model

The model of the WSN should provide a good overview of the whole system with all its components and their interaction properties. Further, it should keep track of all application and system relevant parameters while trying to keep it as simple as possible.

After an analysis of the four possible models and their benefits as well as their drawbacks, a fifth one which fits the needs best is presented. The fifth model is a combination of Model 3 and Model 4, which can be seen in Figure 5.5. They seem to be a good compromise, because all the system settings are bundled under a node named **Configuration**, so there is one point to look up all setting of a WSN node. Another big advantage is the directly connected UUID to a microcontroller, a **Configuration** sub-graph. Therefore, it can be used for several microcontrollers with the same configuration, this reduces the number of nodes in the whole graph. Furthermore, the **Sensor** knows the topic it publishes to as a direct successor, so it is easy to identify under which certain topic a node and sensor publish its data.

6.3 Brick extension for Wireless Sensor Networks

The open-source semantic description effort Brick uses OWL as well as the RDF framework to describe the models. In Brick, it is possible to extend the existing ontology by user-defined classes and relationships. These extensions can enhance existing Brick classes/relationships or basic classes/relationships. In order to use the provided tools for visualization as well as to perform queries on Brick models, the syntax of Brick Ontologies should be respected. Although the Brick Ontology supports a huge number of classes and relationships, it does not fully support the modelling of a WSNs basic parts of electrical equipment, but a variety of sensors are already included. Also, entities and relations to model communication of between different components are missing in Brick. In order to create models as presented in the Section 5.2, an extension to Brick has been developed. This extension includes an abstract model of a **Controller Unit** and also concrete models of a **Sensor node**, a **Server** as well as entities and relations that are need to model communication processes, for example, a **Network Port** and an **IP-Address**. Furthermore, the opportunity to model the configuration of components within the WSN is added with this extension, for instance, it is possible to express how a **Sensor node** communicates on a specific channel or a **Server** listens on a predefined **Network Port** and has a static **IP-Address**.

6.4 Discussing result regarding research questions

The first research question aims towards finding a suitable point in the WSN to perform data aggregation.

RQ 1: Which location in the environment is most suitable for data aggregation in an ongoing process?

The aggregated data should be available at the point of aggregation without further communication overhead and the location should be near the source of the data, since the amount of incoming data gets higher the further the aggregation point is away from the source. In order to answer RQ1, a table in Chapter 5 shows the relation between available points for data aggregation in an application and the information added to the measurement-data. At the beginning of the chapter, the aggregation points are evaluated depending on the defined properties. This theoretical assumption **Sensor node** and **Gateway** being the most suitable points for data aggregation, has also been validated with the proof of concept. In the proof of concept, the data aggregation of the physical unit and the **Location** are performed on the **Gateway**, because the physical unit is a predefined parameter for the whole WSN, this information it necessary to be added at the **Sensor node**. Furthermore, the **Gateway** has enough computation power and unlimited lifespan.

The second research question aims towards answering how to use a graph environment to visualize the aggregated data to get a quick and compact overview of the whole WSN.

RQ 2: What are possible solutions for visualizing aggregated sensor data into a graph environment?

For answering RQ2, chapter 5 presents four different possibilities to visualize the WSN with gateways and an application server. The chapter displays each approach using a simple generic WSN application and highlights the advantages and drawbacks of each model. After considering the previously defined features and properties of the proposed models, another adaption to combine the advantages and fit the needs has been designed. The resulting modelling approach gives a compact overview of the whole WSN with its components and their interactions. Furthermore, the model also present necessary data for communication and parametrization of the components. Due to efficient grouping of different parameters into a subtree that can be connected to several nodes and the number of redundant parameters occurrences is kept minimal. This leads to a structured overview and reduces the number of edits if an parameter has to be changed. In Section 4.4, an implementation of the fifth version using the software Brick [33] is described.

Conclusion

The aim of this thesis was to build up a three tier WSN architecture, similar to the one presented in [3]. Furthermore, the WSN, especially the sensor nodes and their direct communication partner - sensors and Gateways - should be modelled and presented in a graph environment. The WSN is used as big data source [1] for the the upper layers, but due to limited lifespan and lightweight protocols, only a minimum amount of data is transmitted from the sensor node to sink. Due to this restriction, the most important data is sent only while other information, for example, meta-data is lost. Therefore, the three tier architecture is further split up into independent components, which were analyzed to find the most suitable point for aggregating meta-data, e.g. the location of measurement. The result of the analysis marks two major points for being most suitable to perform data aggregation, without interfering with the main purpose of the WSN, measuring and providing environment data to the application.

A model of the whole WSN was designed to give a compact overview of all involved components and to keep track of configuration parameters of every single component. In addition to that, it should also be possible to query the model to find settings of different components and to generate entire configuration files. These files are either header files for the firmware of the Sensor nodes which are written in the C programming language or configuration files which are passed to programs at the start-up. The model should fulfill certain requirements: provide a compact overview, avoid redundant modelling as good as possible of parameters and it should differentiate between physical components and meta-data. In order to achieve this, different model approaches are proposed and evaluated to satisfy required properties. The model approach that meets the proposed requirements best, has been realized and implemented as a proof of concept.

Future work could extend the data aggregation concept to add timestamps when the data is generated or combine data that comes from sensor nodes in similar locations. This enables the possibility of logging and also certain events can be reconstructed for debugging purposes. Combinations of data could increase the lifetime of a node

7. CONCLUSION

since not every measurement is sent, especially if it sends redundant information. Also, the generation process of the models can be accelerated, in order to speeding up the generation of the configuration if only one or two node configurations are changed. The toolchains of configuration writing and the compiler of the firmware could be combined in order to generate needed configuration at compile time only.

List of Figures

| | | |
|------|---|----|
| 2.1 | A 3-tier WSN architecture, based on and adapted from [3] | 4 |
| 2.2 | A basic SoC system model, from [35] | 5 |
| 2.3 | IPv6 Address scopes, based on and adapted from [16] | 9 |
| 2.4 | IPv6 Header format, based on and adapted from [14] | 9 |
| 2.5 | Example IPv6 Header and IPv6 Header chain, based on and adapted from [14] | 10 |
| 2.6 | MQTT connection, based on diagram 1 and 2 and adapted from [18] . . | 13 |
| 2.7 | Various MQTT-SN Architectures for PUBLISH to server, based on and adapted from [21] | 15 |
| 2.8 | Thread protocol stack, from [23] | 16 |
| 2.9 | Full- and Partially connected Mesh Networks | 17 |
| 2.10 | OpenThread Devices types, from [24] | 18 |
| 2.11 | OpenThread network structure, from [27] | 19 |
| 2.12 | OpenThread RLOC16 and Mesh-Local address examples, from [24] | 20 |
| 2.13 | Foundational Ontology | 22 |
| 4.1 | Abstract overview of the components and their interconnection | 27 |
| 4.2 | Brick model of a Sensor node | 35 |
| 4.3 | Flow of information from sensor to Application MQTT Broker | 38 |
| 4.4 | Output of Sensor node during initialization and operation | 39 |
| 4.5 | Sequence diagram of communication in the WSN | 40 |
| 4.6 | MQTT Broker on the Gateway; published temperature from the Sensor node | 40 |
| 4.7 | MQTT Broker on the Gateway; published temperature from the Python script | 41 |
| 4.8 | Application MQTT Broker; bridge temperature topic from MQTT Broker on the Gateway | 41 |
| 5.1 | Wireless Sensor Network Model 1 | 46 |
| 5.2 | Wireless Sensor Network Model 2 | 47 |
| 5.3 | Wireless Sensor Network Model 3 | 48 |
| 5.4 | Wireless Sensor Network Model 4 | 49 |
| 5.5 | Wireless Sensor Network Model 5 | 50 |

List of Tables

| | | |
|-----|--|----|
| 2.1 | Example IPv6 Address, from [14] | 7 |
| 2.2 | Shortened IPv6 Addresses | 7 |
| 2.3 | Common scope values for IPv6 Multicast addresses, from [15] | 8 |
| 2.4 | Meaning of IPv6 header fields | 10 |
| 2.5 | Most frequently used extension headers of IPv6 | 11 |
| 2.6 | Example RLOC Address, from [24] | 20 |
| 2.7 | Example Device IPv6 Address including RLOC16, from [24] | 20 |
| 2.8 | Reserved Multicast addresses in a Thread Network, from [24] | 21 |
| 2.9 | Predefined Anycast locator address, from [24] | 21 |
| 5.1 | Possible points in a WSN for data aggregation of Unit and Location | 44 |
| 5.2 | Transformation of message content and Topic from source to sink | 44 |

Listings

| | | |
|-----|---|----|
| 4.1 | Change pins of USART | 31 |
| 4.2 | Declarations for LED functionality | 31 |
| 4.3 | Content of file gpio.c | 32 |
| 4.4 | Command to compile OpenThread Library | 33 |
| 4.5 | Code to create node and configuration | 35 |
| 4.6 | Query UUID of Office1 Node | 36 |
| 4.7 | Configuration of Office1 Node | 36 |

Bibliography

- [1] H. Harb, A. Makhoul, A. Idrees, O. Zahwe, and M. Taam, "Wireless sensor networks: A big data source in internet of things," *International Journal of Sensors, Wireless Communications and Control*, vol. 07, 09 2017.
- [2] D. Tsitsipis, S.-M. Dima, A. Kritikakou, C. Panagiotou, and P. S. Koubias, "Data merge: A data aggregation technique for wireless sensor networks," pp. 1–4, 09 2011.
- [3] A. A. A. Ari, A. C. Djedouboum, A. M. Gueroui, O. Thiare, A. Mohamadou, and Z. Aliouat, "A three-tier architecture of large-scale wireless sensor networks for big data collection," *Applied Sciences (Switzerland)*, vol. 10, no. 15, 2020.
- [4] A. K. Idrees and A. K. M. Al-Qurabat, "Energy-efficient data transmission and aggregation protocol in periodic sensor networks based fog computing," *Journal of Network and Systems Management*, vol. 29, no. 1, 2021.
- [5] H. Harb, A. Makhoul, S. Tawbi, and R. Couturier, "Comparison of different data aggregation techniques in distributed sensor networks," *IEEE Access*, vol. 5, pp. 4250–4263, 2017.
- [6] B.-S. Kim, K.-I. Kim, B. Shah, F. Chow, and K. H. Kim, "Wireless sensor networks for big data systems," *Sensors (Basel, Switzerland)*, vol. 19, p. 1565, Apr 2019. 30939722[pmid].
- [7] S. Boubiche, D. E. Boubiche, A. Bilami, and H. Toral-Cruz, "Big data challenges and data aggregation strategies in wireless sensor networks," *IEEE Access*, vol. 6, pp. 20558–20571, 2018.
- [8] M. Nkomo, G. P. Hancke, A. M. Abu-Mahfouz, S. Sinha, and A. J. Onumanyi, "Overlay virtualized wireless sensor networks for application in industrial internet of things: A review," *Sensors (Basel, Switzerland)*, vol. 18, p. 3215, Sep 2018. 30249061[pmid].
- [9] S. Sharma, "Issues and challenges in wireless sensor networks," 12 2013.
- [10] *Internet of Things*. Pearson Education India, 2019.

- [11] IEEE Internet of Things, “IEEE Internet of Things Homepage.” <https://iot.ieee.org/>. [Online; accessed 15-August-2020].
- [12] I. E. T. Force, “Internet Protocol, Version 6 (IPv6) Specification–.” <https://tools.ietf.org/html/rfc2460>. [Online; accessed 17-May-2020].
- [13] I. E. T. Force, “Internet Protocol, Version 6 Addressing Architecture.” <https://tools.ietf.org/html/rfc2373>. [Online; accessed 17-May-2020].
- [14] H. P. Institut, “IPv6 in modernen Netzwerken.” <https://open.hpi.de/courses/ipv6-2018>. [Online; accessed 17-May-2020].
- [15] IANA, “IPv6 Multicast Address Space Registry.” <https://www.iana.org/assignments/ipv6-multicast-addresses/ipv6-multicast-addresses.xhtml>. [Online; accessed 22-May-2020].
- [16] E. Kompendium, “IPv6-Address-Scopes (Gültigkeitsbereiche).” <http://www.elektronik-kompendium.de/sites/net/2107111.htm>. [Online; accessed 20-May-2020].
- [17] OASIS, “MQTT - Official Homepage.” <http://mqtt.org>. [Online; accessed 15-March-2020].
- [18] R. K. Kodali and S. Soratkal, “MQTT based home automation system using ESP8266,” in *2016 IEEE Region 10 Humanitarian Technology Conference (R10-HTC)*, pp. 1–5, Dec 2016.
- [19] OASIS, “MQTT - Official Documentation.” <https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html>. [Online; accessed 6-April-2020].
- [20] OASIS, “MQTT-SN - Official Documentation.” http://mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN_spec_v1.2.pdf. [Online; accessed 6-April-2020].
- [21] K. Govindan and A. P. Azad, “End-to-end service assurance in iot mqtt-sn,” in *2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC)*, pp. 290–296, 2015.
- [22] H. Kim, S. Kumar, and D. E. Culler, “Thread/openthread: A compromise in low-power wireless multihop network architecture for the internet of things,” *IEEE Communications Magazine*, vol. 57, no. 7, pp. 55–61, 2019.
- [23] T. Group, “Thread - official homepage.” <https://www.threadgroup.org/>. [Online; accessed 28-April-2020].
- [24] Google, “Openthread - official homepage.” <https://openthread.io/guides/thread-primer>. [Online; accessed 28-April-2020].

- [25] W. Rzepecki and P. Ryba, “Iotsp: Thread mesh vs other widely used wireless protocols – comparison and use cases study,” in *2019 7th International Conference on Future Internet of Things and Cloud (FiCloud)*, pp. 291–295, 2019.
- [26] A. Sammes, S. C. Misra, S. Misra, and I. Woungang, *Guide to Wireless Mesh Networks*. Computer Communications and Networks, London: Springer London, 2009.
- [27] T. Group, “Thread technical overview.” https://www.threadgroup.org/Portals/0/documents/resources/Thread_Technical_Overview.pdf. [Online; accessed 9-May-2020].
- [28] J. Busse, B. Humm, C. Lubbert, F. Moelter, A. Reibold, M. Rewald, V. Schlüter, B. Seiler, E. Tegtmeier, and T. Zeh, “Was bedeutet eigentlich Ontologie?,” *Informatik-Spektrum*, vol. 37, 08 2014.
- [29] H. Stuckenschmidt, *Ontologien; Konzepte, Technologien und Anwendungen*. Informatik im Fokus, Berlin, Heidelberg: Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
- [30] P. Cimiano, A. Maedche, S. Staab, and J. Völker, *Ontology Learning*, pp. 245–267. 05 2009.
- [31] OWL Working Group, “Web Ontology Language (OWL).” <https://www.w3.org/OWL/>. Accessed: 2020-03-18.
- [32] W3C Recommendation 10 February 2004, “OWL Web Ontology Language Overview.” <https://www.w3.org/TR/owl-features/>. Accessed: 2020-03-18.
- [33] B. Team, “Brickschema website.” <https://brickschema.org/>. Accessed: 2020-09-08.
- [34] B. Balaji, A. Bhattacharya, G. Fierro, J. Gao, J. Gluck, D. Hong, A. Johansen, J. Koh, J. Ploennigs, Y. Agarwal, M. Berges, D. Culler, R. Gupta, M. Kjærgaard, M. Srivastava, and K. Whitehouse, “Brick: Towards a unified metadata schema for buildings,” in *Proceedings of the 3rd ACM International Conference on systems for energy-efficient built environments*, BuildSys ’16, pp. 41–50, ACM, 2016.
- [35] M. J. Flynn and W. Luk, *Computer system design : system-on-chip*. Hoboken, NJ: Wiley, 2011.
- [36] B. Oates, “Researching information systems and computing,” 2005.
- [37] S. T. March and G. F. Smith, “Design and natural science research on information technology,” *Decision Support Systems*, vol. 15, no. 4, pp. 251 – 266, 1995.
- [38] P. Checkland, “Soft systems methodology: A thirty year retrospective,” *Systems Research and Behavioral Science*, vol. 17, p. S11–S58, 11 2000.

- [39] A. R. Hevner, S. T. March, J. Park, and S. Ram, “Design science in information systems research,” *MIS Q.*, vol. 28, p. 75–105, Mar. 2004.
- [40] J. Hughes and T. Wood-Harper, “Systems development as a research act,” *Journal of Information Technology*, vol. 14, pp. 83–94, Mar 1999.
- [41] W. Kastner, S. Gaida, H. Tellioglu, “Human erception and building automation systems.” <http://humbas.org/>. ICT of the Future, Project Number 867681.