MASTER'S THESIS

Design and implementation of an automated gateway prototype for the Internet of Things

HSB
Hochschule Bremen
City University of Applied Sciences

Institution
Hochschule Bremen
Flughafenallee 10
28199 Bremen

Author
Marvin Soldin, 5010034
msoldin@stud.hs-bremen.de

First examiner
Prof. Dr.-Ing Jasminka Matevska

Second examiner
Prof. Dr. Lars Braubach

Study program
Informatik M. Sc.

Semester
Wintersemester 2021/2022

May 30, 2022

Declaration of copyright

Hereby I declare that my Master's Thesis was written without external support and that I did not use any other sources and auxiliary means than those quoted. All statements which are literally or analogously taken from other publications have been identified as quotations.

This declaration also applies to graphics, sketches and illustrations contained in the thesis as well as to sources from the Internet.

I have not yet submitted the work in the same or similar form, even in excerpts, as part of an examination or course assignment.

I certify that the submitted electronic version of the thesis is completely identical to the printed version.

First and last name	Matriculation number
Date, Place	Signature

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit entsprechend gekennzeichneten Teile der Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfs-mittel benutzt habe. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, sind durch Angaben der Herkunft kenntlich gemacht.

Diese Erklärung erstreckt sich auch auf in der Arbeit enthaltene Grafiken, Skizzen, bildliche Darstellungen sowie auf Quellen aus dem Internet.

Die Arbeit habe ich in gleicher oder ähnlicher Form auch auszugsweise noch nicht als Bestandteil einer Prüfungs- oder Studienleistung vorgelegt.

Ich versichere, dass die eingereichte elektronische Version der Arbeit vollständig mit der Druckversion übereinstimmt.

Vor- und Nachname	Matrikelnummer
Datum, Ort	Unterschrift

Abstract

The Internet of Things is becoming increasingly important as the number of devices grows. Nowadays, almost everything is interconnected, so the amount of data produced is constantly increasing. Thus, a gateway is a decisive part of the Internet of Things. As per IoT architecture, a gateway is a device that acts as a connection point between IoT devices and their applications. It is an essential aspect of an IoT System because most IoT devices cannot connect to the cloud directly. While a great deal of research has already been done on IoT in cloud computing, fog computing, and edge computing, little seems to have happened in the field of gateways in particular. Nevertheless, gateways have to deal with many different problems, such as the diversity of protocols and the custom needs of each environment. While basic gateways only act as a proxy between low-end IoT devices and data centres, automated gateways have significantly more functions to solve those problems. For this reason, the following work examines the current state of the art and, based on this, presents a concept of an automated gateway. The concept deals more specifically with the problems of protocol conversion, device management, middleware abstraction, resource management and traffic optimisation. In order to evaluate the concept, it was also implemented and tested based on a concrete scenario. The gateway uses a modularised plug-and-play architecture to solve the mentioned problems. Different protocols can be added via their software modules and modified in user-defined ways. Currently, there are additional software modules for MQTT, ZigBee, WebSocket and Amazon WebServices that can be added via this architecture. However, the gateway has externally defined interfaces that can be used to develop additional software modules for other protocols and services. Finally, the gateway defines its protocol and translates incoming messages into a uniform format. The client can also use the format directly to enable more complex message flows.

Contents

List of Figures

List of Tables

List of Listings

Glossary

DSSS Direct Sequence Spread Spectrum is a modulation method applied to digital signals. It increases the signal bandwidth to a value much larger than needed to transmit the underlying information. In DSSS, spreading codes that are independent of the original signal are used to achieve the goal of bandwidth expansion. Both a sender and a receiver agree on a spreading code, which is regarded as a shared secret between them [Liu+10, p. 2].

Gzip GNU Gzip is an open-source program used for file compression and decompression. Jean-loup Gailly and Mark Adler created the program as a free software replacement for the compress programs used in early Unix systems. Gzip is based on the DEFLATE algorithm, a combination of LZ77 and Huffman coding. [Wik22a].

HTB The Hierarchical Token Bucket (HTB) algorithm is one of the most widely used algorithms for rate limiting. HTB allows to classify various types of traffic into different queues. The different queues can be configured with a priority level and a rate limit, allowing to schedule the packets and shape the traffic accordingly. HTB relies on two token buckets for controlling the bandwidth usage of a link [Bos+21, p. 1].

I2C I2C is a synchronous, multi-controller, packet switched, serial communication bus intervented in 1982 by Philips Semiconductors. It is widely used for attaching lower speed peripheral to processors and microcontrollers. [Wik22b].

JAAS The Java Authentication and Authorization Service (JAAS) is the Java implementation of the standard Pluggable Authentication Module (PAM) information security framework. [Cor22b].

JMX The JMX API is a standard API for management and monitoring of resources such as applications, devices, services, and the Java virtual machine..

Modbus Modbus is a data communication protocol originally published by Modicon in 1979 for use with its programmable logic controllers. Modbus has become the de facto standard communication protocol in industrial environments and is now commonly available. [Wik22c].

MTBF MTBF is defined as the mean time between failures. It is the predicted elapsed time between inherent failures of a mechanical or electronic system, during normal system operation.[Wik22f].

MTTF MTTF is defined as the mean time to failure and it describes the predicted elapsed runtime of the system to the next failure. [Wik22f].

MTTR MTTR is defined as the mean time to recovery after a system failure. This indicates how long it takes on average to restore a system. It is, therefore, an important parameter for system availability. [Wik22f].

PROFINET PROFINET is an open Industrial Ethernet Standard devised by PROFIBUS International (PI) for either modular machine and plant engineering or distributed IO. Profinet uses TCP/IP and IT standards, is real-time capable and enables the integration of fieldbus systems. [Fel04].

Watchdog A watchdog timer, or simply a watchdog, is an electronic or software timer that is used to detect and recover from computer malfunctions. [Wik22g].

WQF Weighted fair queueing (WFQ) is a network scheduling algorithm. The algorithm provides fair output bandwidth sharing according to assigned weights. Weighted fair queue is a variant of fair queue equipped with a weighted bandwidth allocation. [MT+16].

Acronyms

AMQP Advanced Message Queuing Protocol.

APO Application Object.

APS Application Support Sublayer.

AWS Amazon Web Services.

BE Best Effort.

CoAP Constrained Application Protocol.

CPU Central processing unit.

D2D device-to-device.

DDoS Distributed denial-of-service attack.

DoS Denial-of-service attack.

FFD Full Function Devices.

FIFO First in, first out.

GPIO General-purpose input/output.

GPS Global Positioning System.

HP High Priority.

HTTP Hypertext Transfer Protocol.

IDE Integrated development environment.

IoT Internet of Things.

JSON JavaScript Object Notation.

LoWPAN Low-Power Wireless Personal Area Networks.

M2M Machine to Machine.

MAC Media access control.

MQTT Message Queuing Telemetry Transport.

OGC Open Geospatial Consortium.

OPC-UA OPC-Unified Architecture.

openHAB open Home Automation Bus.

OSGi Open Services Gateway initiative.

OSI model Open Systems Interconnection model.

POM Project Object Model.

QoS Quality of Service.

RBAC Role-Based Access Control.

RFD Reduced Function Devices.

RFID Radio-frequency identification.

SGS Semantic Gateway as Service.

SLA Service-level agreement.

SQL Structured Query Language.

SSH Secure Shell Protocol.

SSL Secure Sockets Layer.

TCP Transmission Control Protocol.

UDP User Datagram Protocol.

UML Unified Modeling Language.

URI Uniform Resource Identifier.

USB Universal Serial Bus.

WSN Wireless Sensor Networks.

XMPP Extensible Messaging and Presence Protocol.

ZDO Zigbee Device Object.

Chapter 1

Introduction

In recent years, many smart devices have been connected to the Internet to record and process data for various purposes. For example, the applications smart health, smart city and smart home are most common. However, also various private applications fall into this field. In general, this concept can be described as the Internet of Things (IoT). [FA18, p. 1]

With the growing number of devices, this environment is becoming more and more important. Nowadays, almost everything is networked, so the amount of data being produced is constantly increasing. In 2025, it is predicted that the total number of IoT devices will be approximately 75.44 billion, as shown in figure 1.1. [Ala18, p. 1]

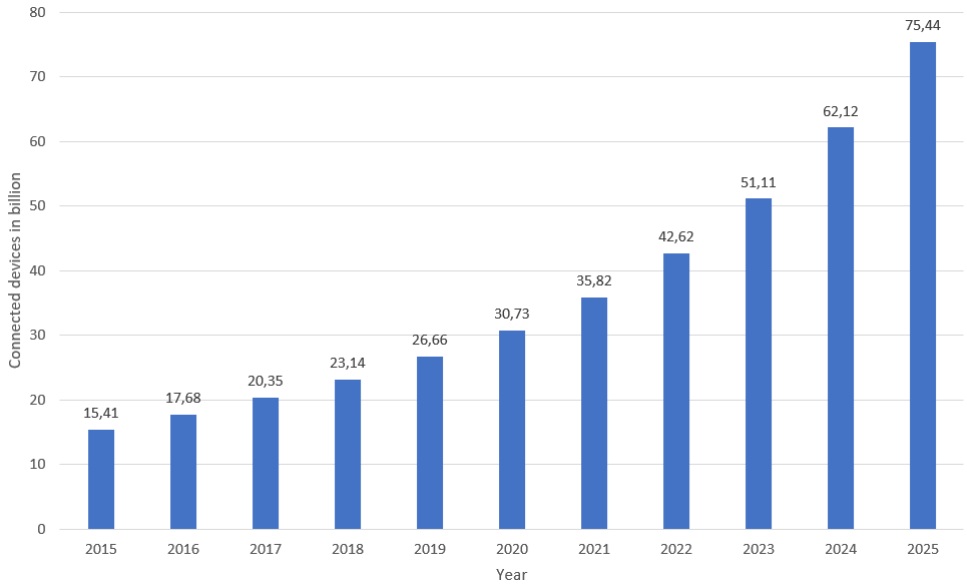


Figure 1.1: Forecast of connected devices from 2015 to 2025 (in billions). [Ala18, p. 1]

Because of this importance, the Internet of Things has gone through many evolutionary steps. Cloud computing is used to handle the amount of data. It provides a suitable utility model that offers on-request access to the application users. However, the IoT devices exponentially scaled up, leading to a tremendous amount of data, creating specific issues in latency and bandwidth for real-time applications. [BS21, p. 2]

Cisco then introduced Fog computing in 2012 to address the massive issues in IoT applications. Fog computing was not a substitute for cloud computing but an expansion. It acts as a distributed computing model that extends cloud services to the network's edge, thus allowing billions of devices to be served. [BS21, p. 2]

The devices, including sensors and actuators, rely on a gateway to aggregate the data transmitted by heterogeneous devices using various communication protocols. The gateway then forwards this data to the already mentioned structures. Thus, a gateway is a decisive part of the Internet of Things. As per IoT architecture, a gateway is a device that acts as a connection point between IoT devices and their applications. It is an essential aspect of an IoT system because most IoT devices cannot connect to the cloud directly. [BS21, p. 2]

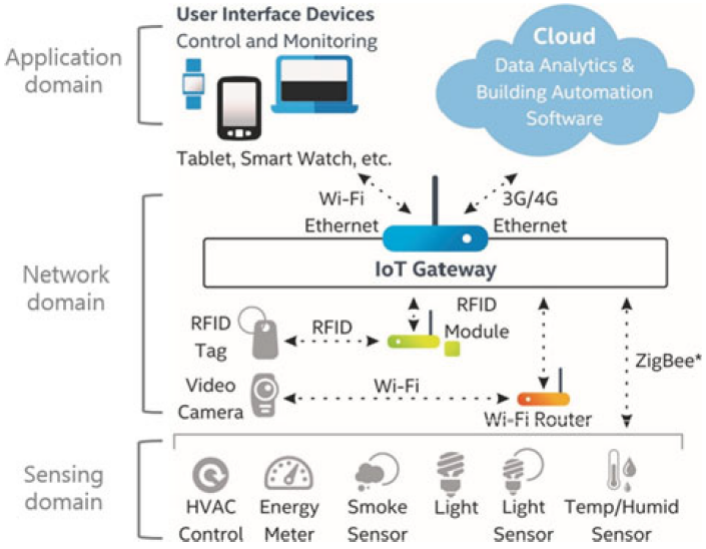


Figure 1.2: General features of an IoT gateway. [KKC17, p. 1]

1.1 Problem statement

While a great deal of research has already been done on IoT in cloud computing, fog computing, and edge computing, little seems to have happened in the field of gateways in particular. [BS21, p. 2]

Therefore, there are still many problems to be solved in this field. One of the obstacles to IoT development is the lack of standardization of communications, especially considering the heterogeneity of protocols for communication with the devices and the high variability of these devices, which leads to the so-called interoperability crisis in IoT. It is common for sensor manufacturers to use close standard protocols or different messaging protocols that prevent or hinder their use by third-party applications. This issue becomes even more severe for IoT devices that operate only at the second layer of the Open Systems Interconnection model (OSI model), as these devices depend on a gateway to send data used by IoT applications. Examples of this would be a device which is connected by the Universal Serial Bus (USB) [Fil+17, p. 1]

Furthermore, the protocols heterogeneity causes problems and the device management itself. Each device must be configured in order to connect to the cloud. This involves configuration on the device side and the cloud side. Unfortunately, current gateways operate either passively or semi-automatically. This means that the user has to add new devices manually. [KKC17, p. 1]

Unfortunately, these are not the only challenges a gateway has to deal with. Another problem that emerges from this challenge is that IoT gateways need to forward a large amount of data in a short time, which requires IoT gateways to have the ability to handle concurrent mass data. If more devices are added, the gateway may be overloaded and no longer be able to process the amount of data. [Min+14, p. 1]

However, these problems are only small sections of the overall subject. For this reason, more work needs to be done in the field of gateways.

1.2 Research Aim, Objectives and Questions

Based on the mentioned problems, the thesis concentrates on researching and designing a concept for an automatic gateway. Since not all gateways problems can be solved here, a unique scenario was designed for this thesis.

In this concrete example scenario, the goal is to provide a customer with a scalable solution for their business. The fleet of IoT devices in use is expected to increase in the future, both in terms of the number of protocols and devices. It is also essential to the end-user that the gateway is scalable so that there is little to no downtime. Finally, the end-user must not have to constantly access the gateway to connect new devices to the cloud. This means that there should be as little human interaction as possible.

Based on this scenario, the following sub-problems were set as mandatory objectives. Furthermore, these sub-problems are the first steps towards an automated gateway. The reason for this will be explained later.

- *Protocol conversion*: The gateway should contain an approach to solve the so-called interoperability crisis in the IoT area. Therefore, the gateway should be able to support several protocols. The incoming messages should then be transformed into another protocol explicitly selected for the cloud communication.
- *Device management*: Device management should be as automatic as possible, i.e. devices that connect to the gateway should be automatically registered and integrated into the communication with the cloud.
- *Middleware abstraction*: As already described, the gateway is to mediate between intelligent devices and IoT middleware/cloud applications. It is possible to connect to different cloud providers available on the market with such a function.
- *Ressource management*: The gateway should work as reliably as possible and react automatically to overloads. This means that the available resources are monitored and automatically distributed. Therefore, the gateway should be able to recognise high-priority applications. The goal of operability is crucial here because the gateway should always be accessible for remote maintenance.
- *Traffic optimization*: There are interesting possibilities to gain performance using streaming optimisation techniques, such as segmenting data into queues, performing packet prioritisation on specific queues, and queuing compression and deduplication for transmission.

Furthermore, since the gateway cannot solve heterogeneity directly for all protocols, it focuses on three selected example protocols. In this case, they are MQTT, ZigBee and WebSocket. Nevertheless, the design of the protocol connections is kept as modular as possible so that other protocols can be integrated without problems. The reasons for selecting the protocols can be found in the following chapter 3 on page 22.

A cloud provider is also the end consumer to represent the real world as closely as possible. Nevertheless, the choice of the end consumer was not a relevant question since the design implements a middleware abstraction already. For this reason, Amazon Web Services (AWS) was used since another project at Bremen University of Applied Sciences also uses it.

The goal was to implement the described gateway for this specific scenario. This implies that sensor data is sent to the AWS IoT Core service via the defined gateway using the protocols mentioned. All functionalities for the gateway should be implemented and tested. In this case, the sensor data itself is not essential and can be considered negligible.

1.3 Outline

The thesis is divided into ten parts, the first of which is the introduction. In the next section, a brief overview of the topic of gateways is given. Here, open problems, relevant publications to date and the current state of the art are also summarised. The following section gives a detailed explanation of the protocols used. It also explains how and why these protocols were chosen for the work. This is followed by the exact specification of the prototype for the elaboration. This means that the exact requirements for the system are described. In addition, the software and hardware environment used are also discussed. This is followed by a chapter on the design of the prototype. There, all the essential structures and processes are described in more detail. This is followed by the implementation chapter, which explains missing processes and highlights some unique code features in more detail. It also explains how to reproduce the test environment exactly. The requirements mentioned are then verified in the following chapter. More precisely, the implementation is tested against the functional requirements. In addition to testing, the application is evaluated in more detail in the next chapter. This means that it is tested for performance and other parameters found in the specification. Based on this, the results are documented and evaluated. The penultimate chapter summarises all the results achieved in this elaboration. Finally, the last chapter gives an outlook on the subject matter.

Chapter 2

IoT Gateway

In the Internet of Things, a gateway acts as an intermediary between several different sensors and cloud platforms. These gateways aim to solve the heterogeneity created by different devices and protocols at the sensor level and to forward the resulting data to the cloud. [BS21, p. 2]

According to Gunjan Beniwal and Anita Singhrova, gateways can be fundamentally divided into the basic gateway and the smart gateway. While basic gateways only act as a proxy between low-end IoT devices and data centres, smart gateways have significantly more functions. In contrast, a smart gateway handles data efficiently by preprocessing, filtering, analyzing, and delivering only the related or necessary data to the data centre. Furthermore, these intermediate devices are designed to handle harsh environmental conditions to recover from failure while solving the communication gap in minimal time. In addition, a smart gateway can be divided into three subtypes based on its functionalities, as shown in the following figure. [BS21, p. 3]

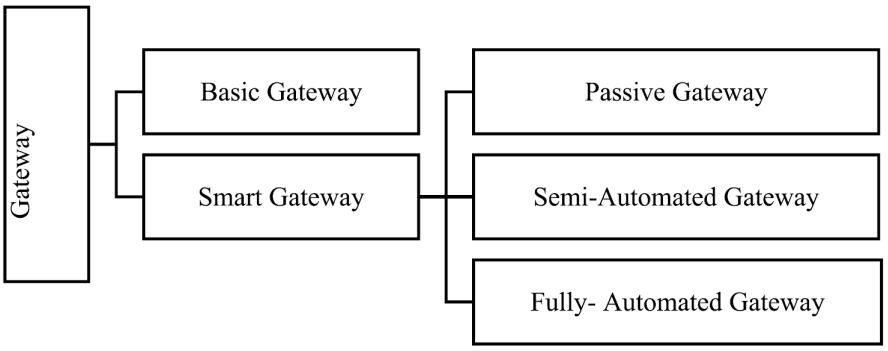


Figure 2.1: Types of gateways in the Internet of Things. [BS21, p. 4]

Sometimes gateways are designed to act smart by discovering IoT devices, registering them to the network, or removing them for better performance. When these functionalities are performed manually, it is a passive gateway. The user is adding or removing devices by himself while using a specific setup manual. However, these gateways are not customizable and flexible. [BS21, p. 3]

On the other hand, a semi-automated gateway manages a link between newly added devices and automatically creates connections to new devices. Furthermore, these gateways support a pluggable configuration architecture, which means they can be plugged in based on the requirement of the network device. As a result, these gateways are more flexible than passive gateways and perform better in real-time applications. [BS21, p. 3]

In contrast, there is the automated gateway, as it is self-configurable and self-manageable. There is no human intervention, and the devices can be added and removed automatically. Furthermore, these gateways can operate efficiently in a heterogeneous network and communicate over many different protocols such as WI-FI, Message Queuing Telemetry Transport (MQTT), Constrained Application

Protocol (CoAP), Bluetooth, ZigBee or Ethernet and many more. Work has been done in this specific field, but it is still in the growing phase, and researchers focus on making gateways smart enough to provide better performance and Quality of Service (QoS). [BS21, p. 2 f.]

Based on the available literature, gateways have mainly been categorized depending on their functional, non-functional, and architectural requirements. The complete overview is presented in figure 2.2. The specific functional requirements are further divided into data management, resource management, QoS management, device management, security management, and protocol translation, as shown in figure 2.3 on the next page. Non-functional requirements mainly include features to deal with difficult situations, such as the high number of units. Thus, requirements such as scalability, heterogeneity, reliability, low latency, energy efficiency, and mobility must be implemented. Finally, architectural requirements are mainly about end-user requirements. This means, for example, whether the gateway acts automatically or whether it is adaptable for multiple environments. [BS21, p. 5]

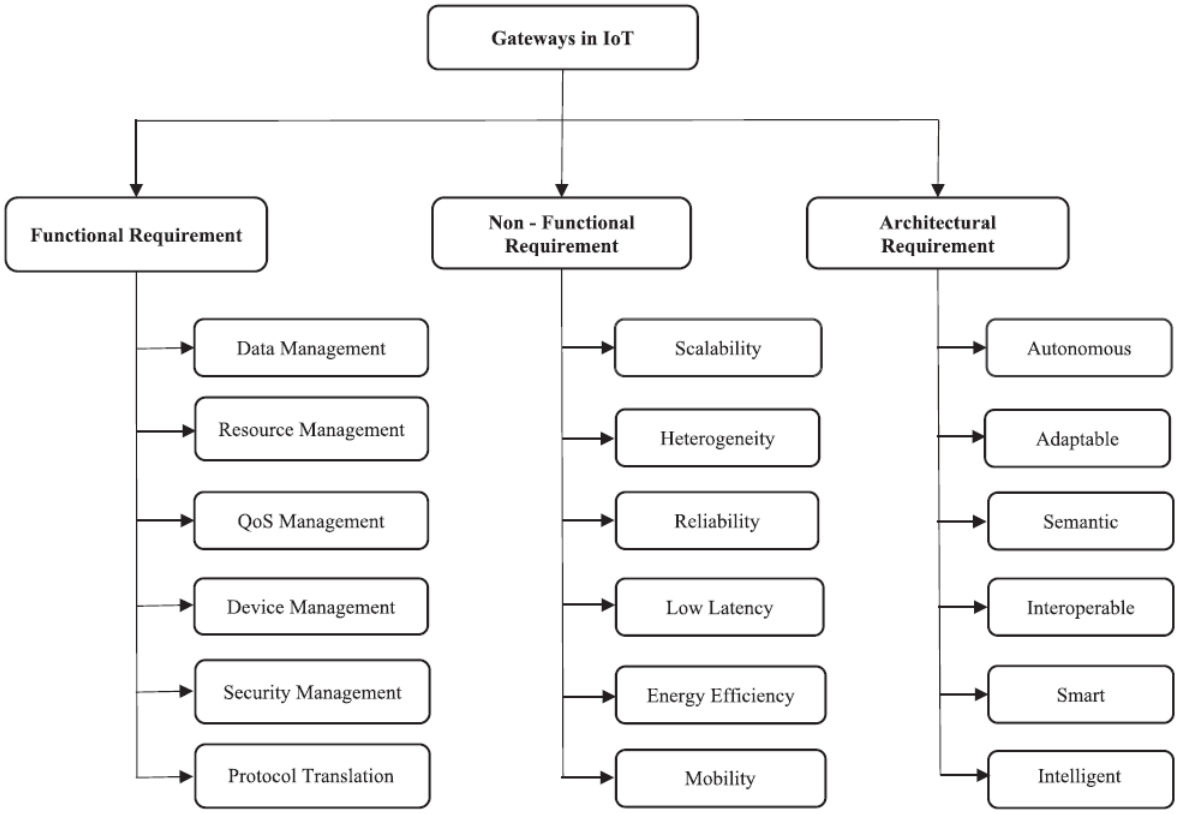


Figure 2.2: Requirements of gatways in the Internet of Things. [BS21, p. 7]

The functional requirements can also be further subdivided. Data management is further subdivided into data aggregation, storage, analysis, and offloading. It deals with the management of collected data from different sensors and devices. Databases can be used to store pre-processed data. Gateways are also responsible for offloading data after the complexity of the data has been analysed. Data can be offloaded completely, partially or not, depending on the computing capacity required. Resource management is divided into resource discovery, resource provisioning and resource scheduling. The primary responsibility of the gateway is to allocate the required resources to the IoT nodes for processing. Resources are discovered and allocated according to predefined criteria or algorithms. Resource scheduling involves scheduling tasks and assigning priorities to handle latency-driven or essential tasks. QoS management is categorised into various parameters such as throughput, latency, jitter, power consumption and response time. This includes data management, scheduling and resource allocation to improve the QoS of the overall system. Device management can also be classified into device discovery, configuration, and registration. For any communication, devices must be discovered to configure or register them. Smart gateways handle passive, semi-automated or fully automated provisioning of devices. Security

management is further categorised into authentication, data security and privacy, context awareness, and trust modelling from security aspects. Gateways are the entry point into the network and thus the most vulnerable points for threats or attacks. Many already known algorithms or machine learning can be used to improve security. Finally, protocol translation can also be further divided into communication and message protocols. WI-FI, Bluetooth, Ethernet, ZigBee, and Low-Power Wireless Personal Area Networks (LoWPAN) are some of the communication protocols used. Protocols including MQTT, CoAP, Extensible Messaging and Presence Protocol (XMPP), Advanced Message Queuing Protocol (AMQP), and Hypertext Transfer Protocol (HTTP) are well-known protocols adopted for message transmission in IoT Systems. [BS21, p. 10-13]

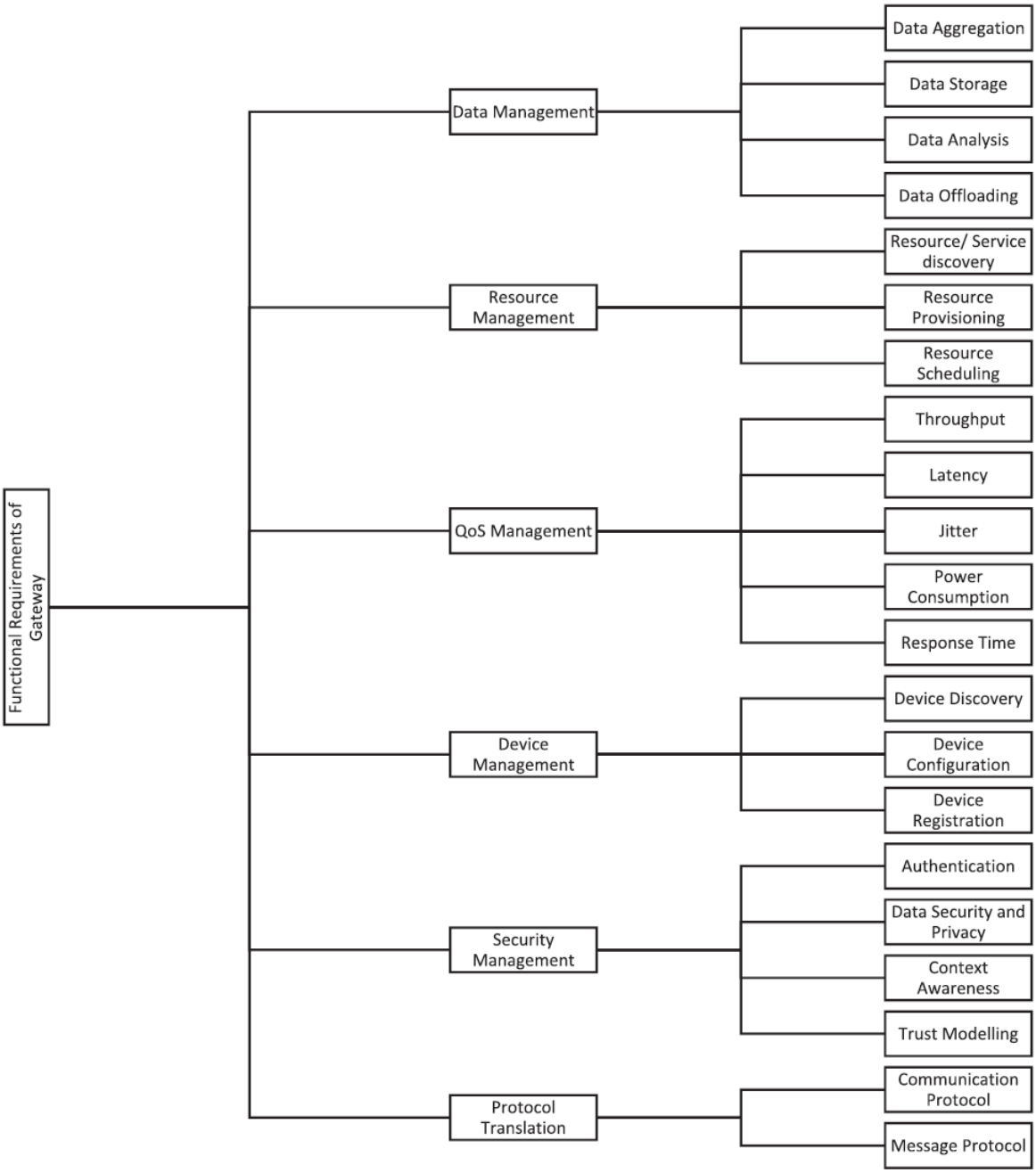


Figure 2.3: Functional requirements of gatways in the Internet of Things. [BS21, p. 11]

2.1 Related Work

In order to provide an accurate picture of the scientific work published to date, the following chapter will summarize previous and related work. First, the chapter will provide an overview of the topic, and then more specific articles will be discussed to shed more light on particular aspects used in this thesis.

2.1.1 A systematic literature review on IoT gateways

The paper "A systematic literature review on IoT gateways" has conducted a systematic literature review on this topic. A total of 2347 articles from the last ten years, i.e., from 2011 till July 2021 were considered. From these articles, 67 were then selected for a more detailed analysis based on defined criteria. [BS21, p. 1]

Based on this article, some interesting questions can be answered to see what work has already been done in this environment.

What type of gateway has attracted more attention in the field of IoT?

Currently, passive gateways have a 75% contribution in research compared to 18% semi-automated and only 7% fully auto gateways. In the past, the focus has been only on specific functionalities, leading to the popularity and adoption of the passive gateway, which needs to be handled manually for configuring and registering any new device to the network. However, as the technologies have become much more complicated, the focus is more on fully automated gateways. These gateways could eliminate any human intervention to provide a better user experience. [BS21, p. 10]

Which evaluation parameters are considered for QoS provisioning in IoT gateways?

The QoS parameters considered in the study are response time, latency, energy efficiency, network bandwidth, security, reliability, cost, execution time, and scalability. Most of the previously reviewed literature focused on improving the system's energy efficiency. Others mainly concentrated on improving the architecture of the gateway to perform better with low latency, high reliability, and scalability. A detailed analysis is shown in figure 2.4. [BS21, p. 12 f.]

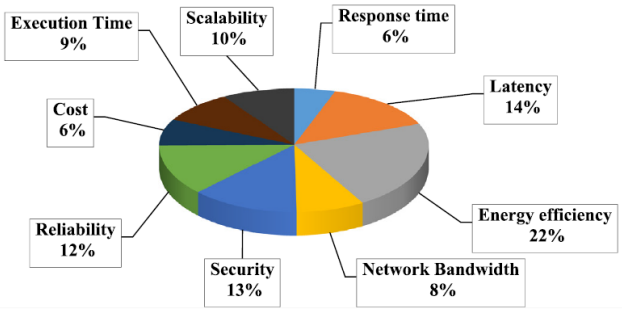


Figure 2.4: Parameters for QoS at IoT gateways. [BS21, p. 13]

What tools/platforms are used for the evaluation of IoT gateways?

As already shown in figure 2.5 the Raspberry Pi and Arduino are the most used testbeds, along with other microcontrollers and platforms. [BS21, p. 9]

> Around 20% of work was implemented and evaluated using Raspberry Pi. Approximately 19% of work was implemented and evaluated using combinations of different microcontrollers like XBee daughterboard, Pandaboard, Zigduino, Smart RF06 board, DPWSim, Octopus X, MCU Node, Docker Containers, Weka tool, Jena, Orange Pi and Beagle boards. Around 17% using Arduino board, 17% other platforms including personalized platforms like IoTivity and case studies, 7% using smartphones, 5% using MATLAB simulator, 5% using self-made UT-GATE, 3% OMNET++ simulator, 3% Cisco Packet Tracer, 2% Cooja simulator and 2% AllJoyn platform. [BS21, p. 9 f.]

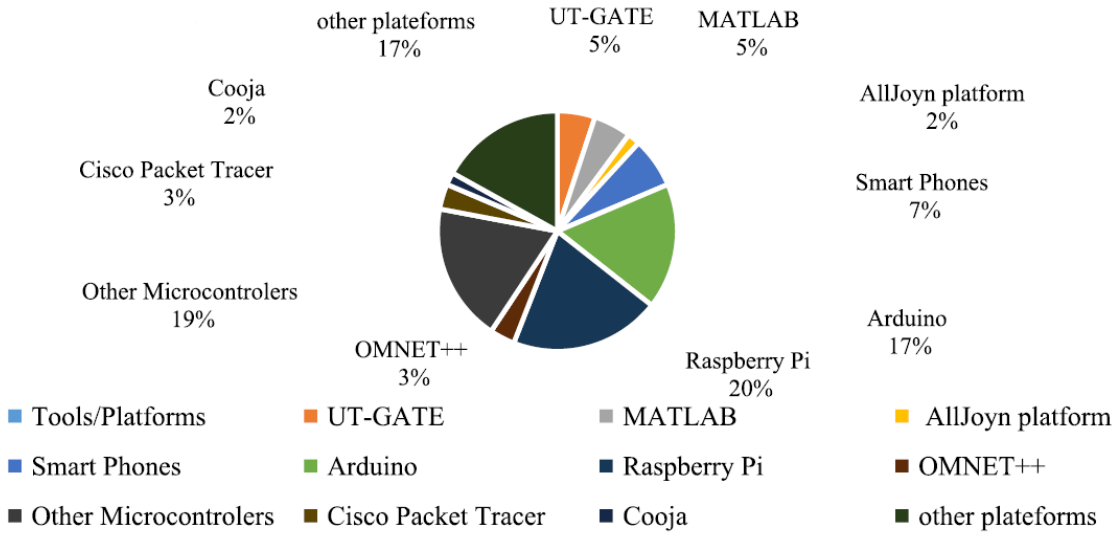


Figure 2.5: Types of tools and platforms adopted for implementation and evaluation of proposed work. [BS21, p. 10]

What are existing challenges and open issues of research in IoT gateways?

While there was already a significant development in the internet of things, there is still much bigger room for research and improvement for fully automated gateways. Some of the existing issues and research challenges are summarized in the following. [BS21, p. 13]

- *Heterogeneity*: IoT applications deal with many heterogeneous smart devices to gather data in different formats and sizes. The gateway is a bridge between all the connecting devices and should handle heterogeneous data among different protocols. While several models and architectures have been proposed to solve this problem, there is still no standardized solution.
- *Scalability*: Gateways should handle the increasing amount of IoT devices without deteriorating the services applied to them. With growing technology, smart devices are also exponentially increasing, leading to issues regarding scalability. For this reason, it is a big problem to distribute the computing resources smartly so that failures do not occur.
- *QoS*: While dealing with a large amount of data and services, the gateway should also provide a quality of service while dealing with real-time data. Many parameters were already considered, like reliability, latency, energy efficiency, scalability individually, but parameters like throughput,

roundtrip delay and jitter were overlooked. Simultaneous parameters should also be considered for improving QoS at gateways in IoT.

- *Security and Privacy*: Since gateways are one of the significant data entry points to the network, it is more vulnerable to threats and malicious activities. There was already some work focused on network security, but algorithms and architectures can also opt for more secure gateways and IoT architecture.
- *Intelligence*: Gateways should be smart enough to deal with intelligent data offloading. It is still a problem how much data should be offloaded, in which case, for how much processing.
- *Robustness*: Based on the previous work, no concepts of recovery of the gateway, fault tolerance, and self-healing were found. Therefore, no work has been invested in these topics yet.

What are the prospective future directions for research in IoT gateway?

Based on the existing issues and research gaps, future directions are summarized in this section. [BS21, p. 13 f.]

- While much work has already been done to deal with heterogeneity, scalability and interoperability problems individually, there is still a need for a standardized solution to solve these problems simultaneously.
- QoS parameters can also be simultaneously measured in the future to provide a better performance of IoT systems.
- Machine learning can be used within traditional security algorithms to deal with privacy and security concerns.
- Fully automated gateways need to be worked on in the future. Gateways need to make spontaneous decisions based on user behavioural patterns and complex situations. Working more with intelligent and smart gateway can be an exciting challenge.
- In future, there is a need for self-healing and fault-tolerant gateways to deal with the colossal amount of data. In dealing with Big Data, gateways should be reliable, fault-tolerant, self-manageable and self-healed.

2.1.2 Qos Scheduling Algorithm for a Fog IoT Gateway

Packet scheduling is an essential function in most network systems. The following paper "QoS Scheduling Algorithm for a Fog IoT Gateway" investigated this topic further.

In this work, they proposed an approach to include a classifier, which analyzes the messages received by the gateway and separates them into queues. The messages were divided into High Priority (HP) and Best Effort (BE). A modified version of the Hierarchical Token Bucket (HTB) algorithm was used, which aimed to achieve fair treatment of the requests. The experimental evaluation showed that high priority messages achieved a 24% reduction in latency without discards or losses of packages in this queue. [Cal+19, p. 2]

For the evaluation, three scenarios were constructed, either with or without active scheduling. Half of the devices used the Transmission Control Protocol (TCP) for transmission, and half User Datagram Protocol (UDP). [Cal+19, p. 5-6]

1 A single burst with 20 simulated sensor devices. The devices were kept in operation for 10 minutes, having sent 400 requests to the gateway.

2 Three bursts with 20 simulated sensor devices. The first at the beginning of the experiment, the second at the second minute of the experiment and the third at fourth minute of the experiment. All devices were kept in operation for 10 minutes, and a total of 1200 requests were sent to the gateway.

3 Three bursts with 40 simulated devices. The first at the beginning of the experiment, the second at the second minute of the experiment and the third at the fourth minute. All devices were kept in operation for 10 minutes, and 2,400 requests were sent to the gateway.

	TCP		UDP	
Scenario				
Scenario 1 without QoS	18.1	54.0	35.3	61.3
Scenario 1 with QoS	19.6	50.3	41.0	73.9
Scenario 2 without QoS	141.6	171.2	372.2	287.8
Scenario 2 with QoS	80.1	74.2	344.8	173.3
Scenario 3 without QoS	255.8	344.2	736.4	592.3
Scenario 3 with QoS	192.8	228.8	862.8	670.8
Overall total	168.5	251.5	576.8	568.1

Table 2.1: Mean and standard deviation of delay (all times are in ms) [Cal+19, p. 6]

2.1.3 Semantic Gateway as a Service architecture for IoT Interoperability

Interoperability is one of the significant challenges in achieving the vision of IoT. The following paper "Semantic Gateway as a Service architecture for IoT Interoperability" offers a solution approach to solve this problem.

This paper proposes a gateway to provide interoperability between systems, which utilizes established communication and data standards. The Semantic Gateway as Service (SGS) allows translation between messaging protocols such as XMPP, CoAP and MQTT via a multi-protocol proxy architecture. [DSA15, p. 313]

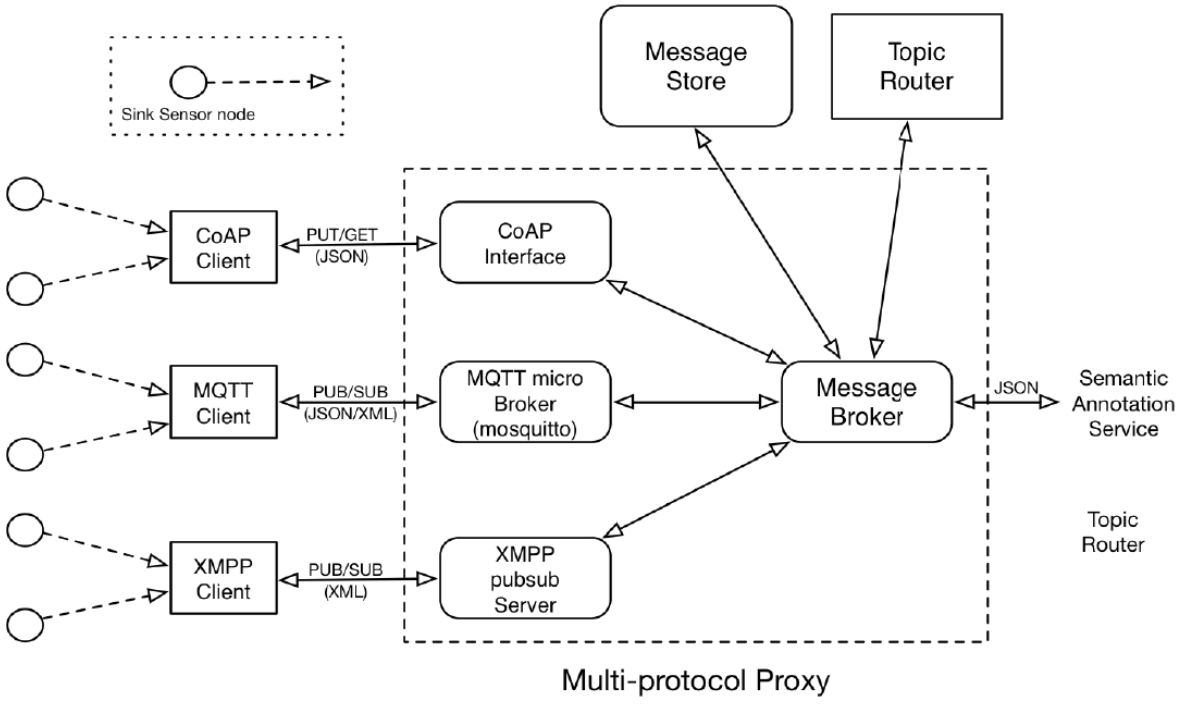


Figure 2.6: Multi-protocol proxy, communicating with sensor nodes. [DSA15, p. 316]

Figure 2.6 shows the message translation between various protocols. When a sensor generates a message or changes its state, the corresponding client sends that change to the SGS, which gets captured by the multi-protocol proxy. The proxy aligns that resource with the appropriate topic from the topic router and fetches the subscriber list. After passing through the semantic annotation block, the proxy forwards that message to these subscribers. [DSA15, p. 316]

The semantic annotation service component is an additional feature in this gateway to process each sensor message received from the sink node before forwarding it further to the gateway interface. The annotations service provides domain-specific ontologies to annotate the incoming data. As a result, other services can then read this data using the same domain-specific ontologies. The annotation itself is based on Open Geospatial Consortium (OGC) defined standards. [DSA15, p. 316]

2.1.4 Design and Implementation of a Smart IoT Gateway

The paper "Design and Implementation of a Smart Iot Gateway" suggests a novel adjustable smart IoT gateway with three key advantages. To begin with, the gateway has a pluggable design, whose modules with different communication protocols can be customized and plugged in according to different networks. Second, it has a set of uniform external interfaces that are well-suited for flexible software development. Finally, it offers a flexible methodology for converting sensor data into a standardized format. [Guo+13, p. 720]

The proposed result packages data on the gateway in its data format to overcome different data problems. The data communication frame structure is shown in figure 2.7. [Guo+13, p. 722]

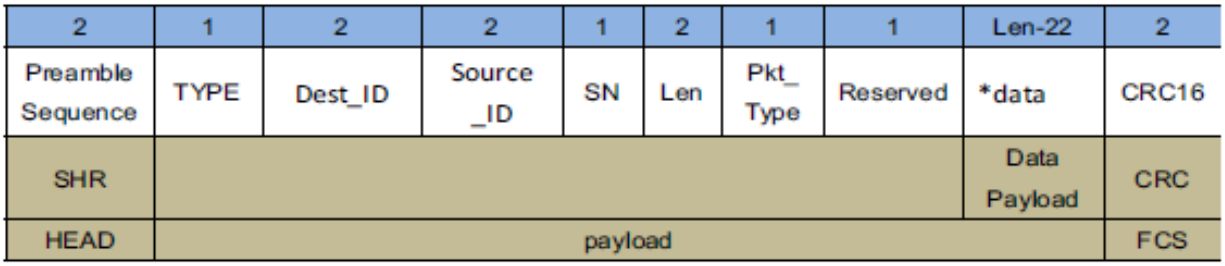


Figure 2.7: Data communication frame structure. [Guo+13, p. 722]

The function of different parts of the frame are explained as follows. [Guo+13, p. 722]

- *Preamble sequence*: communication data package synchronization byte, including leader character and frame beginning interval character. The two bytes are defined as 0x AA 44
- *Type*: indicating the data is uplink (0x01) or downlink (0x02).
- *Dest_ID*: Destination adress, 2 bytes.
- *Source_ID*: Source address, 2 bytes.
- *Seq*: Message numbers (distinguish the proper arrival of messages), 1 byte.
- *Length*: Length of the whole message.
- *Pkt_Type*: The type of packages, including Pkt_Type and Pkt_SubType, 1 byte.
- *Reserved*: Redundancy can be used to distinguish the different sensors on the same mac address, 1 byte.
- *FCS*: CRC16, inspect the integrity of the data in the packages.

2.1.5 An experimental study of a reliable IoT Gateway

The paper "An experimental study of a reliable IoT Gateway" is another paper that focuses on IoT compatibility by defining a reliable and self-configurable IoT gateway that is developed in a laboratory testbed.

The proposed result is similar to the client/server model of HTTP and uses CoAP for device-to-device (D2D) communication. An operation is equivalent to that of HTTP and is sent by a client to request action on a resource identified by a Uniform Resource Identifier (URI) on a server. The server then sends a response with a response code. The solution makes use of the *GET*, *PUT*, *POST*, *DELETE*, *OBSERVE* and *INIT* operations in a similar manner to HTTP with semantics. The details of each operation are described as follows: [KC18, p. 131]

- *GET* retrieves a representation for the information that currently corresponds to the resource identified by request URI. If the request includes an accept option, this indicates the preferred content format of the response.
- *POST* requests that the representation enclosed in the request be processed. The actual function performed by this method is determined by the origin server and dependent on the target device. It usually results in a new IoT device being created or the target device being updated.
- *PUT* requests that the resource identified by request URI be updated or created with the enclosed representation. The representation format is specified by the media type and content coding given in the option of the content format if provided.
- *DELETE* requests that the resource identified by request URI be deleted. A 2.02 (deleted) response code should be used on success or if the resource did not exist before the request.
- *OBSERVE* fetches and registers as an observer for the value of an IoT device. The handling of observation registration is application-specific. It should not be assumed that a device is observable or can handle any specific number of observers. The registration is considered successful if the master server responds with a success (2.12) code.
- *INIT* brings a device and its attribute information to the master server. Normally, this operation is generated by a sensor or an IoT machine and sends this information to the IoT gateway. If the IoT gateway receives this message, the gateway updates its device and attribute table based on the received data.

2.1.6 A Survey on IoT Security: Application Areas, Security Threats, and Solution Architectures

The paper "A Survey on IoT Security: Application Areas, Security Threats, and Solution Architectures" presents a detailed review of the security-related challenges and sources of threat in IoT applications.

Basically, according to the author, every IoT application can be divided into four layers: (1) sensing layer, (2) network layer, (3) middleware layer, and (4) application layer. Each layer uses diverse technologies that bring several issues and security threats. For example, figure 2.8 shows the possible attacks on these four layers and the additional broad layer of the Gateway. [Has+19, p. 5]

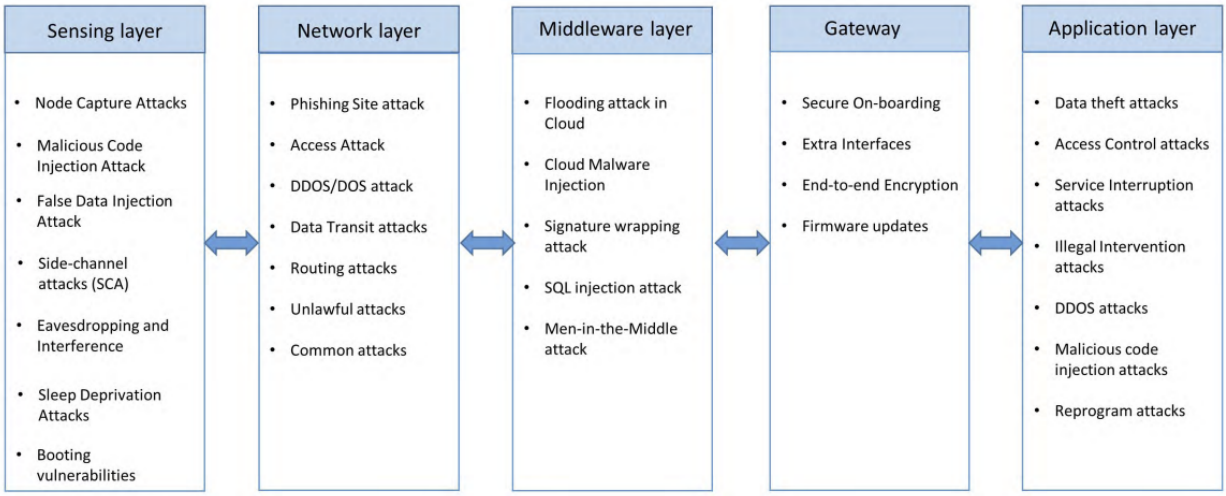


Figure 2.8: Type of attacks on IoT. [Has+19, p. 6]

The sensing layer mainly deals with physical IoT sensors and actuators. In this layer, various sensing technologies are used like Radio-frequency identification (RFID), Global Positioning System (GPS) or Wireless Sensor Networks (WSN). Major security threats in this layer are explained in the following. [Has+19, p. 6]

- *Node capturing*: The attacker may try to capture or replace a node in the system with malicious code. The node may appear to be part of the system but is controlled by the attacker.
- *Malicious code injection attack*: The attack involves the attacker injecting some malicious code into the node's memory. Generally, the firmware or software of nodes is upgraded on the air, giving attackers the chance to inject malicious code.
- *False data injection attack*: Once the node is captured, the attacker may inject erroneous data onto the system. This may lead to false results or malfunctioning of the whole application. This attack can also be used to cause a Distributed denial-of-service attack (DDoS) attack.
- *Side-channel attacks*: Apart from direct attacks on the nodes, various side-channel attacks may lead to sensitive data leaking. The microarchitectures of processors, electromagnetic emanation, and power consumption reveal adversaries' sensitive information.
- *Eavesdropping and interference*: The attackers may eavesdrop and capture the data during different phases like data transmission and authentication.
- *Sleep deprivation attacks*: In such attacks, the adversaries try to drain the battery of the low-powered edge devices. This leads to denial of service due to a dead battery. This can be done by running infinite loops in the edge device or artificially increasing these devices' power consumption.
- *Booting attacks*: If there is no secure boot activated on edge devices, the attacker may take advantage of this vulnerability and attack the node devices when they are being restarted.

The critical function of the network layer is to transmit data. Therefore, significant security issues in this layer are listed below. [Has+19, p. 6 f.]

- *Phishing site attack*: Phishing attacks often refer to attacks with minimal effort targeting several devices. There is a possibility of encountering phishing sites on the internet. Once a user's account and password are compromised, the whole IoT environment might be too.
- *Access attack*: This type of attack in which an unauthorized person gains access to the network. The attacker can stay in the network undetected for a long duration. This kind of attack aims to steal valuable data or information.
- *Denial-of-service attack (DoS)*: In this kind of attack, the attacker floods the target servers with many unwanted requests. If multiple sources are used, such an attack is termed DDoS. Such attacks are not limited to IoT, but due to the heterogeneity and complexity of IoT networks, it is prone to such attacks.
- *Data transit attacks*: In IoT applications, there is a lot of data movement between sensors, actuators or cloud. Different connection technologies are used in such data movements, and therefore IoT applications are susceptible to data breaches.
- *Routing attacks*: In such attacks, malicious nodes in an application may try to redirect the routing paths during data transit. Sinkhole attacks are a specific kind of routing attack in which an attacker advertises an artificial shortest routing path and attracts node route traffic through it. A worm-hole attack is another attack that can become a severe security threat if combined with other attacks such as sinkhole attacks. A worm-hole is an out of band connection between two nodes for fast packet transfer. For example, an attacker can create a worm-hole between a compromised node and a device on the internet and try to bypass the basic security protocols in an IoT application.

The role of the middleware in IoT is to create an abstraction layer between the network layer and application layer. Middleware can also provide powerful computing and storage capabilities. Although the middleware layer helps provide a reliable and robust IoT application, it is also susceptible to various attacks. These attacks can take control of the entire application by infecting the middleware. Various possible attacks are discussed as follows. [Has+19, p. 7]

- *Man-in-the-middle attack*: The MQTT protocol uses a publish-subscribe model of communication between clients and subscribers using the MQTT broker, which effectively acts as a proxy. If the attacker can control the broker and become a man-in-the-middle, then it is possible to get complete control of all communication without knowing the clients.
- *SQL injection attack*: The middleware is also susceptible to SQL injection attacks. The attacker can embed malicious SQL statements in a program in such attacks. Then, the attacker can obtain private data and even alter records in the database.
- *Signature wrapping attack*: The attacker breaks the signature algorithm and can execute operations or modify eavesdropped messages in a signature wrapping attack.
- *Cloud malware injection*: In cloud malware injection, the attacker can obtain control, inject malicious code, or inject a virtual machine into the cloud. The attacker pretends to be a valid service by creating a virtual machine instance or a malicious service module.
- *Flooding attack in cloud*: This attack works almost the same as DoS attacks in the cloud and affects the QoS. The attacker continuously sends multiple requests to the service for depleting cloud resources.

Gateway is a broad layer that has a vital role in connecting multiple devices, people, things and cloud services. IoT systems today are heterogeneous, including LoraWan, ZigBee, Z-Wave and TCP/IP stacks with many gateways in between. Some of the security challenges for IoT gateway are discussed below. [Has+19, p. 8]

- *Secure onboarding*: When a new device is installed in the system, it is imperative to protect encryption keys. Gateways act as an intermediary between the new devices and the managing services, and all the keys pass through the gateways. As a result, the gateways are susceptible to man-in-the-middle attacks and eavesdropping to capture the encryption keys, especially during onboarding.
- *Extra interfaces*: Minimizing the attack surface is an essential strategy that needs to be kept in mind while installing IoT devices. Only necessary interfaces and protocols should be implemented by an IoT gateway manufacturer. In addition, to avoid backdoor authentication and information breach, some of the services and functionalities should be restricted for end-users.
- *End-to-end encryption*: True end-to-end application layer security is required to ensure the confidentiality of the data. The application should not let anyone except the unique recipient decrypt the encrypted messages. Although ZigBee and Z-Wave protocols support encryption, this is not end-to-end encryption because, in order to translate the information from one protocol to another, the gateways are required to decrypt and re-encrypt messages.
- *Firmware updates*: Most IoT devices are resource-constrained, and therefore they do not have a user interface or the computation power to download and install firmware updates. Generally, gateways are used to download and apply firmware updates. The current and new firmware versions should be recorded, and the validity of the signatures should be checked for secure firmware updates.

The application layer directly deals with and provides services to end-users. Many IoT applications also consist of a sublayer between the network layer and application layer, usually termed as an application support layer or middleware layer. Major security issues encountered by the application layer are discussed below. [Has+19, p. 8]

- *Data thefts*: IoT applications deal with a lot of critical and private data. The data in transit is even more vulnerable to attacks than data at rest, and in IoT applications, there is a lot of data movement. Data encryption, data isolation, user and network authentication, and privacy management, are some techniques and protocols used to secure IoT applications against data thefts.
- *Access control attacks*: Access control is an authorization mechanism that allows only legitimate users or processes to access the data or account. Once the access is compromised, the complete application becomes vulnerable to attacks.
- *Service interruption attacks*: In the existing literature, these attacks are also referred to as illegal interruption attacks or DDoS attacks. There have been various instances of such attacks on IoT applications. Such attacks deprive legitimate users of using the services of IoT applications by artificially making the servers or network too busy to respond.
- *Malicious code injection attacks*: If the system is vulnerable to malicious scripts and misdirections due to insufficient code checks, that would be the first entry point that an attacker would choose. Generally, attackers use XSS (cross-site scripting) to inject malicious scripts into an otherwise trusted website. A successful XSS attack can result in the hijacking of an IoT account and can paralyze the IoT system.
- *Sniffing attacks*: The attackers may use sniffer applications to monitor the network traffic in IoT applications. This may allow the attacker to gain access to confidential user data if there are not enough security protocols to prevent it.
- *Reprogram attacks*: If the programming process is not protected, the attackers can reprogram the IoT objects remotely. This may lead to the hijacking of the IoT network.

Additionally, the paper also discussed the existing and upcoming solutions for IoT security threats, including blockchain, fog computing, edge computing, and machine learning. However, this is not in the scope of the following work and will not be summarised further.

2.2 State of the Art

While the theoretical point of view was already explained, this chapter will revisit the state of the art in the economic sector. Although no universal standard for IoT gateways has been defined in the literature, some solutions try to solve the problems mentioned.

Particular attention must be paid to the open-source applications since the closed source solutions solve some problems but are usually tied to the manufacturer. Another disadvantage of these applications is that they often cannot be customized to meet specific requirements. This usually results from the fact that the source code is not accessible and depends on the manufacturer.

For this reason, mainly open source applications are summarized below. In order to evaluate the suitability of these applications, they are examined with the help of the defined mandatory objectives.

2.2.1 Eclipse Kura

Eclipse Kura is an extensible open-source IoT Edge Framework based on Java/OSGi. Kura offers API access to the hardware interfaces of IoT Gateways (serial ports, GPS, Watchdog, GPIO or I2C). It features ready-to-use field protocols (including Modbus or OPC-UA), an application container, and a web-based visual data flow programming to acquire data from the field, process it at the edge, and publish it to leading IoT Cloud Platforms through MQTT connectivity. [Fou22a]

Figure 2.9: Eclipse Kura [Fou22a]

While Eclipse Kura already partially solves the protocol conversion for devices, only MQTT is offered on the cloud side. In addition, only one device is created in the cloud, the gateway itself. However, the gateway can still call all endpoints and thus modify the data of other devices, but it cannot automatically register devices due to the implementation. There is no middleware abstraction either, as only the cloud platforms such as Eurotech Everyware Cloud, Amazon AWS IoT Core, Azure IoT Hub, Eclipse Kapua and Eclipse Hono are supported. At the moment, nothing can be found in the documentation about resource management. The message exchange is defined via so-called “wires”. The message is then processed in a simple FIFO queue. [Fou22a]

2.2.2 EdgeX

EdgeX Foundry is a highly flexible, scalable and vendor-neutral open-source framework hosted by the Linux Foundation that facilitates the development of data collection, analytics and cloud connector services. The applications act as cloud-to-edge middleware with a plug-and-play distributed microservice architecture. [Fou22e]

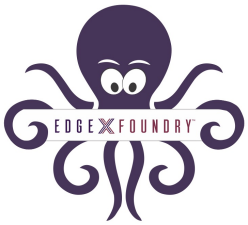


Figure 2.10: EdgeX Foundry [Fou22e]

EdgeX Foundry was designed to provide broad industry support. For this reason, not only many device protocols are supported, but also different protocols on the cloud side. Furthermore, EdgeX Foundry's architecture enables middleware abstraction, so nobody depends on specific cloud providers. Also, the application supports resource management mechanisms by measuring parameters such as round trip time to achieve the lowest possible latency. In addition, EdgeX can identify which data needs to be sent to the cloud and processed cost-effectively on edge. However, there seem to be no prioritization options for individual packets. In device management, nevertheless, there is static and automatic device discovery, so devices no longer need to be added manually. [Fou22e]

2.2.3 OpenHAB

The open Home Automation Bus (openHAB) is an open-source home automation software based on Java and OSGi. It is deployed on-premise and connects to devices and services from different vendors. As of 2019, openHAB already included around 300 bindings available as OSGi modules. [Com22]

Figure 2.11: OpenHAB [Com22]

While openHAB is a home automation software, that does not mean it cannot be used for IoT purposes. For example, support for different device protocols and services has been implemented, yet integrating different cloud applications is missing. In fact, openHAB only focuses on the openHAB Cloud. There seems to be no concrete resource management or traffic optimization mechanism either. Nevertheless, the application has a device discovery function so that devices can be added automatically. [Com22]

2.2.4 ThingsBoard IoT Gateway

The ThingsBoard IoT Gateway is an open-source solution to integrate devices with ThingsBoard. However, the IoT Gateway is built on top of Python and is different from similar projects that leverage OSGi technology. The gateway supports custom connectors to connect to new devices or servers and custom converters for processing data from devices. [Aut22]

Figure 2.12: ThingsBoard [Aut22]

The ThingsBoard IoT Gateway also solves the protocol translation on the device side, yet it also lacks different cloud protocols. This fact leads to inevitably having to work with the ThingsBoard solution. Furthermore, there is no indication of resource management or a traffic optimization mechanism. However, the gateway can create devices automatically, so there is no need to create them manually. [Aut22]

2.2.5 Summary

The individual results of the applications presented are summarized in the following table.

Gateway	Protocol conversion	Device management	Middleware abstraction	Ressource management	Traffic optimization.
Eclipse Kura	o	o	-	-	-
EdgeX	+	+	+	o	o
openHAB	o	+	-	-	-
ThingsBoard Gateway	o	+	-	-	-

Table 2.2: Summary of the presented IoT gateways

Legend: -: Not mentioned in the documentation or not implemented; o: Partially implemented; +: Fully implemented;

From the presented application, it is clear that today's gateways implement not all necessary functionalities. Only the EdgeX Foundry application covers most of the requirements for an automated gateway, but there are also minor points that are not covered by this application. In addition, no standard has been developed to date, so any regulations do not bind these applications.

While EdgeX Foundry could already be used for the application development in this thesis, it is also about showing how to solve the mentioned problems. For this reason, the development of a new gateway offers itself to enrich the literature so that in the future, a new standard may arise which solves the problems mentioned.

2.3 Classification of the proposed type of gateway

After presenting the literature on gateways, the question arises about how the proposed gateway can be classified based on its functionalities. It is important to note that this classification is only based on the mentioned review paper. This means that these classifications are not standardised.

Based on the objectives presented in chapter 1.2 on page 3, the gateway presented can probably be classified as a semi-automated gateway. While the device registration in the passive gateway has to be done manually, the proposed gateway already has automatic functionalities for adding these. The functionalities do not yet allow device discovery but simplify the process significantly. In addition, the gateway has some functionalities to manage itself.

There are still some steps for a fully automated gateway, as there are still many points that have not yet been addressed in this thesis.

Chapter 3

Protocols

Due to the fact that the elaboration is based on a proof-of-principle scenario, not all existing protocols can be implemented in the gateway. For this reason, three unique protocols were selected for evaluating and testing the gateway. In addition, a comparative study of IoT protocols was used to avoid randomly choosing any protocols. In this study, protocols in the network and application layers were compared with each other.

Since protocols are generally better or worse suited for specific use cases, the following use cases were selected:

1. Devices without limitations send sensor data to the gateway.
2. Devices without limitations stream data (for example, camera data) to the gateway.
3. Devices with limitations in battery life and processor performance send sensor data to the gateway.

Based on the selected use cases, the first two can be solved with protocols from the application layer due to no defined restrictions, and the third by a protocol from the network layer.

All compared application protocols are shown in the figure 3.1 on the following page.

	CoAP	XMPP	RESTful HTTP	MQTT	WebSocket	AMQP	DDS
Standard	IETF	IETF	REST	OASIS IBMs	HTML 5s	O ASIS	OMG DDS
Technologie	XML	XML	XML, HTML, JS ON	Irrespective of implementation language	XML JSON	Irrespective of implementation language	C, C ++, C #, Java, Scala, Lua, Pharo and Ruby
Transport	UDP	TCP	TCP	TCP	TCP	TCP	TCP /UDP
Architecture	response request	Publish / Subscribe Request / Response	response request	Publish/ Subscribe Request / Response	publish subscribe	Pub/Sub	Pub/Sub
2G, 3G, 4G Adequacy (noeuds 1000s)	Excellent	Excellent	Excellent	Excellent	-	Excellent	-
Aptitude LLN (noeuds 1000s)	Excellent	Just	Just	Just	-	-	-
Calculate resources	10Ks RAM / Flash	10Ks RAM / Flash	10Ks RAM / Flash	10Ks RAM / Flash	-	-	-
Application	Utility Area Networks	Remote management of consumer products	Smart Energy Profile 2 (basic energy management, home services)	Extending Enterprise Messaging to IoT Applications	Real-time application: Real-time dashboards, Teamwork	Hybrid applications:the integration of computer systems following a merger.	Distributed applications
Security and QoS	Both	Security	Both	Both	Security	Both	QoS
Quality factor	Reliability Authentication, Integrity, Confidentiality	Efficiency, Reusability	-	Reliability	Reliability	Efficiency, Elexibility, Interoperability.	Excellent quality of service levels Reliability, security, Urgency priority, Durability, reliability, flexibility, and performance
Advantages	•Multicast support •Low overhead •Minimizing complexity of mapping with HTTP •Communication models flexibility •Low latency	•Real-time •Low latency •Easily understandable •Easly extensible •Any XMPP server may be isolated	•Application is easy to maintain • No client state management on the server •No client state management on the server	•Easy to implement •Useful for connections with remote location •Small code footprint •Lightweight •Asymmetric client - server relationship	•Simplifies the web communication and co – network compatibility •Connection management	•Complex message queuing implementations •ISO standard •High routing reliability and security •Easly extensible •Symmetric client-server relationship	•Real-time monitoring of quality of service, •Decentralized architecture, •Dynamic detection of broadcasters and subscribers
Disadvantages	•Doesn't enable communication level security •Few existing libraries and solution support	•Heavy data overhead •Not suitable for embedded IoT applications	The need for the customer to locally store all data needed to conduct a query	•No error-handling •Hard to add extensions •Basic message queuing implementations •Doesn't address connection security	•Specific hardware requirements •No useful open source implementations targeted at embedded systems	•Bigger packet size than other protocols. •Doesn't support Last Value Queue (LVQ)	-

Figure 3.1: IoT application protocols [Elh+18, p. 4]

Based on the study, the two protocols, MQTT and CoAP, are emerging as leading lightweight protocols in the IoT market. MQTT is the preferred protocol for sending information constantly, while CoAP is the preferred protocol if document transfer is required. Therefore, MQTT was selected to solve the first use case. [Elh+18, p. 4]

For the second use case, WebSocket was chosen because, according to the study, this protocol is ideally suited for real-time applications, such as data streaming. [Elh+18, p. 4]

For the third use case, protocols from the network layer should be considered since they usually generate a low overhead and have been developed specifically for such restrictions. In this case, the fact that we are using a network protocol does not mean that we can use an application protocol on top of it. Most application protocols require a TCP/IP connection, which is not the case for most network protocols here. The network protocols also have their application layers to run programs on. The network protocols compared are listed in the figure 3.2 and 3.3 on the following page.

Protocol Critères	Wifi	Bluetooth	LoRa	Zigbee	Z-Wave
Specification	Based on 802.11n	Bluetooth 4.2 core specification	LoRaWAN	ZigBee 3.0 based on IEEE802.15.4	Z-Wave Alliance ZAD12837 / ITU-T G.995
Network type	LAN WPAN/P2P	LAN	LAN	LAN	LAN
Topology	Star	Star	Star & Star of Star	Mesh,Star,Tree	Mesh
Power	Low-High	Low	Very Low	Very Low	Very Low
Data Rate	Up to 1.3Gbps	2.1Gbps	0.3Gbps to 100 kbps	250 Gbps	40 Gbps
Frequency band	54 Mb/s	2.4 G Hz	Various	868/915 M Hz, 2.4 G Hz	900MHz (ISM)
Modulation Technique	BPSK, QPSK, OFDM, M-QAM	GFSK, CPFSK, 8-DPSK, π/4-DQPSK	GFSK	BPSK, O-QPSK	BFSK, GFSK
Spread Spectrum	MC-DSSS, CCK, OFDM	FHSS	Chirp	DSSS	DSSS
Range	Up to 100m	<100m	3-5km urban area	10-20m	30km
Security	WPA and WPA2	Shared secret	Per-device AES128 keys, AES128 secret key	CBC-MAC (ext. of CCM)	Data encryption
Cost	Low	Low	Low	good	Effective
Risk of data collision	low	High	-	Medium	-
Energy needed	-	High	-	Medium	-
Market Adoption	Yes	yes	Yes	yes	Yes
Application	Any device with cellular connectivity	Network for data exchange headset	Smart city, sensor networks, industrial automation application	Senor networks Industrial automation	Residential lighting & automation
Network size	Medium	Small	Medium large	Very large	Large

Figure 3.2: IoT network protocols part one [Elh+18, p. 3]

Protocol Critères	Cellulaire	NFC	Sigfox	Neul	6LowPan
Specification	GSM/GPRS/EDGE (2G), UMTS/HSPA (3G), LTE (4G)	ISO/IEC 18000-3	Sigfox	neul	IPv6 over IEEE802.15.4
Network type	MAN	P2P	LPWAN	WAN	LAN
Topology	Mesh	Mesh	Star	Mesh	Mesh,Star
Power	High	Very Low	Low	Low	Very Low
Data Rate	Up to 1Gbps	424 kbits /s	10-1000bps	Few bps up to 100kbp	200 Gbps
Frequency band	850/900; 1800/1900 MHz	13.56 MHz	900MHz	900MHz (ISM), 458MHz (UK), 470-790MHz (White Space)	2.4 GHz
Modulation Technique	GMSK, 8PSK	ASK	UNB LTN	-	O-QPSK o Min LIFS Period
Spread Spectrum	TDMA, DSSS	FHSS	PBSS	FHSS	CSS
Range	Where signal reach	5cm	30-50km	10km	10-20m
Security	-	-	-	-	SNMP
Cost	Less and vice versa	Chip	relatively low	-	Hight
Risk of data collision	-	-	-	-	-
Energy needed	-	-	-	-	-
Market Adoption	yes	yes	yes	yes	Yes
Application	Use in wifi,ADSL,broad basnd ,digital TV&Radio	Payment Transactions, Business Transactions, Contextual Information,	Smart Grid	home automation	Senor networks Industrial automation
Network size	Very large	Small	Medium	Medium	Very large

Figure 3.3: IoT network protocols part two [Elh+18, p. 3]

Accordingly to the study, ZigBee and Bluetooth are the most widely used standards in IoT. However, ZigBee is more suited to the selected use case because it is specially designed for sensor networks and has a lower power consumption than Bluetooth. Therefore the third protocol selected was ZigBee. [KKC17, p. 3]

Summarized the three protocols MQTT, WebSocket and ZigBee were used in this elaboration. Therefore, these protocols will be explained in more detail in the following.

3.1 MQTT

Message Queuing Telemetry Transport (MQTT) is a client-server publish/subscribe messaging transport protocol. It is lightweight, open, simple, and easy to implement. These characteristics make it ideal for use in many situations, especially in constrained environments such as for communication in Machine to Machine (M2M) and IoT contexts, where a small code footprint is required or the bandwidth is limited. The protocol runs over TCP/IP or other network protocols that provide ordered, lossless, bidirectional connections like WebSocket. General features of these protocols are comprehended in the following. [mqtt-v5.0, p. 1]

- *Publish/subscribe*: In MQTT, clients are publishing messages and subscribing to topics. This is commonly considered a publish/subscribe model. If a client publishes a message to a topic, all subscribers of this topic receive the published message.
- *Quality of Service levels*: The protocol describes three Quality of Service levels to ensure that messages are exchanged correctly according to a predefined behaviour.
  - *QoS0 (At most once)*: Messages are delivered according to the best efforts of the operating environment. This means message loss can occur. In general, this level could be used with ambient sensor data where it does not matter if an individual reading is lost as the next one will be published soon after.
  - *Qos1 (At least once)*: Messages are assured to arrive, but it is possible to deliver duplicates of the message. Some logical operations on the receiver could lead to unwanted behaviour.
  - *Qos2 (Exactly once)*: Messages are assured to arrive exactly once. This level could be used with billing systems where a duplicate or lost message could apply incorrect charges.
- *A message transport that is agnostic to the content of the payload*
- *A small transport overhead and protocol exchanges minimized to reduce network traffic*
- *A mechanism to notify interested parties when an abnormal disconnection occurs*

The MQTT protocol operates by exchanging a series of MQTT Control Packets in a defined way.

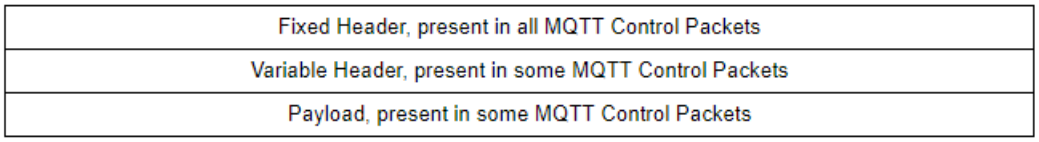


Figure 3.4: MQTT control packet [mqtt-v5.0, p. 21]

Each MQTT control packet contains fixed headers, as shown below.

Bit	7	6	5	4	3	2	1	0
byte 1	MQTT Control Packet type				Flags specific to each MQTT Control Packet type			
byte 2...	Remaining Length							

Figure 3.5: MQTT fixed header [mqtt-v5.0, p. 21]

There are fourteen control packets in total in the current version of the protocol.

Name	Value	Direction of flow	Description
Reserved	0	Forbidden	Reserved
CONNECT	1	Client to Server	Connection request
CONNACK	2	Server to Client	Connect acknowledgment
PUBLISH	3	Client to Server or Server to Client	Publish message
PUBACK	4	Client to Server or Server to Client	Publish acknowledgment (QoS 1)
PUBREC	5	Client to Server or Server to Client	Publish received (QoS 2 delivery part 1)
PUBREL	6	Client to Server or Server to Client	Publish release (QoS 2 delivery part 2)
PUBCOMP	7	Client to Server or Server to Client	Publish complete (QoS 2 delivery part 3)
SUBSCRIBE	8	Client to Server	Subscribe request
SUBACK	9	Server to Client	Subscribe acknowledgment
UNSUBSCRIBE	10	Client to Server	Unsubscribe request
UNSUBACK	11	Server to Client	Unsubscribe acknowledgment
PINGREQ	12	Client to Server	PING request
PINGRESP	13	Server to Client	PING response
DISCONNECT	14	Client to Server or Server to Client	Disconnect notification
AUTH	15	Client to Server or Server to Client	Authentication exchange

Figure 3.6: MQTT control packet type [mqtt-v5.0, p. 21 f.]

The typical MQTT architecture can be divided into two main components, as shown in figure 3.7. The Client could be a publisher or subscriber, and it always establishes the network connection to the Broker. The Broker controls the distribution of information and is mainly responsible for receiving all messages from publishers, filtering them, and deciding who is receiving which message. [SM17, p. 3]

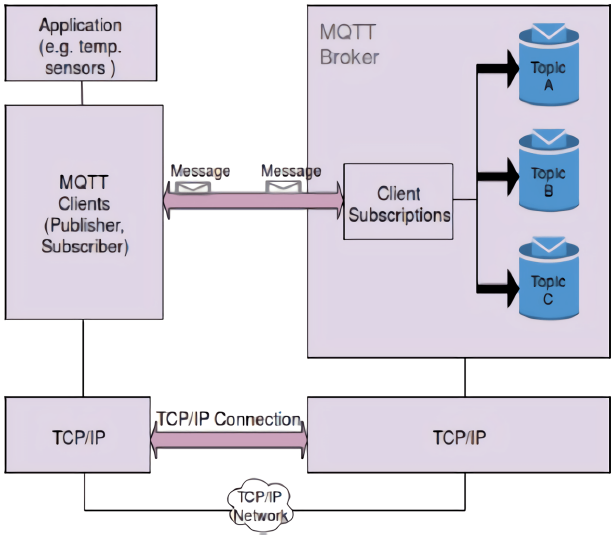


Figure 3.7: MQTT architecture [SM17, p. 3]

The concrete sequence of this process is shown in the following figure 3.8.

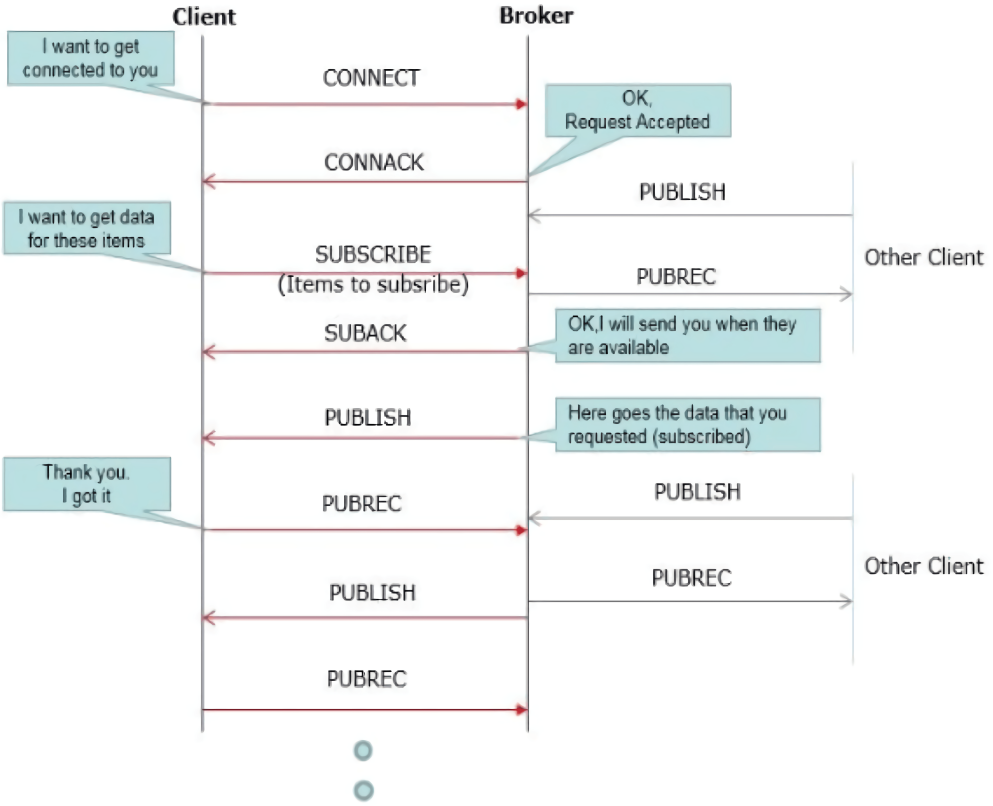


Figure 3.8: MQTT sequence [SM17, p. 3]

3.2 ZigBee

ZigBee is a specification for wireless mesh networking capable of supporting more than 64.000 devices on a single network. The specification was released by the ZigBee Alliance, which was founded in 2001 by more than 230 companies. It was designed to solve many problems for WSN, for example, complexity, large power dissipation, short distance, and networking on a small scale. The specification has two implementation options or Feature Sets: ZigBee and ZigBee PRO. The ZigBee Pro option is the most popular choice of developers because it is focused on low data rates and low power consumption. However, both options are designed to interoperate, ensuring long-term use and stability. ZigBee enhances the IEEE 802.15.4 standard by adding network and security layers and an application framework. This foundation can be used to create a multi-vendor interoperable solution. [SP12, p. 2-3]

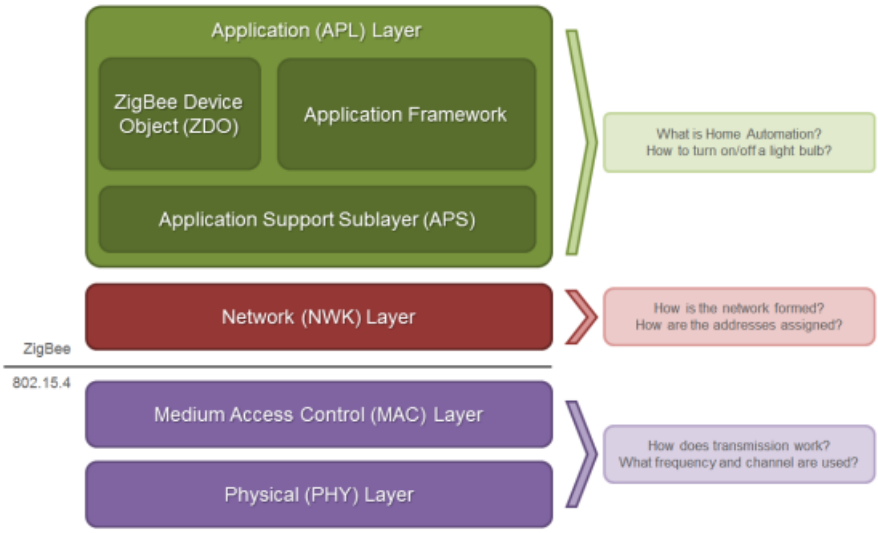


Figure 3.9: ZigBee stack layers [Inc20, p. 77]

3.2.1 Device types

ZigBee devices are a combination of logical and physical device types.

Logical device types

There are three nodes in a ZigBee System, the Coordinator, Router and End devices. [SP12, p. 4]

- The *Coordinator* forms the root of the network tree and might bridge to other networks. There is always exactly one coordinator in each network. It is responsible for initiating the network and selecting network parameters such as radiofrequency, unique network identifier, and setting optional parameters. It can also store information about networks like security keys. [SP12, p. 4]
- The *Router* acts as intermediate nodes, relaying data from other devices. Routers can connect to existing networks and accept connections from other devices and be some re-transmitters to the network. [SP12, p. 4]
- The *End Device* can collect various information from sensors. They have enough functionality to communicate with routers or coordinators but cannot relay data from other devices. This reduced functionality reduces their costs, making them perfect for low energy consuming tasks. Additionally, these devices do not have to stay awake the whole time, while the other types have to. [SP12, p. 4]

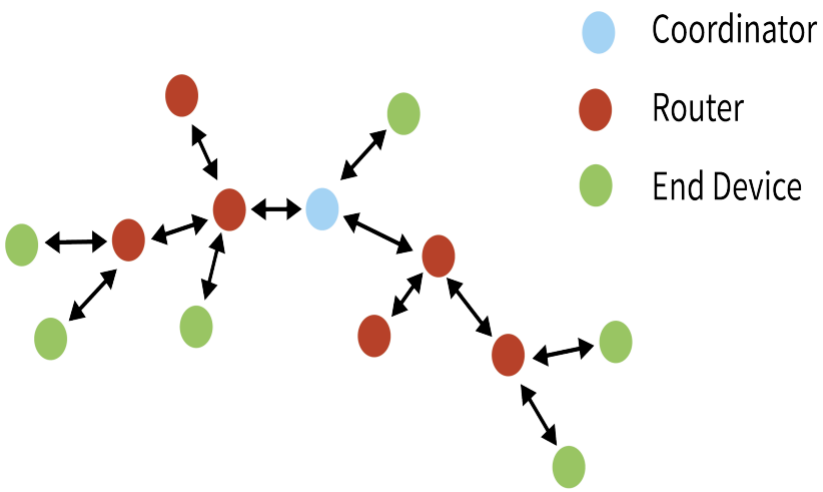


Figure 3.10: ZigBee device types [Inc20]

Physical device types

Based on data processing capabilities, two types of physical devices are provided in IEEE 802.15.4, Full Function Devices (FFD) and Reduced Function Devices (RFD). FFD can perform all available operations within the standard, including routing mechanism, coordination and sensing tasks. The FFD plays the role of the coordinator, router or end device in a ZigBee network. RFD, on the other hand, implements a limited version of the IEEE 802.15.4 protocol. These devices do not route packets and must be associated with an FFD. These devices are mainly sensor actuators collecting temperature data, monitoring lighting conditions, or controlling external devices. [SP12, p. 4-5]

Access Modes

In ZigBee, there are two access modes, Beacon and Non-Beacon. In a Non-Beacon network, any node can send data over the network when the channel is free. In Beacon networks, nodes can only transmit data in a specified time slot. Here, the coordinator assigns each device to a guaranteed time slot. For this to be possible, all devices must be synchronized, this is done by sending a beacon signal from the coordinator. If it does not send this signal, the network cannot have guaranteed time slots and thus cannot be a beacon network. The energy consumption in non-beacon networks is lower because the devices do not have to be awake as often. [SP12, p. 5]

3.2.2 Protocol architecture

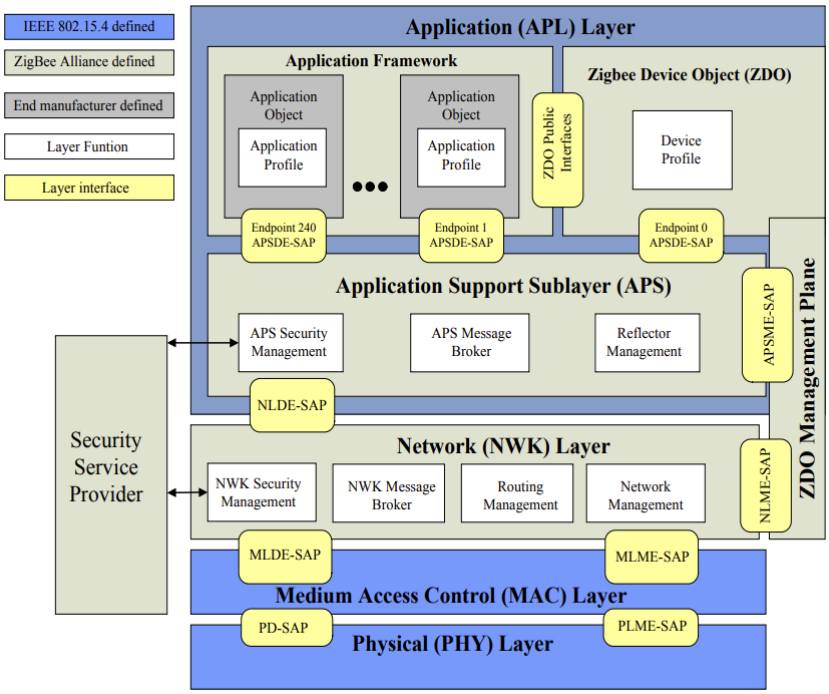


Figure 3.11: ZigBee architecture [Liu09, p. 3]

Physical Layer

The physical layer is a part of the IEEE 802.15.4 standard, the closest part to the hardware. It controls and communicates with the radio transceiver directly. It manages all tasks like hardware access, channel selection, link quality estimation, energy detection measurement and free channel evaluation to support channel selection. Three frequency bands are supported: a 2.45 GHz band using 16 channels, a 915 MHz band using ten channels, and an 868 MHz band using one channel. All three channels are using Direct Sequence Spread Spectrum (DSSS) access mode. [SP12, p. 5-6]

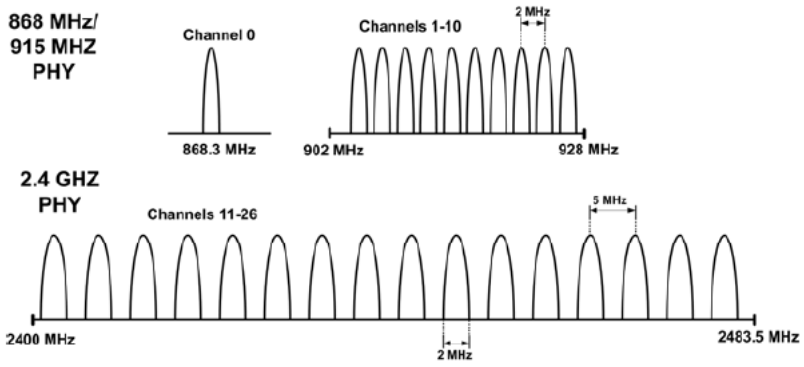


Figure 3.12: IEEE 802.15.4 operating frequencies and bands [STP13, p. 641]

MAC Layer

The Media access control (MAC) layer forms an interface for the physical and network layers. The MAC layer is responsible for generating beacons and synchronizing the devices to the beacon signals. It is also responsible for association and dissociation. [SP12, p. 6]

Network Layer

The network layer forms the interface between the application and the MAC layer. This layer is responsible for network formation and routing. Routing is choosing paths to forward messages through other nodes based on a routing table. In ZigBee, the coordinator or router performs the actual route discovery. In addition, the layer offers security features and the possibility to maximize battery life. Based on the IEEE 802.15.4 standard star, mesh or tree networks are supported, as shown in figure 3.13. [SP12, p. 6]

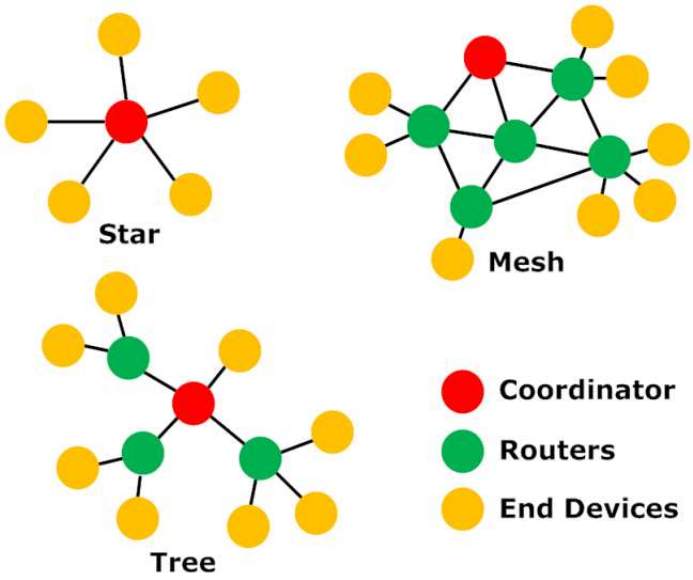


Figure 3.13: Zigbee Based Network Topologies [Sal12, p. 4]

Application layer

The application layer is the highest. However, ZigBee divides this layer into three further sub-layers, the Application Support Sublayer, the ZigBee Device Objects, and the Application Framework with manufacturer-defined Application Objects. [SP12, p. 7]

- The function of an *Application Object (APO)* is to implement the processing logic for the ZigBee application. In other words, APO is the basic unit of an application program. Each application object is assigned to a unique endpoint that other APO's can use to interact with it. There can be up to 240 application objects in a single ZigBee device. [SP12, p. 7]
- The *Zigbee Device Object (ZDO)* is the key definition of ZigBee, which addresses three main operations; service discovery, security and binding. [SP12, p. 7]
- The *Application Support Sublayer (APS)* provides an interface between the network and the application layers through a broad set of services. The sub-layer processes outgoing and incoming frames to securely transmit or receive the frames and establish or manage the cryptographic keys. [SP12, p. 7]
- The *Security service provider* provides a security mechanism for the network and application support layers because both of them are responsible for protecting their frames. The security services include methods for key establishment, key transport, frame protection and device management. [SP12, p. 7]

3.3 WebSocket

WebSocket is a data protocol that exposes TCP on a higher abstraction level to be used almost directly by Web browser applications. The IETF standardized it in 2011 with RFC 6455. [Sar18, p. 3]

Historically, creating web applications that need bidirectional communication between a client and a server has required an abuse of HTTP to poll the server for updates while sending upstream notifications as distinct HTTP calls. This results in a variety of problems: [RFC6455, p. 4]

- The server is forced to use many different underlying TCP connections for each client. One connection for sending information to a client and one for each incoming message.
- The connection has a high overhead, with each client-to-server message containing an HTTP header.
- The client-side script is forced to maintain a mapping from outgoing messages and replies.

To solve the lack of asynchronous communication in an HTTP environment and provide a long-living full-duplex connection. The WebSocket protocol was defined to use a single TCP connection for traffic in both directions. [RFC6455, p. 4] For this reason, WebSocket is also suitable for IoT use cases where a real-time bidirectional interaction with the devices is desired. [Sar18, p. 3]

The WebSocket protocol was designed to replace the existing bidirectional communication technologies that use HTTP as a transport layer to leverage the existing infrastructure (proxies, filtering, authentication). Older technologies were implemented as trade-offs between efficiency and reliability because HTTP was not initially meant to be used for bidirectional communication (see [RFC6202] for reference). It is also designed to work over HTTP ports 80 and 443 as well as to support HTTP proxies and intermediaries. However, the design does not limit WebSocket to HTTP, and future implementations could use a simpler handshake over a dedicated port without reinventing the entire protocol. [RFC6455, p. 4-5]

Generally, the life cycle of a non-TLS secure WebSocket connection consists of three parts, an opening handshake, data transfer and a closing handshake. [RFC6455, p. 5]

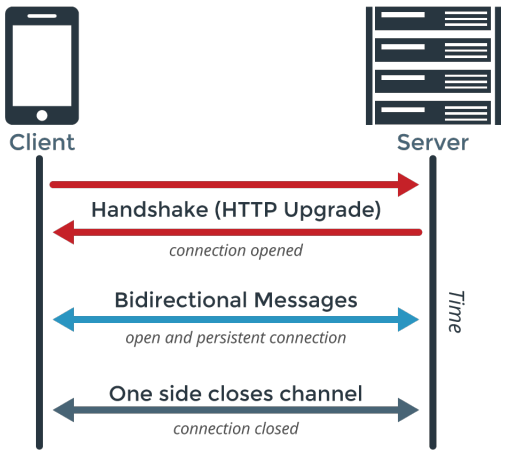


Figure 3.14: WebSocket communication [Wik22h]

The handshake from the client looks as follows:

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

Listing 1: WebSocket client handshake [RFC6455, p. 5]

The handshake from the server looks as follows:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

Listing 2: WebSocket server handshake [RFC6455, p. 7]

After the handshake is completed successfully, the connection can be used to exchange data bidirectionally. The closing handshake is far simpler than the opening handshake. One peer can send a control frame with data containing a specified control sequence to begin the closing handshake. Upon receiving such a frame, the other peer sends a close frame in response if it has not already sent one. Upon receiving that control frame, the first peer then closes the connection, safe knowing that no further data is forthcoming. It is also essential to know that it is safe for both peers to initiate this handshake simultaneously. [RFC6455, p. 9]

In the WebSocket protocol, data is transmitted through a sequence of frames. A high-level overview is shown in the image 3.15 below. [RFC6455, p. 27]

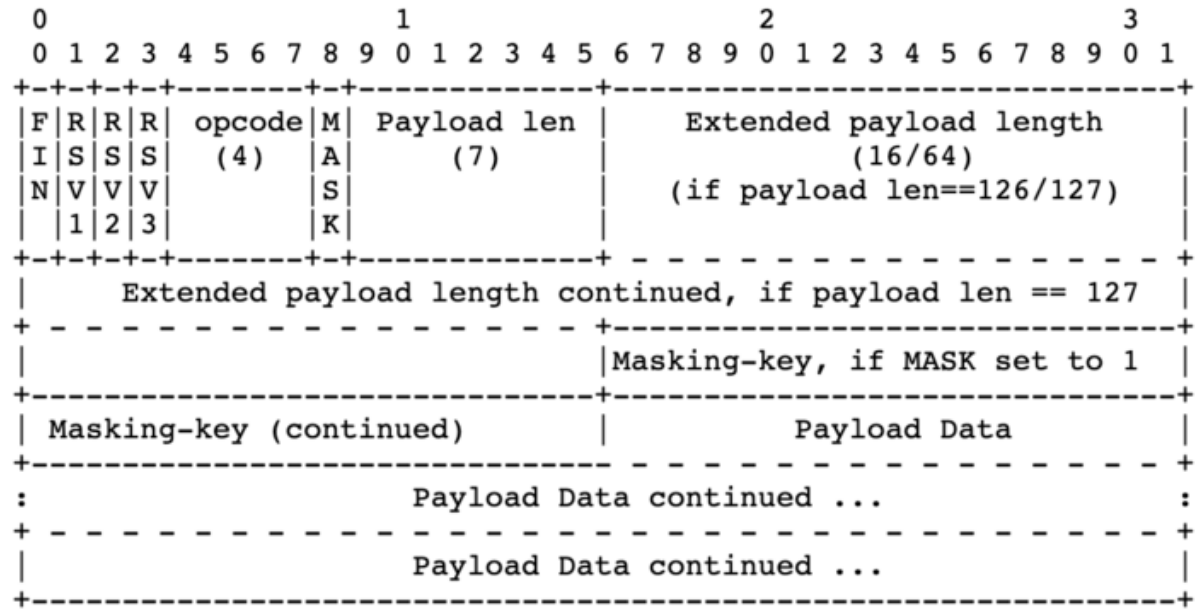


Figure 3.15: WebSocket frame [RFC6455, p. 27]

In the following, all frame fields are explained in more detail. [RFC6455, p. 27 f.]

- *FIN (1 bit)*: Indicates that this is the final fragment in a message.
- *RSV1, RSV2, RSV3 (1 bit)*: MUST be 0 unless an extension is negotiated that defines meanings for non-zero values. If a non-zero value is received and none of the negotiated extensions defines the meaning of such a non-zero value, the receiving endpoint must fail the connection.
- *Opcode (4 bits)*: Defines the interpretation of the payload data. If an unknown opcode is received, the receiving endpoint must fail the connection.
- *Mask (1 bit)*: Defines whether the payload data is masked. If set to 1, a masking key is present in the masking key, which is used to unmask the payload data. All frames sent from client to server have this bit set to 1.
- *Payload length (7 bits, 7+16 bits, 7+64 bits)*: The payload data length in bytes.
- *Masking-key*: All frames sent from the client to the server are masked by a 32-bit value that is contained within the frame. This field is present if the mask bit is set to 1 and is absent if the mask bit is set to 0.
- *Payload data (x+y bytes)*: The payload data is defined as extension data concatenated with application data.
- *Extension data (x bytes)*: The extension data is 0 bytes unless an extension has been negotiated. Any extension must specify the length of the extension data, how that length may be calculated, and how the extension use must be negotiated during the opening handshake. If present, the extension data is included in the total payload length.
- *Application data (y bytes)*: Arbitrary application data, taking up the remainder of the frame after extension data. The length of the application data is equal to the payload length minus the length of the extension data.

3.4 OPC-UA

OPC-Unified Architecture (OPC-UA) is not an integrated part of this thesis, but it is still important to mention. OPC-UA is a cross-platform, open-source, IEC62541 standard for data exchange from sensors to cloud applications developed by the OPC Foundation. It is important to note that this is not just a protocol but a collection of specifications for machine-to-machine communication. [SB13, p. 2]. It is used very frequently in the industry. For example, Siemens complements its PROFINET with OPC-UA [AG18].

OPC UA is based on an extensible framework with a multilayered architecture to accomplish the following design goals. [Fou22b]

1. *Functional equivalence*: Built on the OPC Classic specification.
2. *Platform independence*: Supports embedded microcontrollers and cloud-based deployments.
3. *Security*: Supports encryption, authentication, and auditing.
4. *Extensible*: Supports the ability to add new features without affecting existing applications.
5. *Comprehensive information modelling*: Supports the definition of complex information.

3.4.1 OPC UA Specification

OPC-UA has a series of specifications, also known as IEC 62541 standards. They are split into several smaller specifications illustrated in figure 3.16.

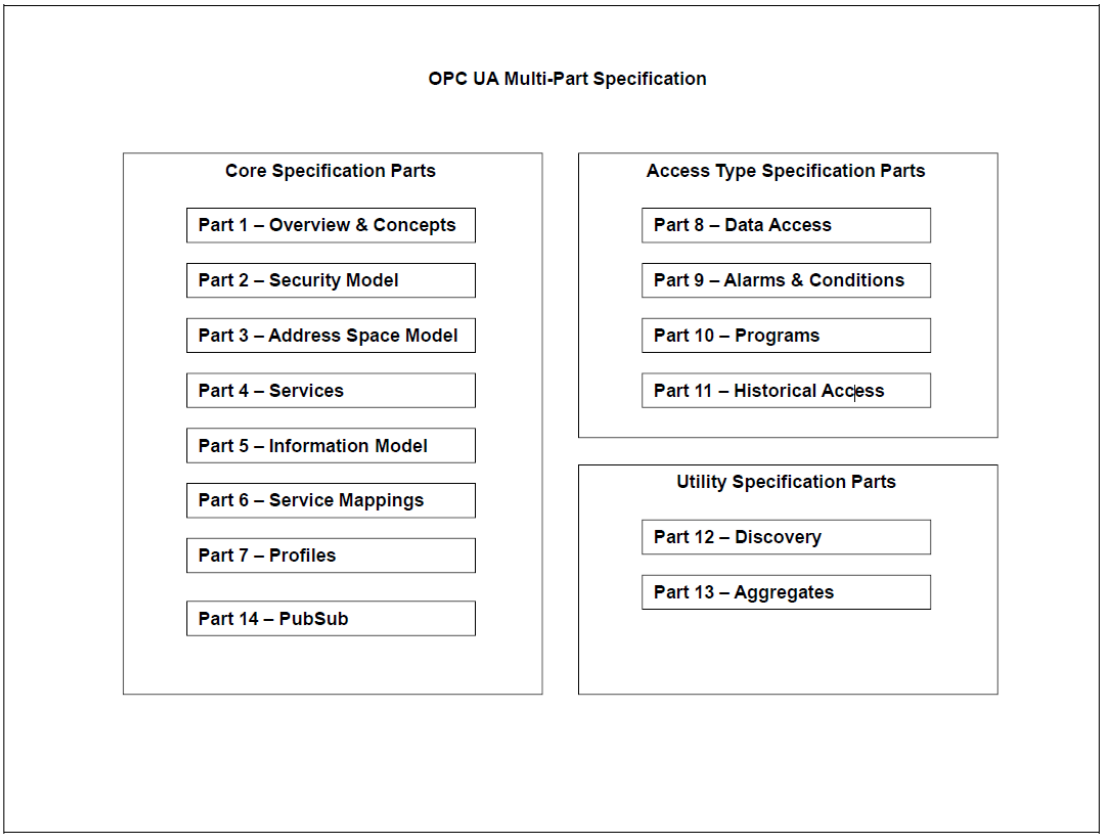


Figure 3.16: OPC UA specifications [Fou17, p. 7]

In the following, the responsibilities of the individual specifications are briefly and succinctly presented. [Fou17, p. 7f.]

1. *Overview and Concepts*: presents the concepts and overview of OPC UA.
2. *Security Model*: describes the model for securing interactions between OPC UA Applications.
3. *Address Space Model*: describes the contents and structure of the servers AddressSpaces.
4. *Services*: specifies the services provided by servers.
5. *Information Model*: specifies the types and their relationships defined for servers.
6. *Mappings*: specifies the mappings to transport protocols and data encodings supported by OPC UA.
7. *Profiles*: specifies the Profiles available for OPC UA Applications. These profiles provide groupings of functionality that can be used for conformance level certification. OPC UA Applications will be tested against the profiles.
8. *Data Acess*: specifies the use of OPC UA for data access.
9. *Alarms and Conditions*: specifies the use of OPC UA support for access to Alarms and Conditions. The base system includes support for simple Events; this specification extends that support to include support for Alarms and Conditions.
10. *Programs*: specifies OPC UA support for access to Programs.
11. *Historical Access*: specifies the use of OPC UA for historical access. This access includes both historical data and historical events.
12. *Discovery*: specifies how Discovery Servers operate in different scenarios and describes how UA Clients and Servers should interact with them. It also defines how UA related information should be accessed using standard directory service protocols such as LDAP.
13. *Aggregates*: specifies how to compute and return aggregates like minimum, maximum or average. Aggregates can be used with current and historical data.
14. *PubSub*: specifies the OPC Unified Architecture PubSub communication model. The PubSub communication model defines an OPC UA publish-subscribe pattern in addition to the Client-Server pattern defined by Services.

3.4.2 Architecture of OPC UA

The OPC-UA system is based on a client/server model. Each system may contain several servers and clients. Each client can communicate with one or more servers, and each server can communicate with one or more clients. [Fou17, p. 13]

The following figure shows the architecture with a combined server and client. [Fou17, p. 13]

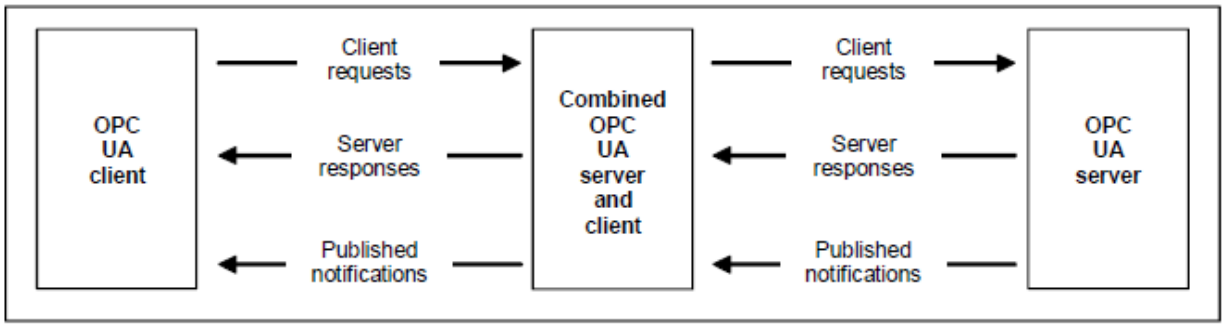


Figure 3.17: OPC UA system architecture [Fou17, p. 13]

OPC UA Clients

The OPC-UA architecture models the Client endpoint of client/server interactions. The following figure 3.18 illustrates the major elements of that. [Fou17, p. 13]

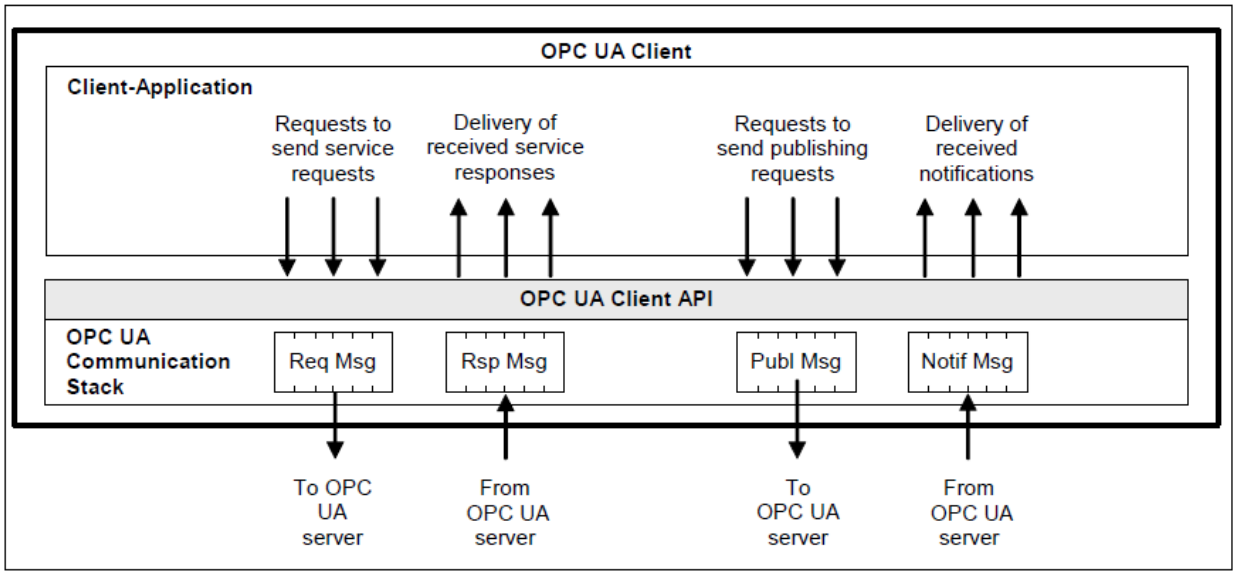


Figure 3.18: OPC UA client architecture [Fou17, p. 13]

The OPC UA Client includes OPC UA Client Structure OPC UA Client. Application, OPC UA Client Communication Stack and OPC UA Client API to be in contact with the server. First, the communication Stack converts the API calls into messages. Then, the Stack obtains responses and notifications messages from the server and passes messages to the client application through the Client API. [Fou17, p. 13]

OPC UA Servers

The OPC-UA Server architecture models the Server endpoint of client/server interactions. The following figure 3.19 illustrates the major elements of that. [Fou17, p. 14]

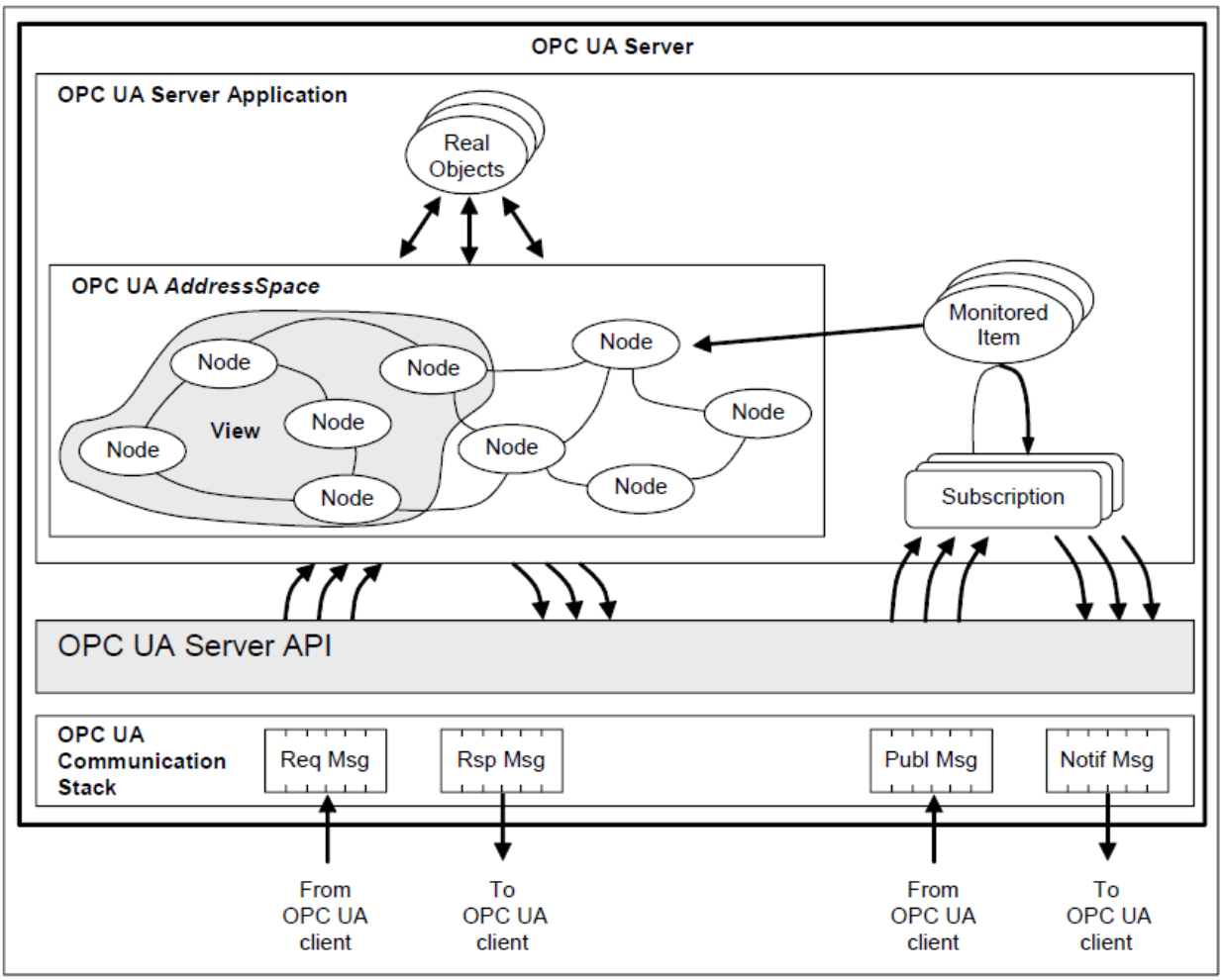


Figure 3.19: OPC UA server architecture [Fou17, p. 14]

The OPC UA Server includes OPC UA Server Application, real objects, OPC UA Address Space, OPC UA Server Stage API and the Communication Stack. The Address Space is composed of nodes and represents real objects. Through OPC Interfaces, the clients can access the nodes and services.

Chapter 4

Prototype specification

In order to evaluate the gateway design, a concrete prototype is needed. This prototype is adapted to the specific requirements to be solved with the proposed gateway.

4.1 Preconditions

Since we need special hardware and software to evaluate and develop the gateway, the following chapter will cover all the prerequisite points.

4.1.1 Hardware-Environment

The comparative study used several tools and platforms to evaluate IoT gateways, with the Raspberry PI being used the most. Therefore, the Raspberry PI is also used here. The concrete specifications can be found in the appendix B on page 152.

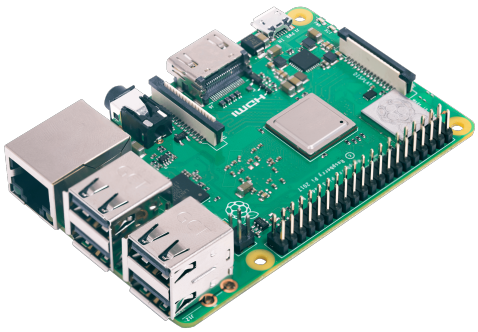


Figure 4.1: Raspberry Pi 3B+ [Ama21]

However, since the Raspberry PI must also be able to communicate via ZigBee, additional hardware is required, and the Digi Xbee 3 ZigBee Mesh Kit will be used for this purpose. This kit contains 3 Digi XBee SMT modules which support the ZigBee standard. The concrete specifications can be found in the appendix A on page 149.

Figure 4.2: Digi Xbee 3 ZigBee Mesk Kit [Inc21a]

4.1.2 Software-Environment

In addition to the hardware used, an operating system must be added to the Raspberry Pi. Here, the Raspberry Pi OS (32-bit) in version 5.10 is used. At the moment, this is the latest version of the Raspberry Pi OS. The critical point is that although the Raspberry Pi has a 64-bit CPU, a 32-bit operating system will be installed. This is because the Raspberry Pi OS version for 64-bit was still in beta at this decision. To configure the Digi Xbee 3 RF modules, the provided software XCTU from Digi International Inc. has to be used. XCTU is a freely available software to allow developers to configure and test Xbee RF modules via a graphical user interface. [Inc21b]

Figure 4.3: XCTU configuration tool [Inc21b]

Finally, a cloud provider is needed as the end consumer of the prototype. Amazon Web Services (AWS) is to be used here, as it is already being used for a project at Bremen University of Applied Sciences. To be more precise, the AWS IoT Core Service is to be used.

Figure 4.4: Amazon Web Services [Ser22]

Figure 4.5: AWS IoT Core [Ser22]

4.2 Requirement Analysis

In order to define the key actors and their interactions, a requirements analysis is performed according to the lecture notes of Prof. Dr.-Ing. Jasminka Matevska. Based on the actors, several use cases are defined, which provides the basis for the functional requirements. [Mat20]

4.2.1 Actors

To determine the requirements for the prototype, it is first necessary to identify the actors in such a system and identify their interactions. The following actors were identified based on the following questions:

- Which user groups does the system support to perform the required work?
- What user groups are required to perform the system's primary functions?
- What external hardware or software system will interact with the prototype?

The following three actors were identified that could play a role in such a system.

- Gateway
- Client
- Administrator

4.2.2 Use cases

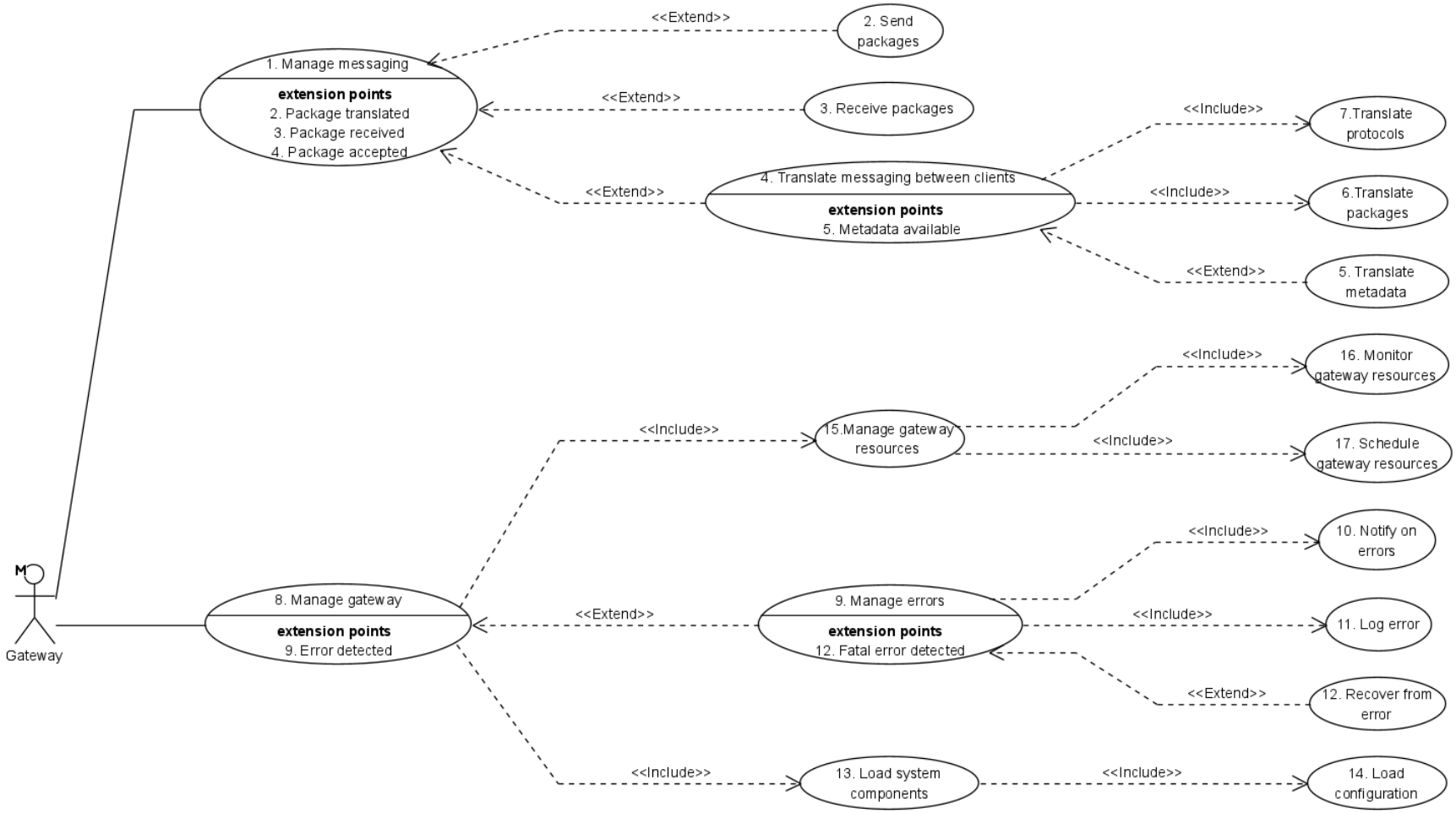


Figure 4.6: Use case diagram part one

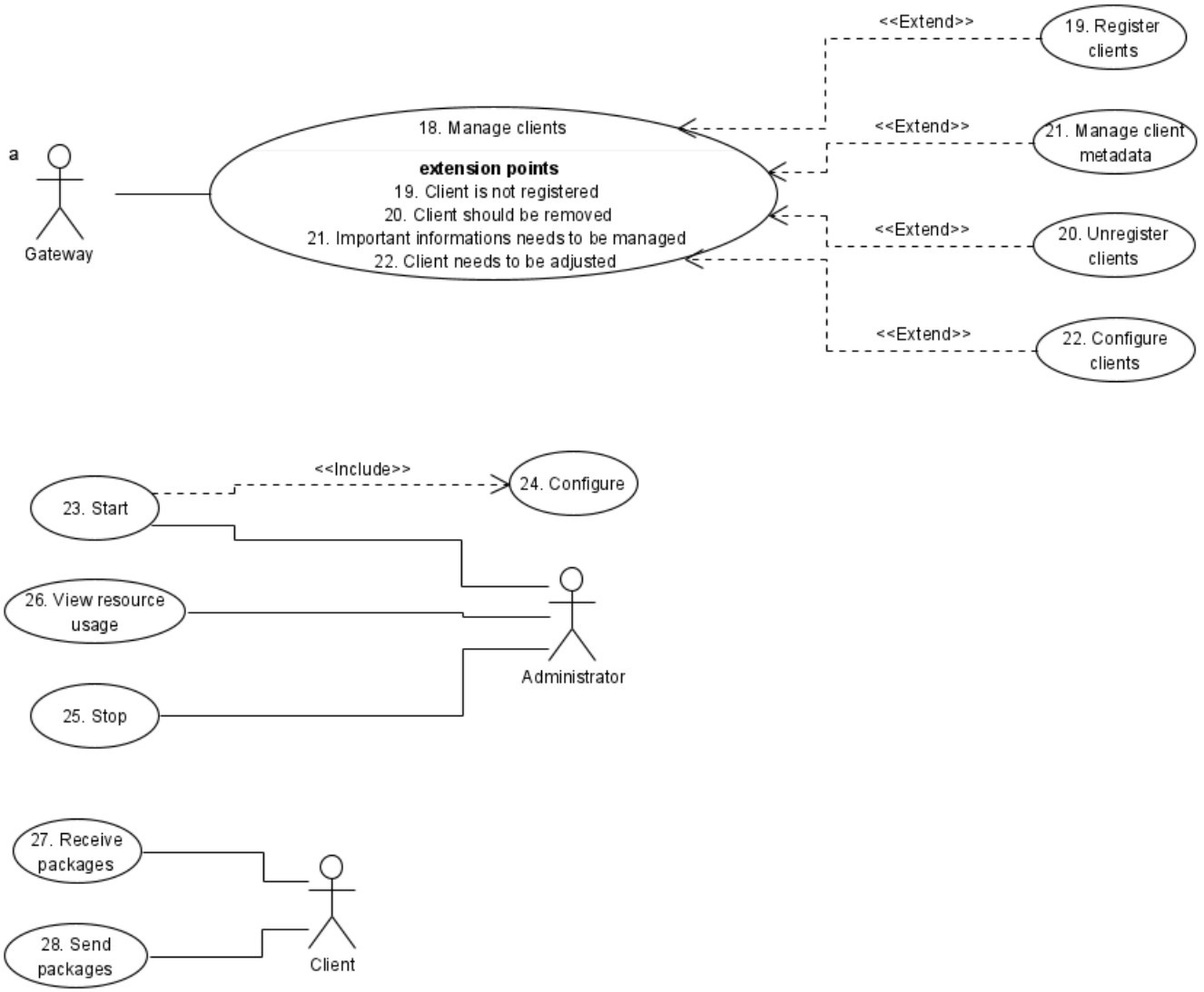


Figure 4.7: Use case diagram part two

Use case	1
Name	Manage messaging
Initiating actor	Gateway
Other actors	Client
Brief description	The gateway manages the exchange of messages between its registered clients.
Precondition	A package for exchange is available.
Postcondition	The data packet was successfully forwarded.
Procedure	The gateway processes the package by accepting it through the use case "3. Receive packet" and transform it through the use case "5. Translate messaging between clients". The transformed message can then be sent using the "2. Send packet" use case.
Alternatives	If the packet gets rejected by use case "3. Receive packet", processing of the packet is stopped.
Exceptions	In case of error, the use case "9. Manage errors" will be used, and the packet gets rejected.
Used use cases	2. Send packages 3. Receive packages 4. Translate messaging between clients 9. Manage errors

Table 4.1: Use case *1. Manage messaging*

Use case	2
Name	Send packages
Initiating actor	Gateway
Other actors	Client
Brief description	The package got forwarded to a client.
Precondition	The package got translated.
Postcondition	The package was delivered successfully.
Procedure	When a packet is translated and ready to be sent, it gets forwarded to a client.
Alternatives	-
Exceptions	In case of error, the use case "9. Manage errors" will be used, and the packet will not be forwarded to a client.
Used use cases	9. Manage errors

Table 4.2: Use case *2. Send packages*

Use case	3
Name	Receive packages
Initiating actor	Gateway
Other actors	Client
Brief description	One package has been received and is being prepared for further processing.
Precondition	A package has arrived.
Postcondition	The package was passed to internal processing.
Procedure	A package arrived and is being prepared for further processing through the use case “4. Translate messaging between clients”.
Alternatives	The package is rejected due to various properties.
Exceptions	In case of error, the use case “9. Manage errors” is used, and the packet gets rejected.
Used use cases	9. Manage errors

Table 4.3: Use case *3. Receive packages*

Use case	4
Name	Translate messaging between clients
Initiating actor	Gateway
Other actors	-
Brief description	All necessary information is processed to forward a packet.
Precondition	A package was accepted.
Postcondition	The package is translated to the protocol of the receiving client.
Procedure	The packet is adapted using the use cases “5. Translate metadata”, “6. Translate packages” and “7. Translate protocols” so that the gateway subsequently forwards it to a recipient.
Alternatives	-
Exceptions	In case of error, the use case “9. Manage errors” is used, and the processing of the packet is stopped.
Used use cases	5. Translate metadata 6. Translate packages 7. Translate protocols 9. Manage errors

Table 4.4: Use case *4. Translate messaging between clients*

Use case	5
Name	Translate metadata
Initiating actor	Gateway
Other actors	-
Brief description	Translates necessary metadata about the messages to be exchanged by the clients.
Precondition	1. A package was accepted. 2. Metadata is needed to exchange the package.
Postcondition	Metadata was adapted to the receiver protocol.
Procedure	Some protocols need metadata like MAC addresses or Secure Sockets Layer (SSL) certificates to exchange data. For this purpose, necessary information is appended to the packet to exchange via the receiver protocol.
Alternatives	-
Exceptions	In case of error, the use case "9. Manage errors" is used, and the processing of the packet is stopped.
Used use cases	9. Manage errors

Table 4.5: Use case *5. Translate metadata*

Use case	6
Name	Translate packages
Initiating actor	Gateway
Other actors	-
Brief description	The incoming package is converted into a format for the receiver protocol.
Precondition	A package was accepted.
Postcondition	The package was adapted to the receiver protocol.
Procedure	The incoming packet is converted into a format for the receiver protocol. Necessary formatting can be from, e.g., a byte array to JavaScript Object Notation (JSON).
Alternatives	If the packet already has the necessary format, it is simply passed on.
Exceptions	In case of error, the use case "9. Manage errors" is used, and the processing of the packet is stopped.
Used use cases	9. Manage errors

Table 4.6: Use case *6. Translate packages*

Use case	7
Name	Translate protocols
Initiating actor	Gateway
Other actors	-
Brief description	The protocol of the package will be adjusted to the protocol of the receiver.
Precondition	The package was accepted.
Postcondition	The protocol of the package was adapted to the receiver protocol.
Procedure	The protocol of the package will be adjusted to the protocol of the receiver. This includes, e.g., the correct appending of calls. With ZigBee, the devices must be addressed by the specific addresses, with MQTT specific endpoints are addressed, and with WebSocket, the correct connection must be selected.
Alternatives	-
Exceptions	In case of error, the use case "9. Manage errors" is used, and the processing of the packet is stopped.
Used use cases	9. Notify on errors

Table 4.7: Use case *7. Translate protocols*

Use case	8
Name	Manage gateway
Initiating actor	Gateway
Other actors	-
Brief description	The software manages the gateway.
Precondition	-
Postcondition	-
Procedure	The software manages the gateway. For this purpose, the computing resources are managed using the use case "10. Manage gateway resources" and the use case detects errors "9. Notify on errors" to notify the administrator.
Alternatives	-
Exceptions	If errors occur in device management, the gateway may be faulty. For this reason, the use case "9. Manage errors" should be used to contact an administrator. In addition, the gateway should attempt to rectify the faulty state itself by restarting. Faulty components should not be reloaded if possible.
Used use cases	9. Manage errors 10. Manage gateway resources

Table 4.8: Use case *8. Manage gateway*

Use case	9
Name	Manage errors
Initiating actor	Gateway
Other actors	Administrator
Brief description	The gateway handles errors automatically to avoid faulty states.
Precondition	An error was detected.
Postcondition	The gateway remains functional.
Procedure	When an error is detected, an administrator is notified via the use case "10. Notify on errors" and this error is written to a file with "11. Log error". If it is a fatal error, the gateway will try to recover from the error by using the use case "12. Recover from error".
Alternatives	-
Exceptions	-
Used use cases	10. Notify on errors 11. Log errors 12. Recover from error

Table 4.9: Use case *9. Manage errors*

Use case	10
Name	Notify on errors
Initiating actor	Gateway
Other actors	Administrator
Brief description	The software contacts an administrator in case of any error.
Precondition	An error was detected.
Postcondition	The administrator was notified.
Procedure	The administrator is notified by e-mail that an error has occurred in the gateway.
Alternatives	-
Exceptions	In case of any error, while notifying the administrator, the software retries the operation one more time.

Table 4.10: Use case *10. Notify on errors*

Use case	11
Name	Log error
Initiating actor	Gateway
Other actors	-
Brief description	The software logs all errors to a specific file.
Precondition	An error was detected.
Postcondition	The error was logged to a file.
Procedure	The error is logged to a file with a timestamp to be easily found in it by the administrator.

Table 4.11: Use case *11. Log error*

Use case	12
Name	Recover from error
Initiating actor	Gateway
Other actors	-
Brief description	The software tries to recover from the detected error.
Precondition	A fatal error was detected.
Postcondition	The gateway remains functional.
Procedure	The software tries to bring the system into a correct state via a restart. Then, if necessary, it tries to find the defective part to start the system without it.
Alternatives	-
Exceptions	While the software is in the recovery process, any error will lead to a system failure that can not be resolved.

Table 4.12: Use case *12. Recover from error*

Use case	13
Name	Load system components
Initiating actor	Gateway
Other actors	-
Brief description	The software loads additional system components.
Precondition	System components are defined in a configuration file.
Postcondition	The system is ready for operation.
Procedure	After the system configurations have been loaded with the use case "14. load configuration", the defined components are loaded and plugged together.
Alternatives	-
Exceptions	Suppose errors occur while loading system components. The gateway is in an unresolvable error state. Therefore the start of the system will be aborted.
Used use cases	14. Load configuration

Table 4.13: Use case *13. Load system components*

Use case	14
Name	Load configuration
Initiating actor	Gateway
Other actors	-
Brief description	The system loads the configuration file.
Precondition	The system was started or restarted.
Postcondition	The configurations were loaded and checked.
Procedure	The system loads the configuration from memory and checks the file's syntax.
Alternatives	-
Exceptions	If an error occurs while loading the configuration file, the system is in an unresolvable error state. Therefore the start of the system will be aborted.

Table 4.14: Use case *14. Load configuration*

Use case	15
Name	Manage gateway resources
Initiating actor	Gateway
Other actors	-
Brief description	The system manages the gateway, i.e., its resources and states.
Precondition	-
Postcondition	-
Procedure	The system manages the gateway, i.e., its resources and states. This is done by using the use cases "16. Monitor gateway resources" and "17. Schedule gateway resources".
Alternatives	-
Exceptions	When an error occurs, it is handled by the use case "9. Manage errors".
Used use cases	16. Monitor gateway resources 17. Schedule gateway resources 9. Manage errors

Table 4.15: Use case *15. Manage gateway resources*

Use case	16
Name	Monitor gateway resources
Initiating actor	Gateway
Other actors	-
Brief description	The system monitors the resource utilization of the gateway.
Precondition	-
Postcondition	The current resource consumption was saved as a data point.
Procedure	The system monitors resource utilisation like CPU, memory, or network utilization. The utilization is written to a file to observe history.
Alternatives	-
Exceptions	When an error occurs, it is handled by the use case "9. Manage errors".
Used use cases	9. Manage errors

Table 4.16: Use case *16. Monitor resources*

Use case	17
Name	Schedule gateway resources
Initiating actor	Gateway
Other actors	-
Brief description	The gateway resources are automatically distributed according to the load.
Precondition	-
Postcondition	The gateway resources were fairly distributed among the running processes.
Procedure	The gateway resources are automatically distributed according to the load. This means, for example, that high-priority packets have priority, and high-load protocols will receive more memory to buffer packets if necessary.
Alternatives	-
Exceptions	When an error occurs, it is handled by the use case "9. Manage errors".
Used use cases	9. Manage errors

Table 4.17: Use case *17. Schedule resources*

Use case	18
Name	Manage clients
Initiating actor	Gateway
Other actors	Client
Brief description	The system manages the connected clients.
Precondition	Clients are connected to the system.
Postcondition	Client connections are correctly working.
Procedure	The system manages all connected clients through the use cases "19. Register clients", "20. Unregister clients", "21. Manage client metadata", and "22. Configure clients".
Alternatives	-
Exceptions	When an error occurs, it is handled by the use case "9. Manage errors".
Used use cases	19. Register clients 20. Unregister clients 21. Manage client metadata 22. Configure clients 9. Manage errors

Table 4.18: Use case *18. Manage clients*

Use case	19
Name	Register clients
Initiating actor	Gateway
Other actors	-
Brief description	Clients are automatically registered on first contact.
Precondition	The client is not registered.
Postcondition	The client is registered.
Procedure	Clients are automatically registered on first contact. This includes registration in the gateway and registration in the connected consumer (AWS, for example).
Alternatives	-
Exceptions	When an error occurs, it is handled by the use case "9. Manage errors".
Used use cases	9. Manage errors

Table 4.19: Use case *19. Register clients*

Use case	20
Name	Unregister clients
Initiating actor	Gateway
Other actors	Client
Brief description	Clients are automatically unregistered on request.
Precondition	The client is registered.
Postcondition	The client is unregistered.
Procedure	Clients are automatically unregistered on request. This includes the deleting process in the gateway and the connected consumer (AWS, for example).
Alternatives	-
Exceptions	When an error occurs, it is handled by the use case "9. Manage errors".
Used use cases	9. Manage errors

Table 4.20: Use case *20. Unregister clients*

Use case	21
Name	Manage client metadata
Initiating actor	Gateway
Other actors	-
Brief description	The system manages client metadata.
Precondition	The client is registered.
Postcondition	Client metadata is correctly saved in a shared memory.
Procedure	The system manages client metadata to establish connections between different protocols and clients. This may include MAC addresses, device names, or SSL certificates. All required data is written to shared memory.
Alternatives	-
Exceptions	When an error occurs, it is handled by the use case "9. Manage errors".
Used use cases	9. Manage errors

Table 4.21: Use case *21. Manage client metadata*

Use case	22
Name	Configure clients
Initiating actor	Gateway
Other actors	Client
Brief description	Clients can be automatically configured via the gateway.
Precondition	Clients are not configured or need to be reconfigured.
Postcondition	Clients are correctly configured according to the gateway restrictions like resource utilisation.
Procedure	Clients can be configured via the gateway. This should be regarded as provisional for the time being since complex configurations should not be carried out. It is first about configurations like, e.g., the sample rate of sensors.
Alternatives	-
Exceptions	When an error occurs, it is handled by the use case "9. Manage errors".
Used use cases	9. Manage errors

Table 4.22: Use case *22. Configure clients*

Use case	23
Name	Start gateway
Initiating actor	Administrator
Other actors	Gateway
Brief description	The administrator starts the system.
Precondition	The system is currently not running.
Postcondition	The system is currently running.
Procedure	The administrator starts the system via the command line. The administrator must have previously configured the system with a configuration file. For example, see the use case "24. Configure".
Alternatives	-
Exceptions	Errors when starting the system create an unrecoverable error state. The error is handled by the use case "9. Manage errors".
Used use cases	-
	24. Configure
	9. Manage errors

Table 4.23: Use case *23. Start gateway*

Use case	24
Name	Configure
Initiating actor	Administrator
Other actors	Gateway
Brief description	The administrator configures the gateway.
Precondition	The system is not running and is not configured yet.
Postcondition	The system is correctly configured.
Procedure	The administrator configures the gateway. For this purpose, there is a configuration file, which allows the configuration of the active interfaces and their connections.

Table 4.24: Use case *24. Configure*

Use case	25
Name	Stop gateway
Initiating actor	Administrator
Other actors	Gateway
Brief description	The administrator stops the system.
Precondition	The system is running.
Postcondition	The system is not running.
Procedure	The administrator stops the system via the command line.
Alternatives	-
Exceptions	Errors when stopping the system create an unrecoverable error state. The error is handled by the use case “9. Manage errors”.
Used use cases	-

9. Manage errors

Table 4.25: Use case *25. Stop gateway*

Use case	26
Name	View resources usage
Initiating actor	Administrator
Other actors	Gateway
Brief description	The administrator fetches an endpoint to observe the resource utilization of the gateway.
Postcondition	The administrator has received an overview of resource utilization.
Procedure	The administrator fetches an endpoint to observe the resource utilization of the gateway.
Alternatives	-
Exceptions	When an error occurs, it is handled by the use case “9. Manage errors”.
Used use cases	

9. Manage errors

Table 4.26: Use case *26. View resource usage*

Use case	27
Name	Receive packages
Initiating actor	Client
Other actors	Gateway
Brief description	The client receives packages from the gateway.
Precondition	The client is connected to the gateway.
Postcondition	The client received a package from the gateway.
Procedure	The client receives a package from the gateway and logs it into a file.
Alternatives	-
Exceptions	Errors are also logged in a file and can lead to program failure.

Table 4.27: Use case *27. Receive packages*

Use case	28
Name	Send packages
Initiating actor	Client
Other actors	Gateway
Brief description	The client sends packages to the gateway.
Precondition	The client is connected to the gateway.
Postcondition	A package was sent to the gateway.
Procedure	The client sends packages to the gateway via a specific protocol. All outgoing packages are logged to a file.
Alternatives	-
Exceptions	Errors are also logged in a file and can lead to program failure.

Table 4.28: Use case *28. Send packages*

4.2.3 Functional requirements

The functional requirements were created from the use cases and the introduction of the prototype, and they refer to the entire architecture. The letters in front of the identification number indicate which part of the application the requirement refers to.

- *A*: Gateway
- *B*: Client

As a verification type of the individual requirements, the following was taken from the lecture script of Prof. Dr. -Ing Matevska. [Mat20]

- *Test (T)*: Use test scenarios or test cases/input data to test expected behaviour or output data.
- *Inspection (I)*: Verification of correctness primarily by inspection of source code or configuration files. (e.g., whether the IP addresses of the default gateways are configured correctly).
- *Review of Design (RoD)*: Review of the relevant design documentation (e.g. if a specific design pattern should be used to achieve a distribution or a specific behaviour).

In addition to this, the MoSCoW method was used to prioritize the requirements. The term MoSCoW itself is an acronym composed of the first letter of each of the four prioritization categories. [Wik22d]

- *M*: Must have
- *S*: Should have
- *C*: Could have
- *W*: Would not like to have

ID	Requirement description	Use case	Verification type	Priority
A1	The system shall be configurable	13, 14, 24	I	M
A1.1	The defined protocol adapters shall be configurable	24	I	M
A1.2	The connections between the protocol adapters shall be configurable	24	I	M
A1.3	The notification mail address shall be configurable	24	I	M
A2	The system shall load configurations from a relative path	14	T	M
A3	The system shall load protocol adapters as an additional component	13	RoD	S
A3.1	The system shall be able to start protocol adapters	13	RoD	C
A3.2	The system shall be able to stop protocol adapters	13	RoD	S
A3.3	The system shall configure protocol adapters	24	I	S
A4	The system shall be managed over the command line	23, 24, 25	RoD	M
A4.1	The system shall be started over the command line	23	RoD	M

Table 4.29 - Continued from previous page

ID	Requirement description	Use case	Verification type	Priority
A4.2	The system shall be stopped over the command line	25	RoD	M
A4.3	The system shall be able to install a protocol adapter over the command line	24	RoD	M
A4.4	The system shall be able to start a protocol adapter over the command line	23	RoD	M
A4.5	The system shall be able to stop a protocol adapter over the command line	25	RoD	M
A5	The system shall save metadata	5, 21	T	M
A5.1	The system shall save the client name	21	T	M
A5.2	The system shall save the used protocol of the client	5, 21	T	M
A5.3	The system shall save specific protocol metadata if necessary	5	T	M
A6	The system shall monitor the gateway computing resources	15, 16	T	S
A6.1	The system shall monitor the CPU utilization	16	T	S
A6.2	The system shall monitor the memory utilization	16	T	S
A6.3	The system shall monitor the protocol workload (packages/s)	16	T	S
A6.4	The system shall monitor the bandwidth	16	T	C
A6.5	The system shall monitor the latency between clients	16	T	C
A7	The system shall use the monitored resources to modify the behaviour of current processes	15, 17	T	S
A7.1	The system shall be able to prioritize packets	17	T	S
A7.2	The system shall be able to deny packages	17	T	S
A8	The system shall manage client connections	18, 19, 20, 21, 22	T	S
A8.1	The system shall identify not registered clients	19	T	S
A8.1	The system shall be able to register clients in available adapters and their consumers	19	T	S
A8.2	The system shall be able to make QoS changes for clients	22	T	C
A9	The system shall be able to handle exceptions	9, 10, 11, 12	T	S
A9.1	The system shall notify the administrator about exceptions	10	T	S
A9.2	The system shall log exceptions	11	T	S
A9.3	The system shall be able to shutdown on fatal exceptions	12	T	S
A9.4	The system shall be able to detect faulty system components	12	T	C
A9.4	The system shall be able to deactivate faulty system components	12	T	C
A10	The system shall provide an HTTP endpoint for resource consumption that can be requested	26	RoD	S
A10.1	The system shall show the defined resource consumption defined in A6	26	T	S
B1	The overall system shall provide an MQTT adapter	1	RoD	M

Table 4.29 - Continued from previous page

ID	Requirement description	Use case	Verification type	Priority
B1.1	The MQTT adapter shall be able to receive data	3	T	M
B1.2	The MQTT adapter shall be able to send data	2	T	M
A1.3	The MQTT adapter shall be able to translate data packets from MQTT into a local format	6	T	M
A1.4	The MQTT adapter shall be able to translate local data packets to MQTT	6	T	M
B2	The overall system shall provide an WebSocket adapter	1	RoD	M
B2.1	The WebSocket adapter shall be able to receive data	3	T	M
B2.2	The WebSocket adapter shall be able to send data	2	T	M
A2.3	The WebSocket adapter shall be able to translate data packets from WebSocket into a local format	6	T	M
A2.4	The WebSocket adapter shall be able to translate local data packets to WebSocket	6	T	M
B3	The overall system shall provide an ZigBee adapter	1	RoD	M
B3.1	The ZigBee adapter shall be able to receive data	3	T	M
B3.2	The ZigBee adapter shall be able to send data	2	T	M
A3.3	The ZigBee adapter shall be able to translate data packets from ZigBee into a local format	6	T	M
A3.4	The ZigBee adapter shall be able to translate local data packets to ZigBee	6	T	M
B4	The overall system shall provide an AWS-MQTT adapter	1	RoD	M
B4.1	The AWS-MQTT adapter shall be able to receive data	3	T	M
B4.2	The AWS-MQTT adapter shall be able to send data	2	T	M
B4.3	The AWS-MQTT adapter shall be able to translate data packets from MQTT into a local format	6	T	M
B4.4	The AWS-MQTT adapter shall be able to translate local data packets to MQTT	6	T	M
B4.5	The AWS-MQTT adapter shall be able to register clients in AWS	19	T	M
C1	Clients shall be able to receive packages	27	T	M
C1.1	Clients shall save received packages in a human readable format	27	T	M
C1.2	A simple MQTT-Client shall be able to receive packages	27	T	M
C1.3	A simple ZigBee-Client shall be able to receive packages	27	T	M
C1.4	A simple WebSocket-Client shall be able to receive packages	27	T	M

Table 4.29 - Continued from previous page

ID	Requirement description	Use case	Verification type	Priority
C1.5	Amazon Web Service IoT Core shall be able to receive packages	27	RoD	M
C2	Clients shall be able to send packages	28	T	M
C2.1	Clients shall save sent packages in a human readable format	28	T	M
C2.2	A simple MQTT-Client shall be able to send packages	28	T	M
C2.3	A simple ZigBee-Client shall be able to send packages	28	T	M
C2.4	A simple WebSocket-Client shall be able to send packages	28	T	M

Table 4.29: Functional requirements

4.2.4 Interface requirements

Since, in this case, most interface requirements are also Functional Requirements, there are no additional points to mention here. However, in addition to the interfaces mentioned in the functional requirements, modularity is now required. The interfaces should be designed so modular that they can be used without problems by not specified protocols.

4.2.5 Quality requirements

Since the defined system monitors the resources and allocates them automatically, performance measurements are to be made. These measurements should show how performant the gateway handles increasing message load. However, there are no concrete requirements on how well the gateway should perform.

The values to be measured are listed and presented once below. They were selected based on the possible values and related work.

- Central processing unit (CPU) utilisation of the system
- CPU utilisation of the process
- Heap memory usage
- Non heap memory usage
- Throughput of messages
- Response time

The gateway should be designed to enable high availability. However, this has to be proven, which is not part of the thesis. Nevertheless, at least the mean time to failure (MTTF) and the mean time between failures (MTBF) should be measured. Unfortunately, the mean time to recovery (MTTR) cannot be measured since it is based on the recovery of service. Since we do not have a service team or a Service-level agreement (SLA) in the thesis, this value will not be further considered.

The system should be as compatible as possible with different hardware, as attempts are being made to solve the heterogeneity problem in the IoT world. For the prototypical application, however, the application must necessarily be able to run on Windowsx64 and Raspberry Pi OS.

4.2.6 Security requirements

The security of the gateway is not listed in the mandatory objectives. For this reason and the time dependency of the thesis, no special security requirements are placed on the prototype. Nevertheless, especially in the environment of IoT gateways, security is a fundamental issue. This is so accurate because so many different interfaces meet at one point. For this reason, section 2.1.6 on page 15 already discussed several security threats that need to be solved. Furthermore, it is essential that not only the listed gateway security risks are essential but also those from the other layers. This is mainly because a smart gateway offers much more functionality than a simple protocol conversion.

4.2.7 Operability requirements

Since the system is to be started via the command line, there are no further requirements. However, the required configuration parameters should all be loaded from a file so that no parameter input in the command line is necessary.

In the further course, the application should show the utilization of the system over an endpoint. Since this is a simple prototype, this can be easily transferred into text form without further effort.

4.3 Development environment

The following chapter deals with the software components used in developing the prototype, which was not a prerequisite. The specific components are explained more precisely to enable other developers to recreate the same environment.

4.3.1 Java

Choosing the programming language for such a problem is relatively tricky. On the one hand, it is only a prototype, and on the other hand, optimizations are to be made. Therefore, C++ would be particularly suitable for the optimizations, but Java was used due to several arguments:

- Java does not require cross-compilation, so it already runs on the required platforms without problems.
- Most systems use Open Services Gateway initiative (OSGi) to create modularity. OSGi is tied to the Java programming language in this case.
- Java was chosen because of the existing knowledge, which is also available in C++ but not profound enough to make any optimizations.

Specifically, the default-JDK version for the Raspberry Pi is to be used. At the current time, it is the OpenJDK11.

4.3.2 Apache Karaf (OSGi)

Apache Karaf is a modern polymorphic application container. It can be used as a standalone container, supporting various applications and technologies. It also supports the "run anywhere" concept because it is based on Java. Apache Karaf is powered by OSGi and uses either Apache Felix or Eclipse Equinox as the main OSGi implementation. Apache Karaf provides additional features on top of the OSGi standard. [Fou22c]

Figure 4.8: Apache Karaf [Fou22c]

- *Hot deployment*: Apache Karaf automatically detects new files in the deploy directory.
- *Complete console*: Apache Karaf provides a complete Unix-like console where it can completely manage the container.
- *Dynamic configuration*: Apache Karaf provides a set of commands focused on managing its configuration. All configuration files are centralized in the *etc* folder. Any change in a configuration file is noticed and reloaded.
- *Advanced logging system*: Apache Karaf supports all the popular logging frameworks (slf4j or log4j). Apache Karaf centralizes the configuration in one file, independent of the logging framework.
- *Provisioning*: Apache Karaf supports a variety of different ways to deploy applications. It also provides the concept of "Karaf Features", which is a way to describe the application.
- *Management*: Apache Karaf is an enterprise-ready container, providing many management indicators and operations via JMX.
- *Remote*: Apache Karaf embeds an Secure Shell Protocol (SSH) server to use the console remotely. In addition, the management layer is also accessible remotely.
- *Security*: Apache Karaf provides a complete security framework (based on JAAS) and provides a Role-Based Access Control (RBAC) mechanism for console and JMX access.
- *Instances*: multiple instances of Apache Karaf can be managed directly from the main instance (root).
- *OSGi frameworks*: Apache Karaf is not tightly coupled to one OSGi framework. For example, Apache Karaf runs with the Apache Felix Framework by default, but it can easily be switched to Equinox.

OSGi itself is a Java framework for developing and deploying modular software programs and libraries. Each bundle is a tightly coupled, dynamically loadable collection of classes, jars, and configuration files explicitly declaring their external dependencies. [Wik22e]

4.3.3 IntelliJ IDEA

IntelliJ IDEA is optional for setting up the project. It is just the Integrated development environment (IDE) used in the master thesis. However, any IDE that supports Maven as project management software can be used to integrate the project.

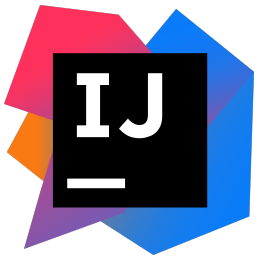


Figure 4.9: IntelliJ IDEA [Jet22]

In summary, IntelliJ IDEA is an intelligent, context-aware IDE for working with Java and other JVM languages like Kotlin, Scala, and Groovy. Additionally, IntelliJ supports many popular frameworks like Spring, Jakarta EE, Micronaut, Quarkus, and Helidon. Moreover, it can also be extended with free plugins developed by JetBrains. [Jet22]

4.3.4 Maven

Apache Maven is a tool for managing and understanding software projects. Based on a Project Object Model (POM) concept, Maven can manage the building, reporting, and documentation of a project from a centralized source of information. [Fou22d]

Figure 4.10: Apache Maven [Fou22d]

The primary goal of Apache Maven is to enable a developer to track the complete status of a development effort in the shortest possible time. For this reason, Maven addresses the following problem areas: [Fou22d]

- Making the build process simple
- Providing a unified build system
- With providing high-quality project information
- With promoting better development practices

4.3.5 XBee Java Library

Xbee Java Library is an easy-to-use API developed in Java to interact with Digi International's Xbee radio frequency modules. [Inc22a]

Generally, no libraries are listed here, as they are all in the Maven POM. However, this project is unique because, firstly, the hardware relies on it, and, secondly, the library had to be recompiled. The library used an ancient interface for serial communication, so it only ran on 32 bit. This interface was replaced so that 64 bit is also supported.

4.3.6 Eclipse Mosquitto

Eclipse Mosquitto is an open-source message broker that implements the MQTT protocol version 5.0, 3.1.1 and 3.1. Mosquitto is lightweight and is suitable for use on all devices, from low power single board computers to full servers. [Res22]

Figure 4.11: Eclipse Mosquitto [Res22]

Additionally Eclipse Paho provides a MQTT client library implementation in a wide variety of languages to enable the communication with MQTT brokers like Mosquitto. [Res22]

4.3.7 CPU-Z

CPU-Z is a freeware that gathers information on some components of a computer system. Additionally it can be used to stress the CPU with a Benchmark. This stress test is used to verify the behaviour of the Gateway on high CPU usage. [CPU22]

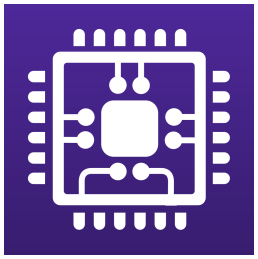


Figure 4.12: CPU-Z [CPU22]

Chapter 5

Design

The following chapter deals with the design of the prototype and is based on the requirements presented. It deals with very complex dependencies and is partly broken down in some places.

5.1 Architecture

First of all, the following figure 5.1 on page 69 will describe the overall system to show which components are part of the system and where they run or can be found. The following table 5.1 on the following page also shortly describes these components.

System	Description
Raspberry Pi (ZigBee)	The Raspberry Pi (ZigBee) is a test environment to simulate a single ZigBee client. Unfortunately, it is impossible to simulate multiple clients as only one serial connection to the XBee 3 is available.
Raspberry Pi (Gateway)	The Raspberry Pi (Gateway) provides an environment to run all necessary components for the gateway. These include all available adapters, the gateway, and Apache Karaf to provide them.
Raspberry Pi (MQTT broker)	The Raspberry Pi (MQTT broker) serves as an additional MQTT broker. It was decided not to install it on the gateway since it consumes additional resources and cannot be optimised due to third parties. However, the broker could also be installed directly on the gateway without any problems.
Testing Environment	The testing environment can be any device with enough processing power. Here, several clients are simulated to increase the gateway's load artificially.
AWS eu-central-1	The cloud server provided by Amazon Web Services.
Execution Environment	**Description**
JVM	The Java virtual machine running on a device to host java applications.
IoT Core	The AWS IoT Core system is the end consumer for this application.
Apache Karaf	The OSGi implementation used to deploy the gateway system parts.
Artifact	**Description**
xbee-client.jar	The xbee-client.jar is used to communicate with the gateway via ZigBee. In order to use it, an Xbee3 must be connected to the system via a USB connection.
xbee-configuration	The xbee-configuration is the configuration that is written to the flash memory of the Xbee3. This configuration specifies which network the device belongs to and which other communication parameters are used.

Table 5.1 continued from previous page

Artifact	Description
xbee-configuration-r	The xbee-configuration-r specifies that this XBee3 device should act as a router. This allows the device to communicate with all other connected devices without any problems.
xbee-configuration-c	The xbee-configuration-c specifies that this Xbee3 devices should act as a coordinator. So this device will create and manage the ZigBee network.
websocket-client.jar	The websocket-client.jar is used to communicate with the gateway via WebSocket.
mqtt-client.jar	The mqtt-client.jar is used to communicate with the gateway via MQTT. This client does not communicate directly with the gateway, but must first communicate with a broker, as this is specified in the MQTT standard.
Eclipse Mosquitto	Eclipse Mosquitto is the MQTT broker used in this case and enables the use of the MQTT-3.1, MQTT-3.1.1 and MQTT-5 specifications.
gateway.jar	The gateway.jar forms the core of the system and provides the actual gateway.
xbee-adapter.jar	The xbee-adapter.jar communicates with all available ZigBee clients and forwards the communicated data to the gateway. It is also able to receive data from the gateway and to send this to one of its clients.
mqtt-adapter.jar	The mqtt-adapter.jar communicates with all available MQTT clients and forwards the communicated data to the gateway. It is also able to receive data from the gateway and to send this to one of its clients.
websocket-adapter.jar	The websocket-adapter.jar communicates with all available WebSocket clients and forwards the communicated data to the gateway. It is also able to receive data from the gateway and to send this to one of its clients.
aws-mqtt-adapter.jar	The aws-mqtt-adapter.jar communicates with Amazon Web Services and forwards the communicated data to the gateway. It is also able to receive data from the gateway and to send this to one of its clients.
iot-core-configuration	The configuration of AWS IoT Core

Table 5.1: Deployment diagram description

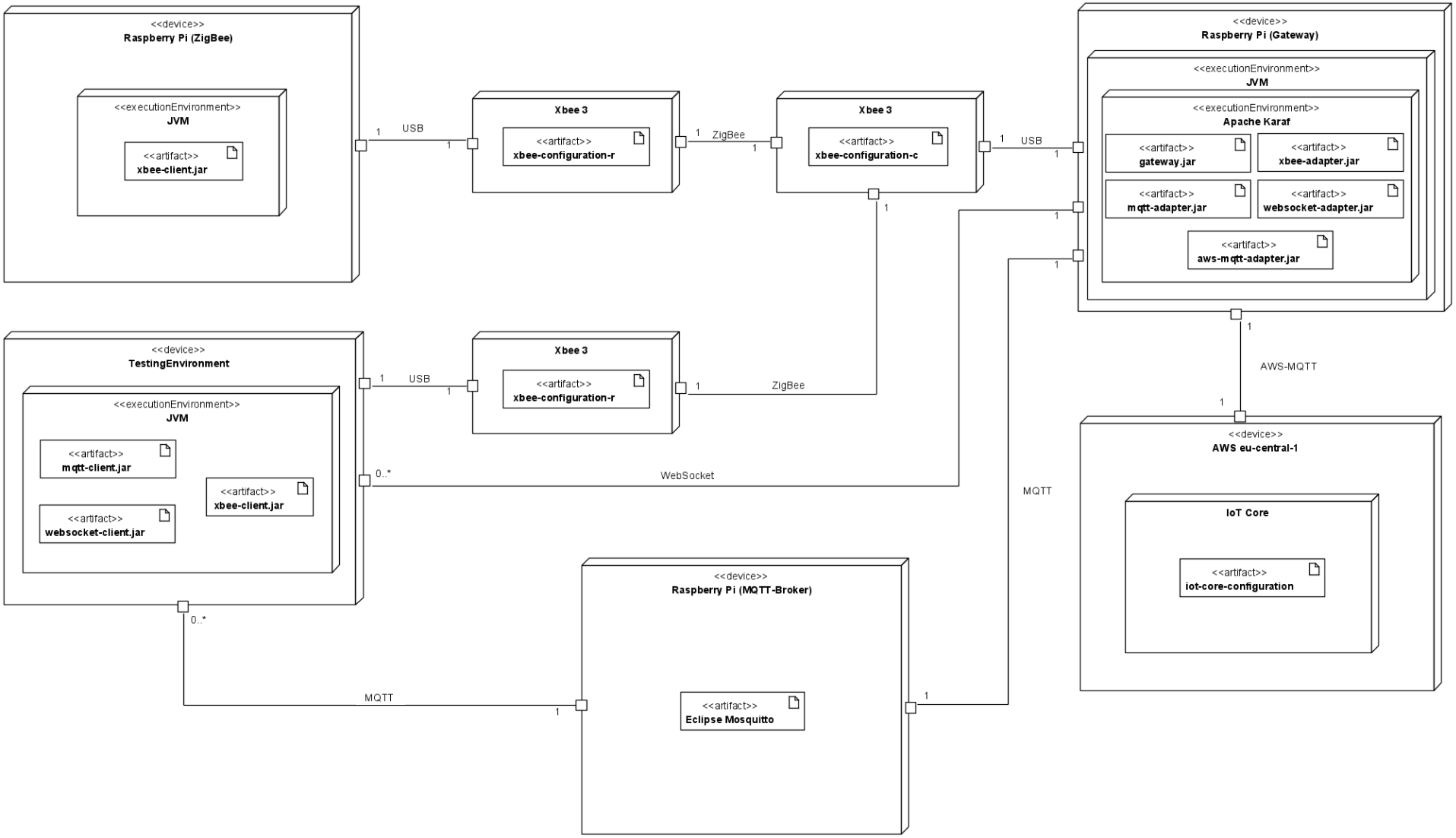


Figure 5.1: Deployment diagram

Various interfaces are defined for the gateway to communicate with the existing adapters. These interfaces are then bundled via another interface, which adapters can use via OSGi. This can be seen in more detail in the following figure 5.2. All available interfaces can be seen in the following figure 5.3 on the following page. In addition, all components are explained in more detail in the following table 5.2 on page 74.

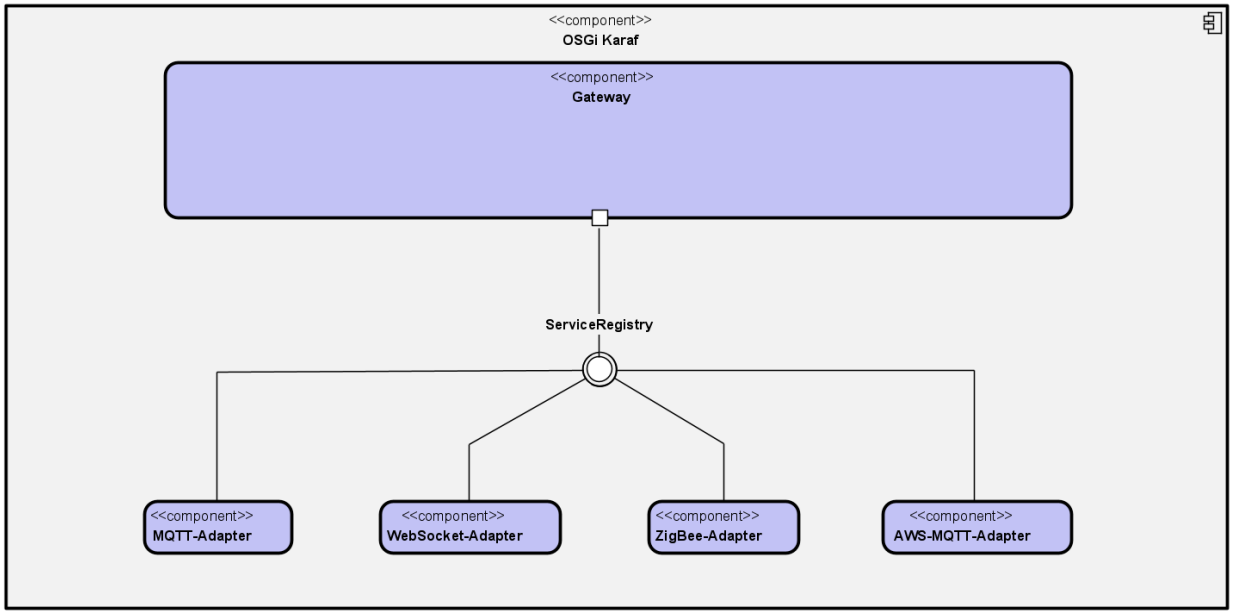


Figure 5.2: Adapter component diagram

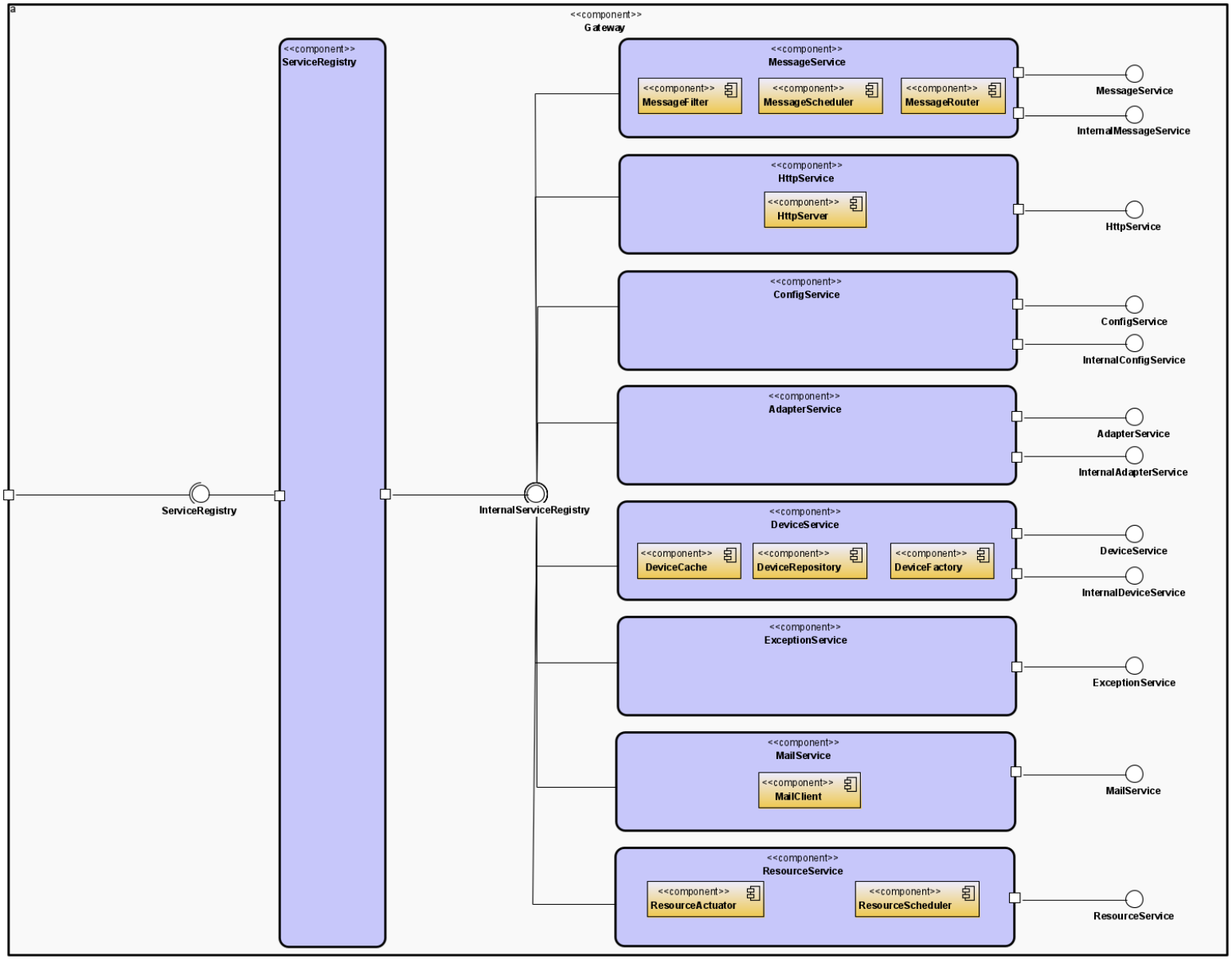


Figure 5.3: Gateway component diagram

Component	Description	Published interfaces
ServiceRegistry	The *ServiceRegistry* contains all services in the gateway and is used to start, stop or manage them.	▪ ServiceRegistry: This interface is freely available and can be used by any adapter to request interfaces to specific system components. ▪ InternalServiceRegistry: This interface is not freely available and can only be used by internal systems to also query internal interfaces of other systems.
MessageService	The *MessageService* is the main functionality of the gateway and generally serves as a message broker. Adapters can publish their received messages here, which are then processed by the service and forwarded to the final recipient.	▪ MessageService: This interface is freely available and can be used by adapters to publish messages. ▪ InternalMessageService: This interface is not freely available and is only used by other system components to measure sent and received packages or to change the behaviour of the message scheduler.
MessageFilter	The *MessageFilter* is a subcomponent of the message service. It filters all invalid messages out of the system or ensures that messages can be rejected in case of high loads.	-
MessageScheduler	The *MessageScheduler* is a subcomponent of the message service. It schedules all incoming messages based on priority and order.	-
MessageRouter	The message router is a subcomponent of the message services. It assigns the messages to the respective recipients and forwards the messages to system components or adapters.	-
HttpService	The *HttpService* provides all the functionality needed to make HTTP endpoints available. HTTP is an independent system component and not an adapter, as both WebSocket and an internal endpoint require HTTP. To prevent two HTTP servers, a separate system component was designed.	HttpService: This interface is not freely available and can be used to register or to delete routes from the HTTP server
HttpServer	The *HttpServer* is a subcomponent of the http service. It provides an implemented http server.	-

Table 5.2 continued from previous page

Component	Description	Published interfaces
ConfigService	The *ConfigService* provides the functionality to load config files from memory.	ConfigService: This interface is freely available and enable other system parts to extract configurations from loaded files
AdapterService	The *AdapterService* manage all registered adapters and also perform the registration process for these adapters.	▪ AdapterService: This interfaces is freely availabe and allows adapters to register themself in the gateway. ▪ InternalAdapterService: This interface is not freely available and is only used by other system components to manage all adapters or to get metrics from them.
DeviceService	The *DeviceService* manage all registered devices and also perform the registration process for these devices.	▪ DeviceService: This interfaces is freely availabe and allows other system components to retrieve information of devices. ▪ InternalDeviceService: This interface is not freely available and allows system components to initialise a registration process.
DeviceCache	The *DeviceCache* is a subcomponent of the device service. It provides a cache for devices to enable faster response time for already loaded information of devices.	-
DeviceRepository	The *DeviceRepository* is a subcomponent of the device service. It provides functionality to load device information from defined storage options.	-
DeviceFactory	The *DeviceFactory* is a subcomponent of the device service. It provides functionality to create devices from incoming messages.	-
ExceptionService	The *ExceptionService* handles all exceptions of the system and includes logic to handle them properly.	ExceptionService: This interface is freely available and can be used to dispatch exceptions from adapter or services.
MailService	The *MailService* provides functionality to send mails to defined adresses.	MailService: This interface is not freely available and can be used by system components to send mails to defined adresses.
MailClient	The *MailClient* is a subcomponent of the mail service. It encapsulates the mailing client implementation so it can be changed without further effort.	-
ResourceService	The *ResourceService* monitors the gateway itself and reacts properly to defined situations to prevent system overloads or failures.	ResourceService: This interface is not freely available and is currently only used to start this system component.

Table 5.2 continued from previous page

Component	Description	Published interfaces
ResourceActuator	The *ResourceActuator* is a subcomponent of the resource service. It measures all defined resources and provides them in an encapsulated object.	-
ResourceScheduler	The *ResourceScheduler* is a subcomponent of the resource service. It reacts on measured system resources with a defined logic to prevent system overloads or failure.	-
MQTT-Adapter	The *MQTT-Adapter* is an adapter of the gateway to receive and send packages over MQTT.	-
WebSocket-Adapter	The *WebSocket-Adapter* is an adapter of the gateway to receive and send packages over WebSocket.	-
ZigBee-Adapter	The *ZigBee-Adapter* is an adapter of the gateway to receive and send packages over ZigBee.	-
AWS-MQTT-Adapter	The *AWS-MQTT-Adapter* is an adapter of the gateway to receive and send packages from or to AWS.	-

Table 5.2: Gateway components descriptions

The following sections will go into more detail on individual system parts to explain them in more detail.

5.2 Service-Registry

The *ServiceRegistry* of the gateway is a central component. It is responsible for ensuring that all components are connected. This means that the services can call each other and be called by the adapters. The following figure shows all existing services and their concrete implementation.

Since not every adapter should be able to access internal gateway interfaces, there is a freely available interface and an internal interface for each available interface. The internal interface extends the freely available and the general service interface so that the gateway can start it. In order to further control access, a proxy class is created for each freely available object which can check the access regulations once again. The dependencies are shown in more detail in the following class diagram. If an interface is only available internally, this subdivision does not apply.

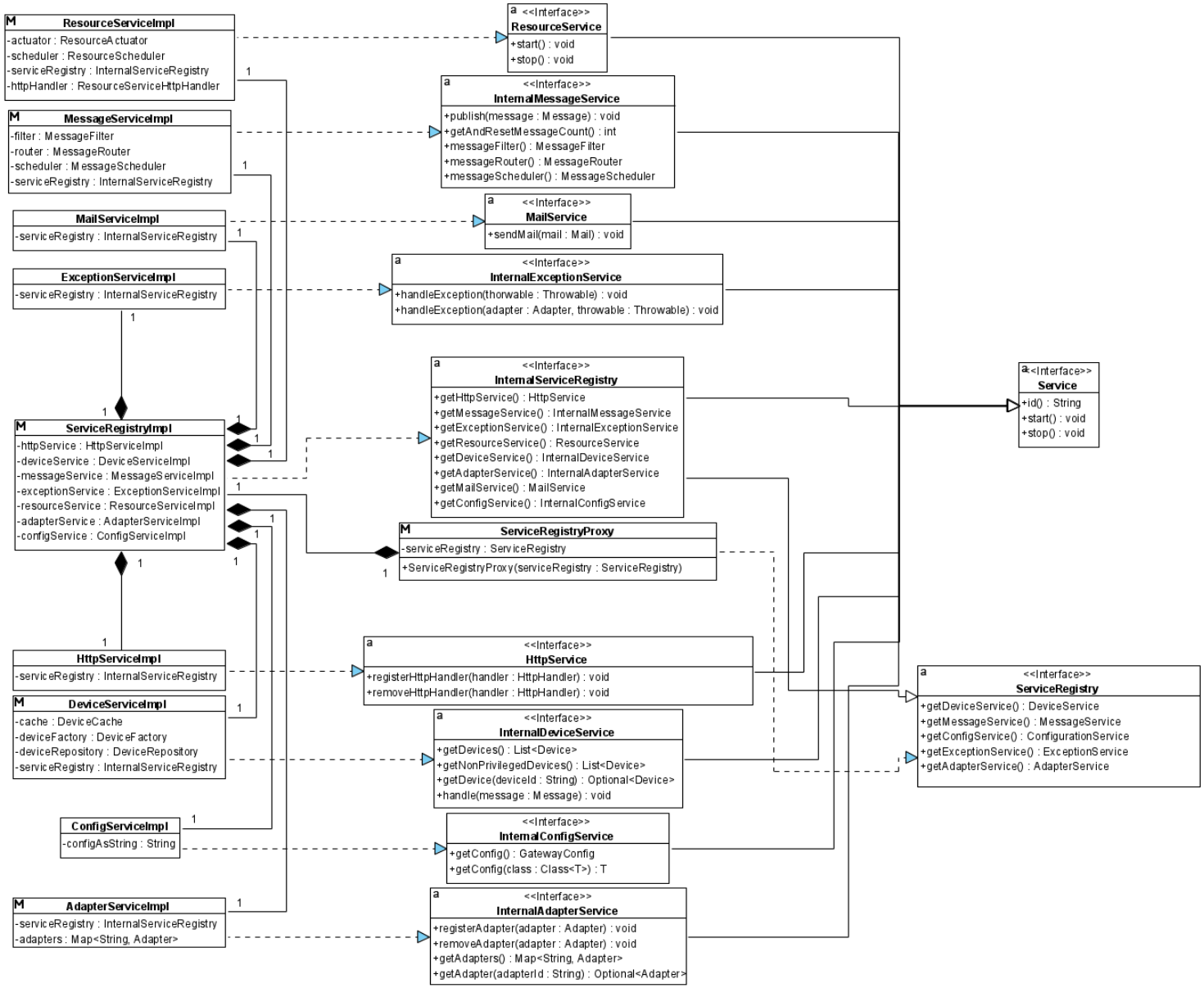


Figure 5.4: Service registry class diagram

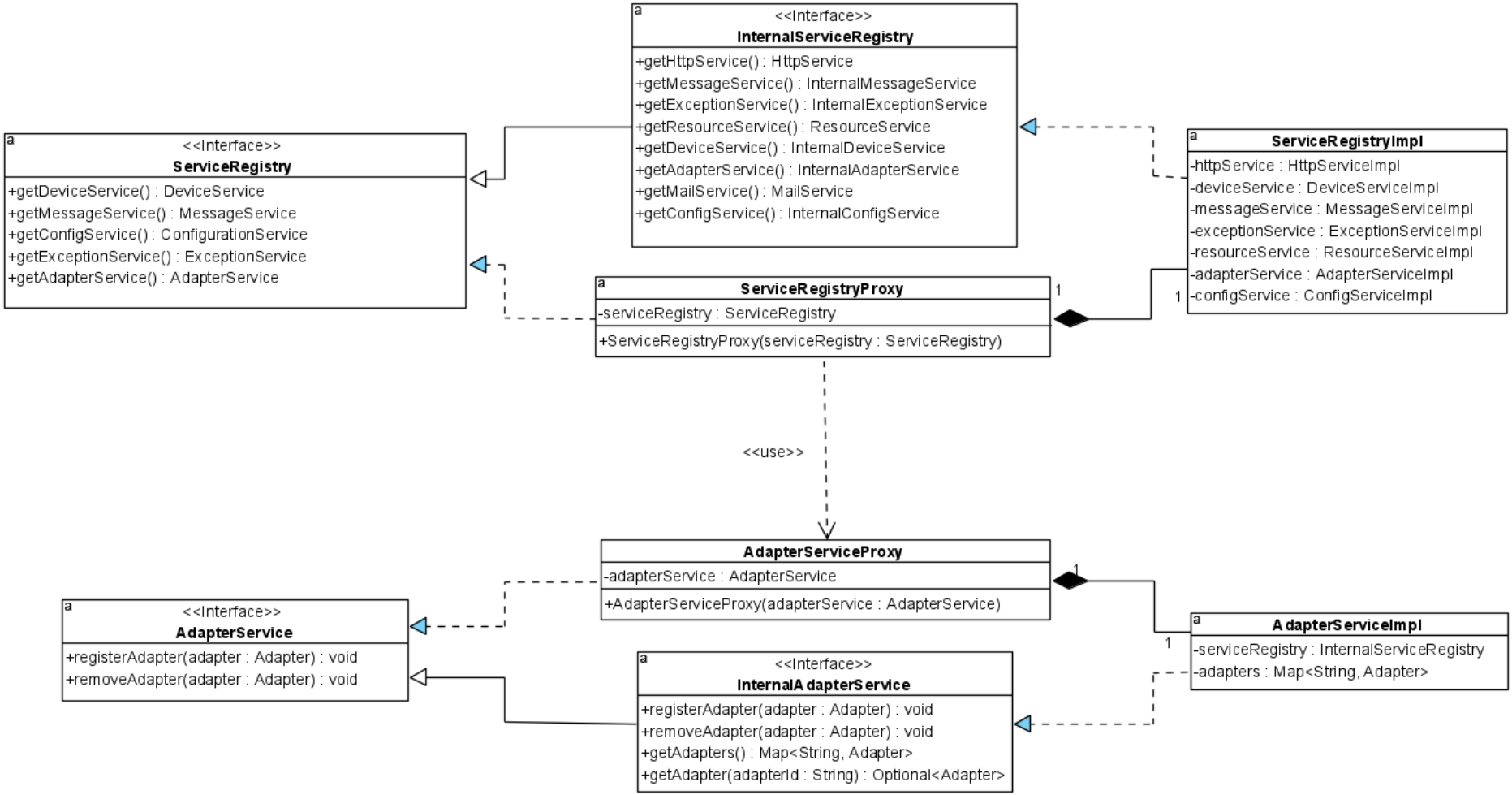


Figure 5.5: Service registry access restriction class diagram

5.3 Message-Service

The messaging of the gateway is one of the most critical problems to be solved. In this case, the goal was to develop an own message format to enable automatic registration or assign priorities. It would also have been possible to use the packets of other protocols, but protocols that act without prioritisation would have meant that messages could not be prioritised. Since many devices available today are not inherently capable of sending messages in this format, all other messages are considered legacy and populated with default values in the gateway. This way, compatibility can still be guaranteed.

Based on the requirements, the following message format was defined, see figure 5.6. The corresponding description of the individual fields follows in the table 5.3.

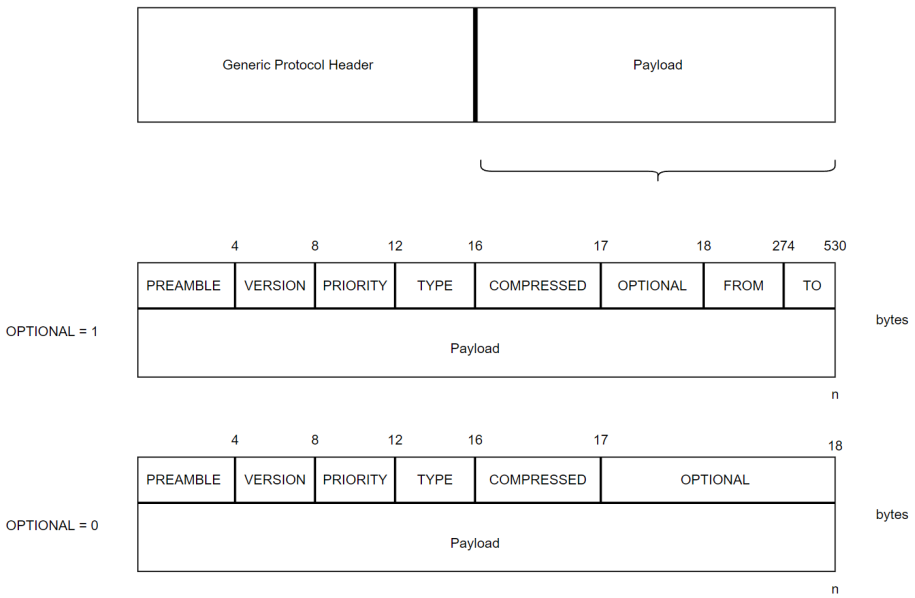


Figure 5.6: Package format

TYPE	Description
PREAMBLE	Defines a fixed value, this value is used as identification. If this value is set, then communication is in the specified format, if not, then the client is considered legacy.
VERSION	The version of the protocol, defined for further changes or upgrades.
PRIORITY	The priority of the message, currently supported are 0=BEST_EFFORT and 1=HIGH_PRIORITY.
TYPE	The type of this message, all types are explained in the following figure.
COMPRESSED	Indicates if this message is compressed with gzip.
OPTIONAL	Defines if the FROM and TO header is available, this option is specifically for low bandwidth protocols like ZigBee.
PAYLOAD	The payload of the message, the format does not matter.

Table 5.3: Message format description

TYPE	VALUE	Description
LEGACY	0	The message type indicate this as a legacy message. This message type is mainly used by the gateway to flag legacy messages.
GET	1	This message type indicates to a client that it should send his actual state to the sender of this message.
UPDATE	2	This message type indicates an update message containing a payload with data.
DELETE	3	This message type indicates that the actual state of the receiver should be deleted.
OK	100	This message type is a general answer for GET requests including a payload. UPDATE and DELETE are not considered to reduce network traffic.
LEGACY_CONNECT	200	This message type indicates a legacy connect request from a client. This message type is mainly used by the gateway to flag legacy messages if this was the first contact.
CONNECT	201	This message type indicates a connect request from a client. This message assumes a payload as a JSON-Object including three fields (int maxQoS, int currentQoS, boolean privileged), otherwise all values will be set to default values by the gateway.
DISCONNECT	202	This message type indicates a disconnect request from a client.
QOS_CHANGED	203	This message type indicates that the QoS level of a client changed. This message assumes that the payload is a JSON-Object including an integer field with the name 'currentQoS'.
CONECT_ACK	300	This message type indicates that the connect request was successful and the client can start to communicate. This message type is only allowed from the gateway.
DISCONNECT_ACK	301	This message type indicates that the disconnect request was successful. This message type is only allowed from the gateway.
QOS_REDUCE	302	This message type indicates that the gateway asked a client to reduce his QoS level. This message type is only allowed from the gateway.
QOS_INCREASE	303	This message type indicates that the gateway asked a client to increase his QoS level. This message type is only allowed from the gateway.

Table 5.4: Message type description

The following message format is implemented in the gateway according to the following class diagram 5.7. The *MessageParser* interface is used to transform the messages. There is a standard implementation for this that all adapters can use. If changes are necessary, the interface can be implemented itself. Since the payload transformation is also necessary for some messages, a *PayloadParser* interface was defined. Up to now, only the *DeviceService* uses this to perceive QoS changes.

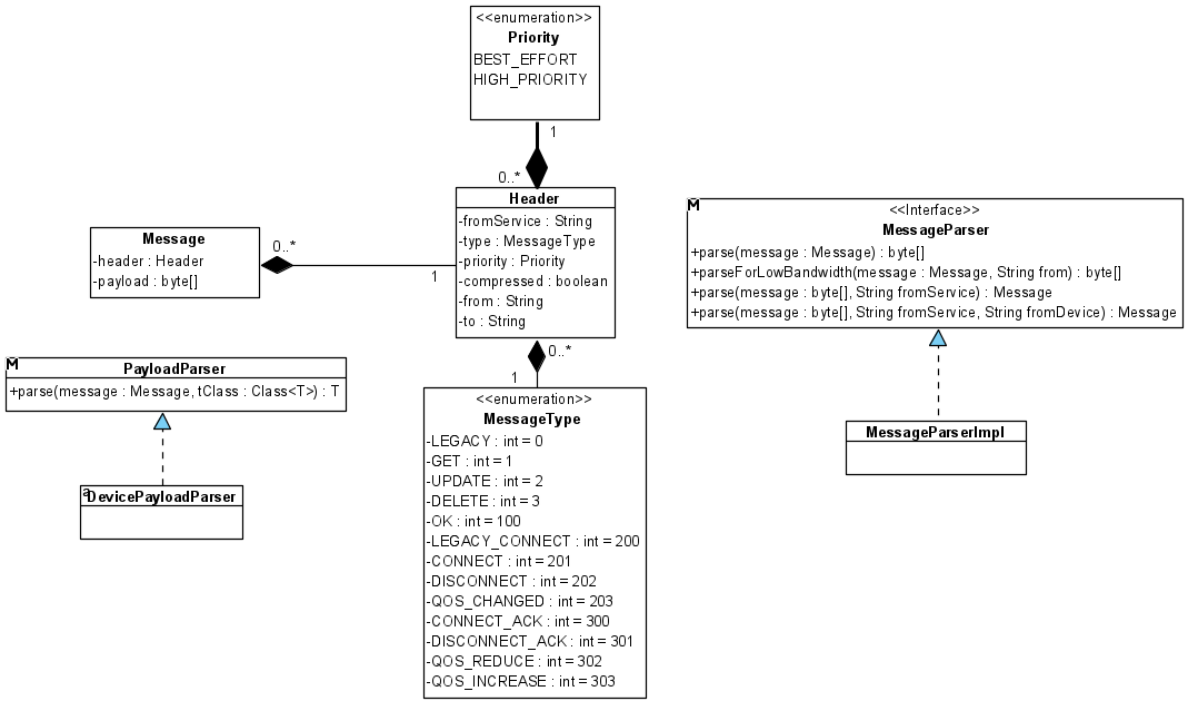


Figure 5.7: Messaging class diagram

In order to use the defined message format and to continue working with it, the following structure was designed in Figure 5.8 on the next page. This class diagram shows how the actual MessageService in the gateway is structured and which components are involved.

The basic process of receiving and sending messages is shown in the following figures 5.9 on page 81 and 5.10 on page 82.

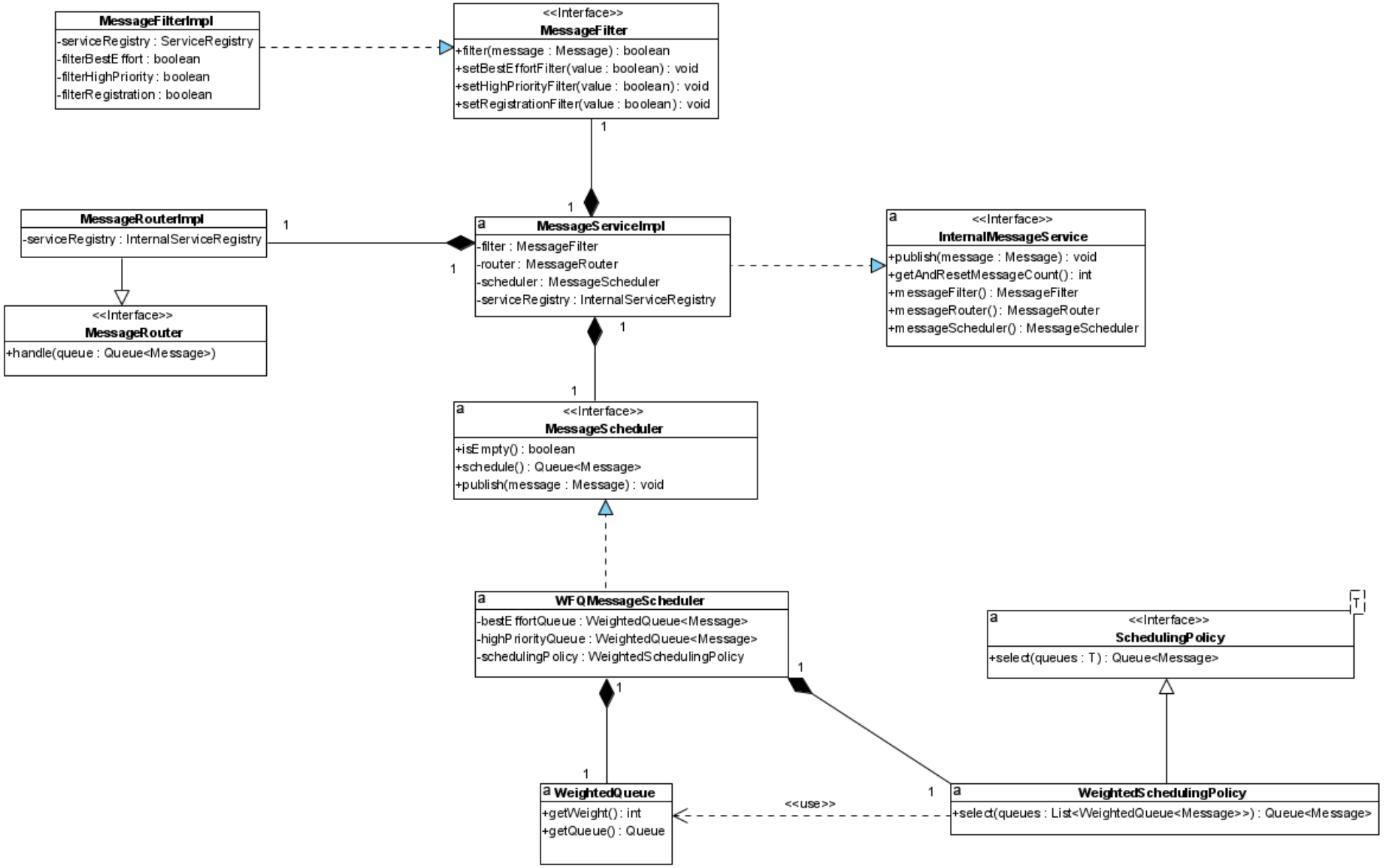


Figure 5.8: Message service class diagram

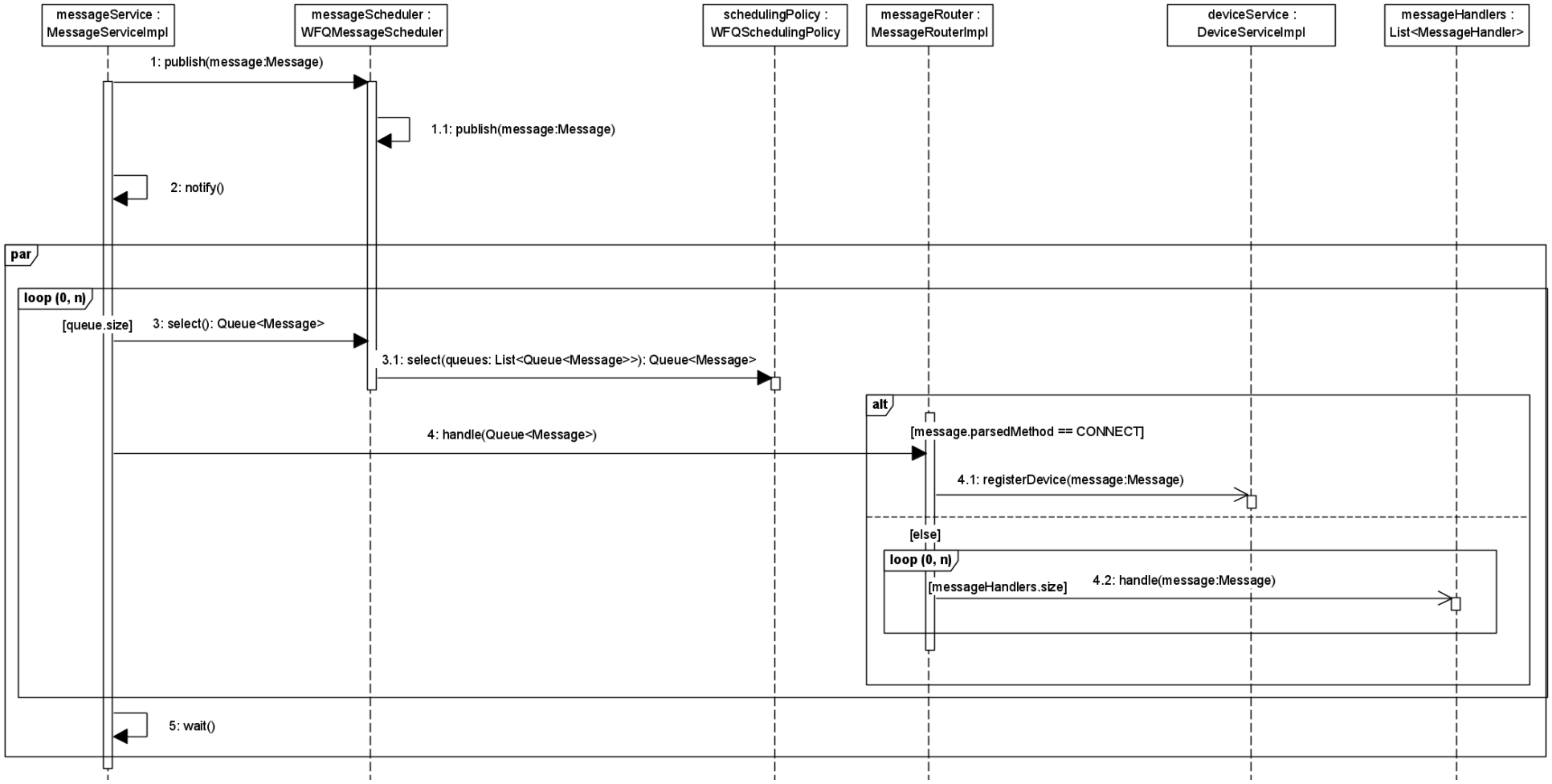


Figure 5.9: Publish message sequence diagram

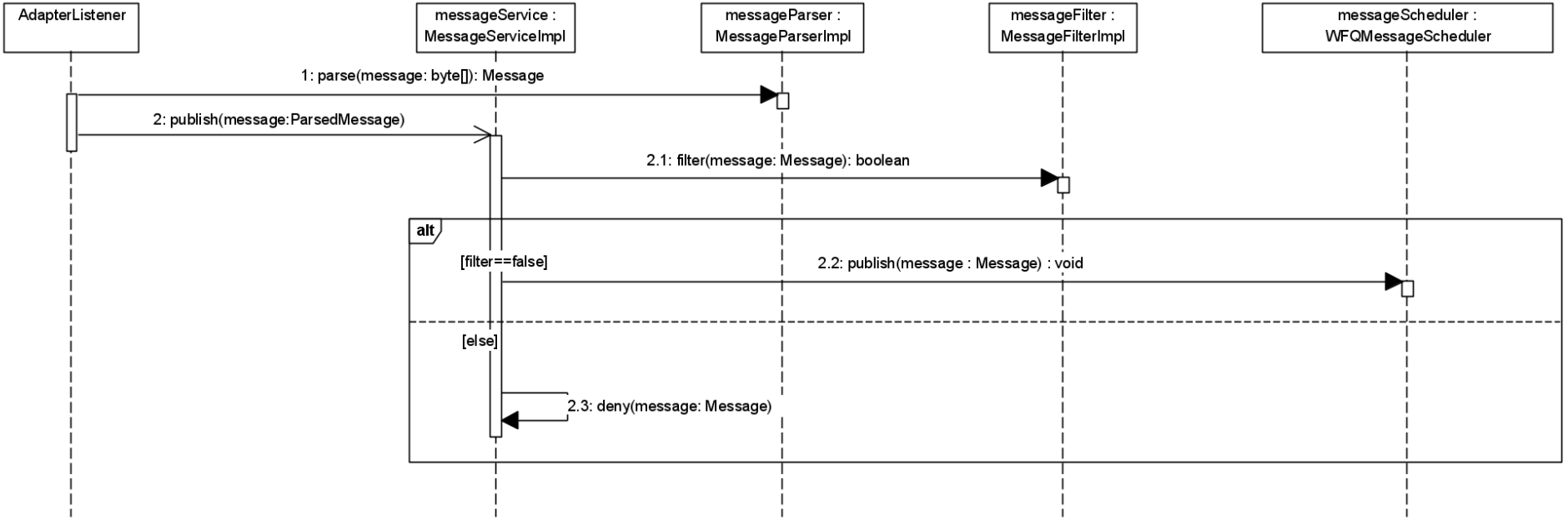


Figure 5.10: Receive message sequence diagram

5.4 Adapter-Service

For the adapters to be registered, a system is required to manage them. For this purpose, the *AdapterService* was designed and responsible for administering, registering, and removing adapters. Other services can also access the adapters in order to use them. A unique feature here is that the *HttpHandler* is registered in the *HttpService* during registration of the adapter itself. This is necessary because the used HTTP server has no idea about the adapters, and therefore the responsibility lies with the *AdapterService* to register them there.

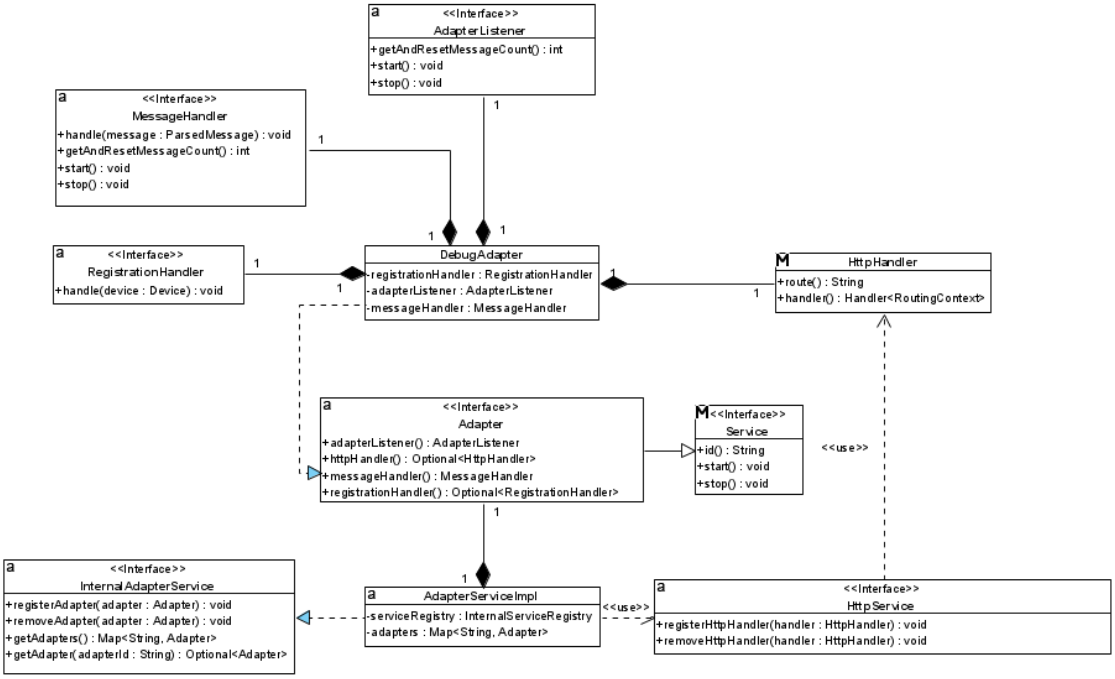


Figure 5.11: Adapter service class diagram

For an adapter to be used by the system, it also needs defined interfaces. For this purpose, there are the interfaces *AdapterListener*, *MessageHandler*, *RegistrationHandler* and *HttpHandler*. The *AdapterListener* is responsible for listening to the protocol to be implemented and then forwarding messages to the *MessageService*. The *MessageHandler*, on the other hand, receives messages from the *MessageService* to forward them to clients. The *RegistrationHandler* is notified when devices are to be registered and executes user-defined logic for the respective protocol. Finally, the *HttpHandler* makes it possible to define an HTTP endpoint to enable communication via WebSocket or HTTP. The *HttpHandler* and *RegistrationHandler* are optional and do not necessarily have to be implemented.

The actual registration process for an adapter has been made very simple to keep the system simple. The process is shown in the following figure 5.12 on the following page.

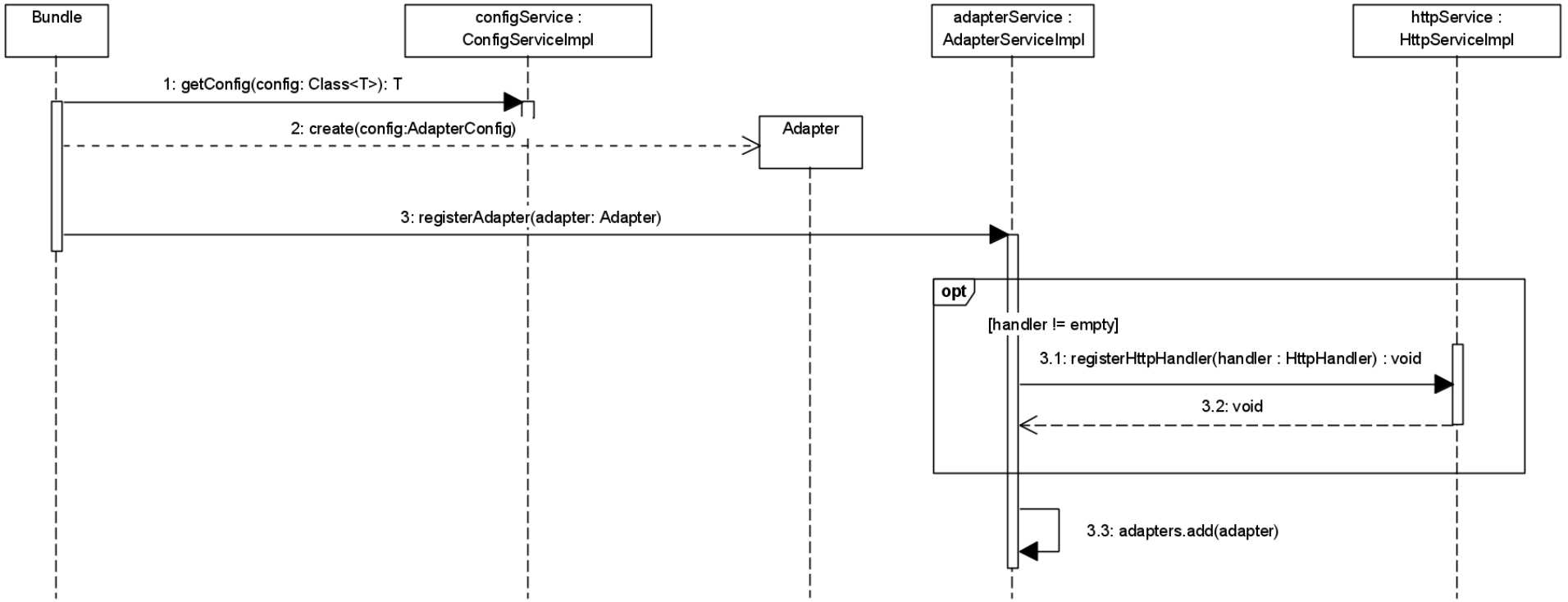


Figure 5.12: Register adapter sequence diagram

5.5 Device-Service

Of course, device management also needs its system component. For this reason, the *DeviceService* was designed which manages all devices and ensure that new devices can be registered or managed. The following figure 5.13 shows all the system components involved in the task mentioned. In addition, to speed up the service, not only a memory but also a cache was added to the design. This should allow all requests to be handled faster.

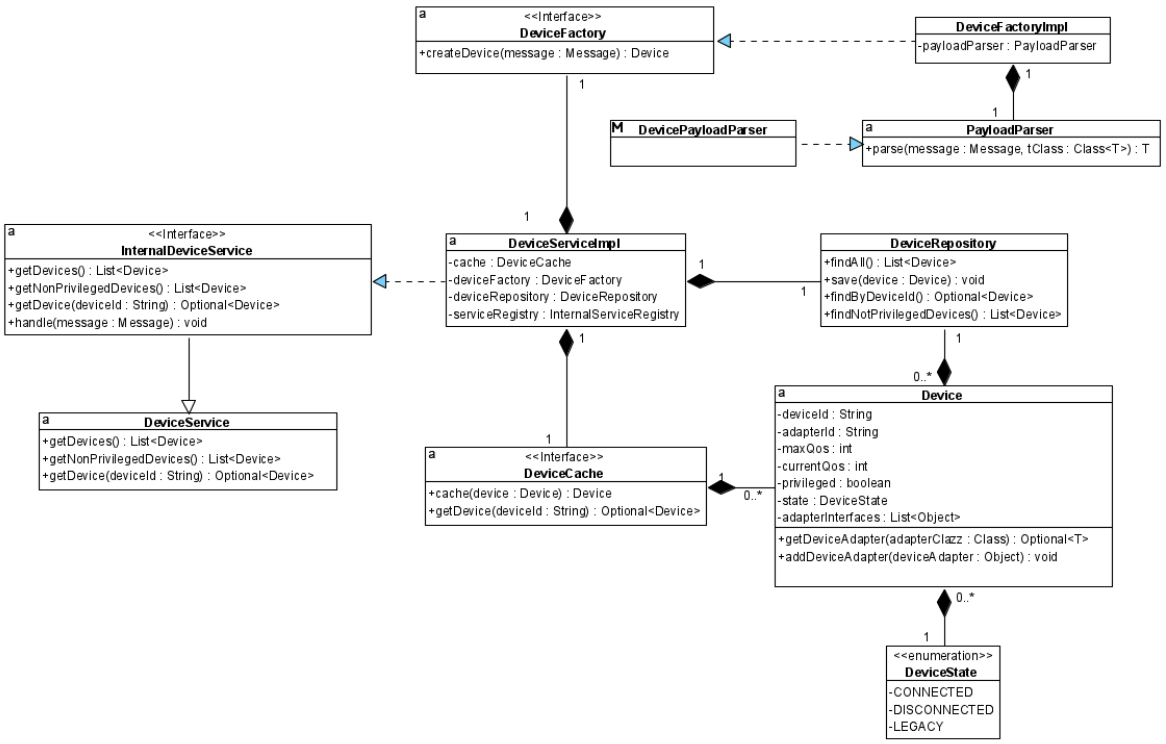


Figure 5.13: Device service class diagram

The registration process of devices is a bit more complicated, as several components are involved in this process. For this reason, the process is shown in full in the following figure 5.14 on the following page.

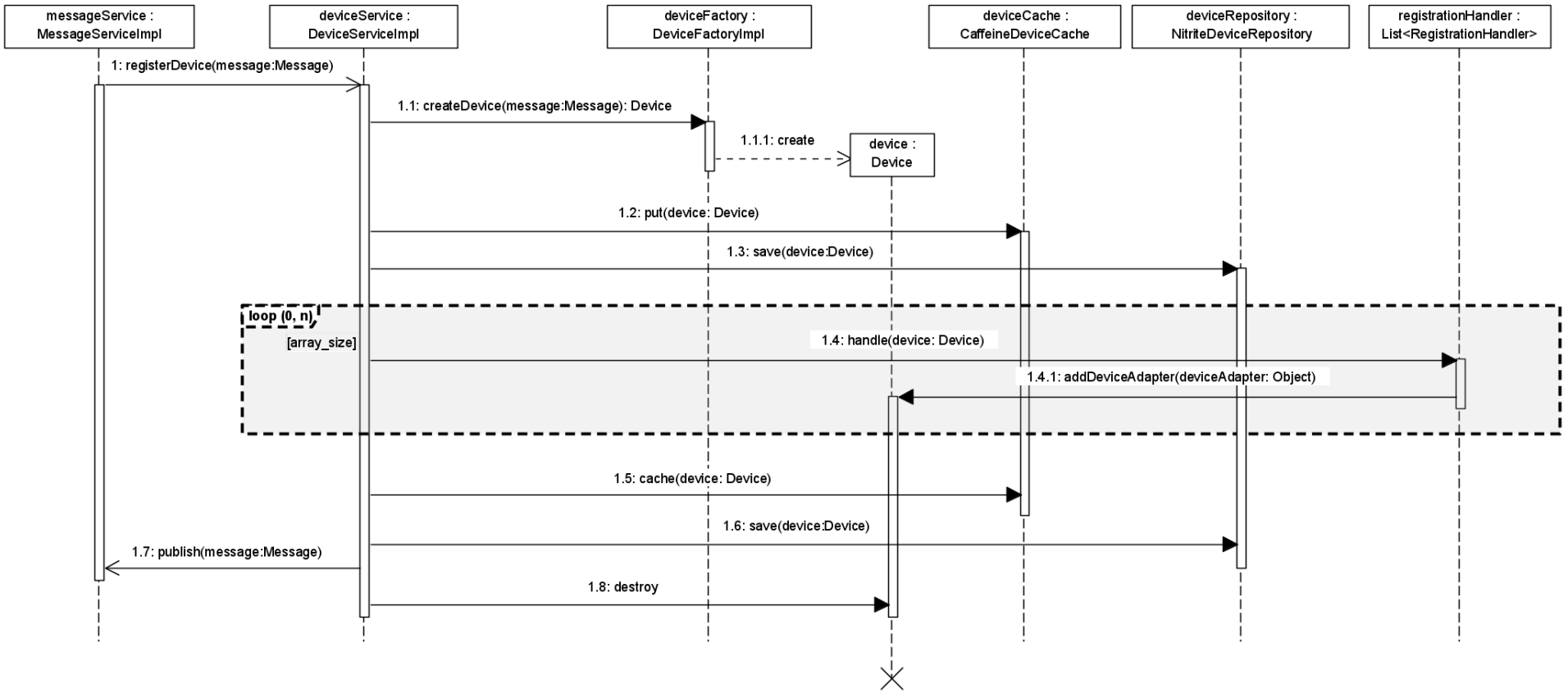


Figure 5.14: Register device sequence diagram

5.6 Resource-Service

The *ResourceService* was designed to monitor the gateway's resources and react to excessive resource consumption. This service can measure all specified values at intervals and then react to them. In addition, this service implements another *HttpHandler* to make the measured values available via an HTTP endpoint. For this reason, the following figure 5.15 shows all system components involved. The following figure 5.16 on the following page also shows how exactly the process of the service works.

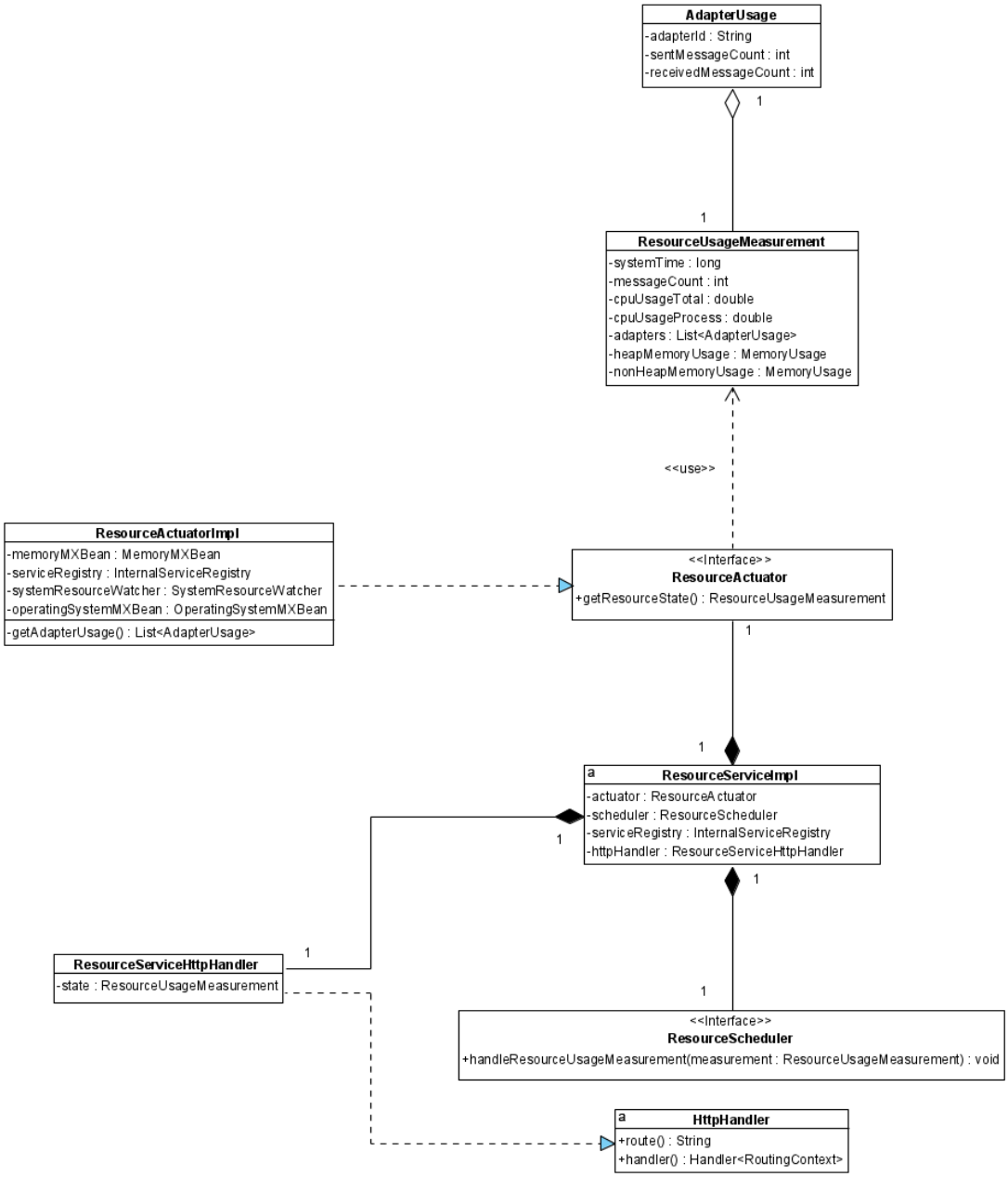


Figure 5.15: Resource service class diagram

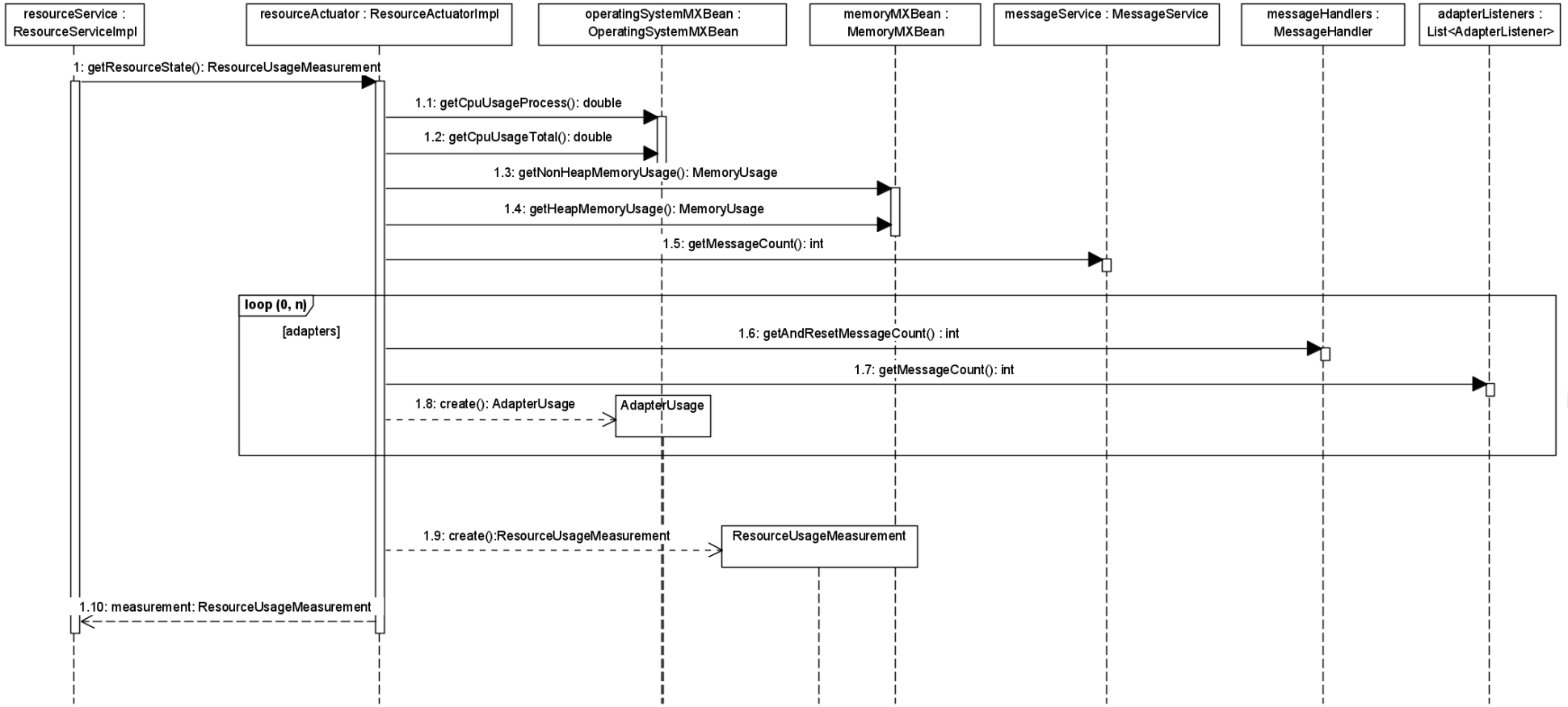


Figure 5.16: Monitor resources sequence diagram

5.7 Adapters

Since the gateway mainly takes overall administration tasks and defines interfaces, the adapters only have to implement and extend them with the protocol logic. Therefore, the design of these adapters is rather simple and strict compared to the gateway itself. For this reason, the class structure of the adapters is also very similar, as can be seen in the following figures 5.17, 5.18 on the following page, 5.19 on page 91, and 5.20 on page 92. In this case, only the WebSocket and the AWS MQTT adapter differ. This is due to either the required *HttpHandler* or *RegistrationHandler*.

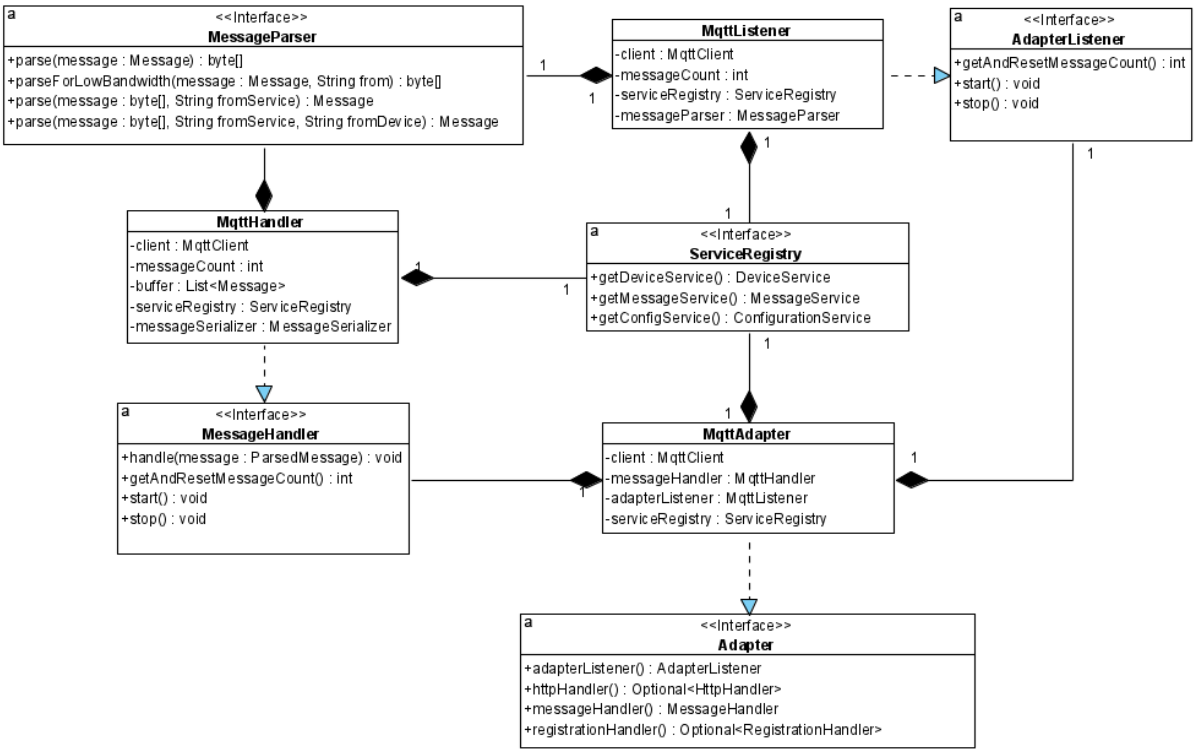


Figure 5.17: MQTT adapter class diagram

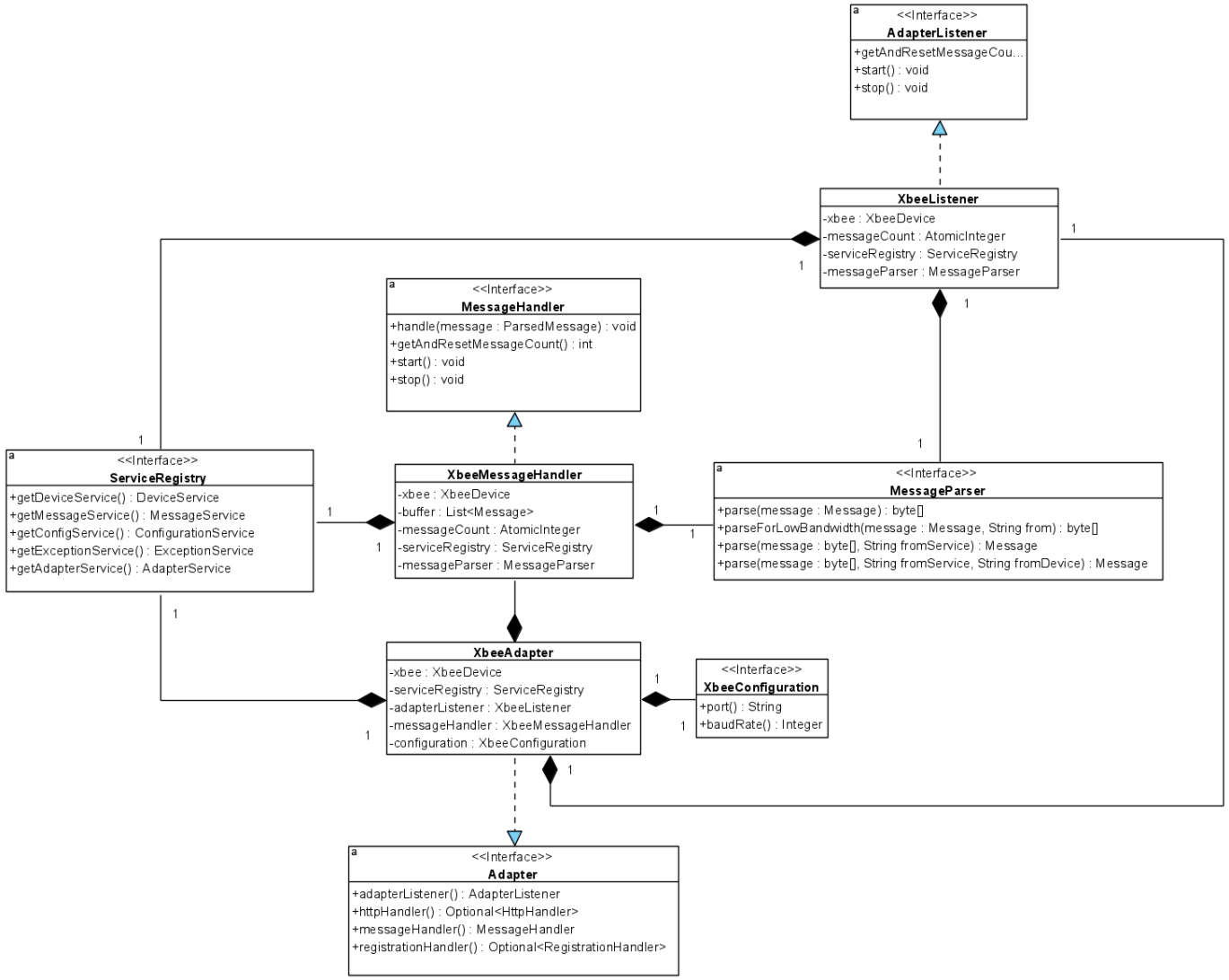


Figure 5.18: Xbee adapter class diagram

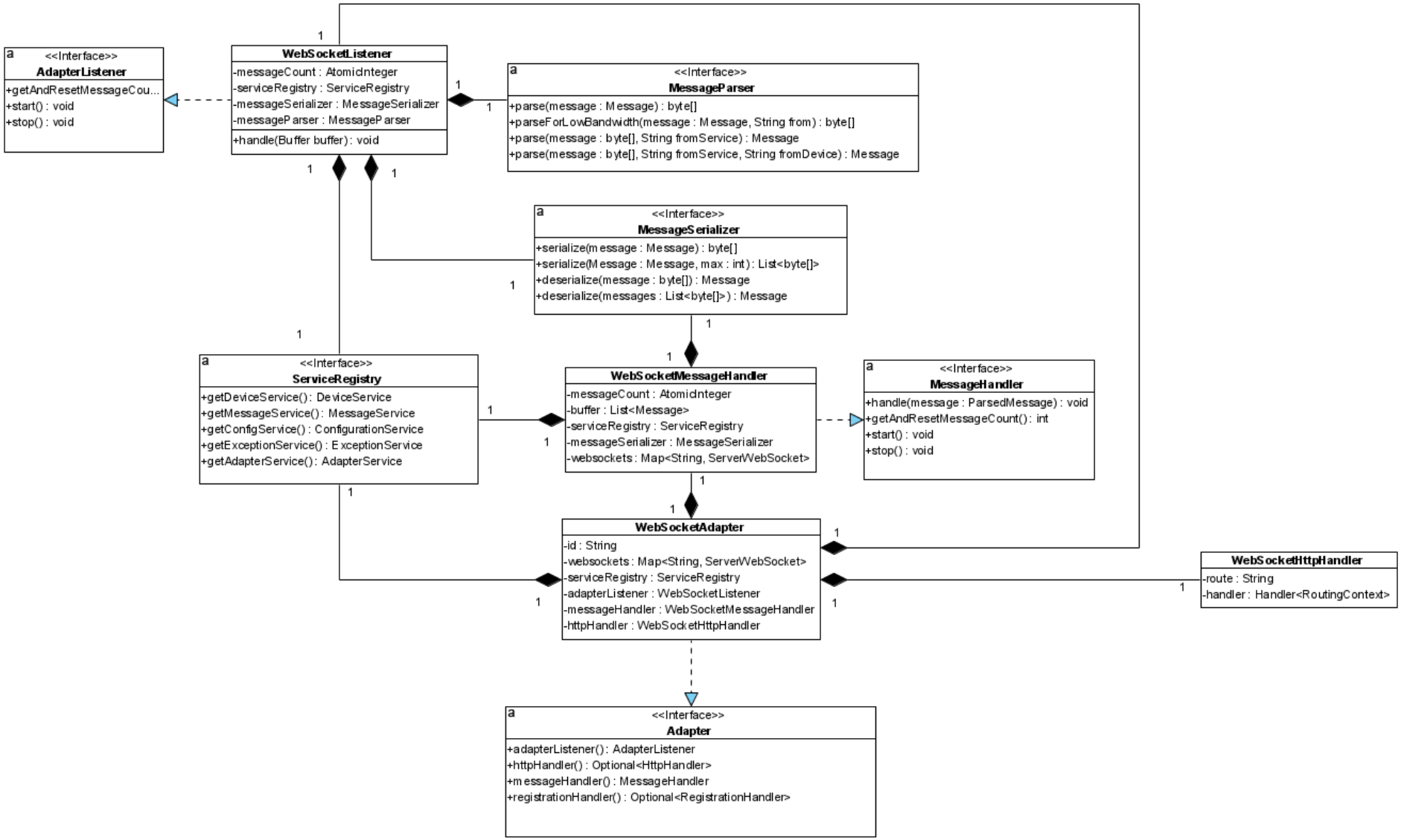


Figure 5.19: WebSocket adapter class diagram

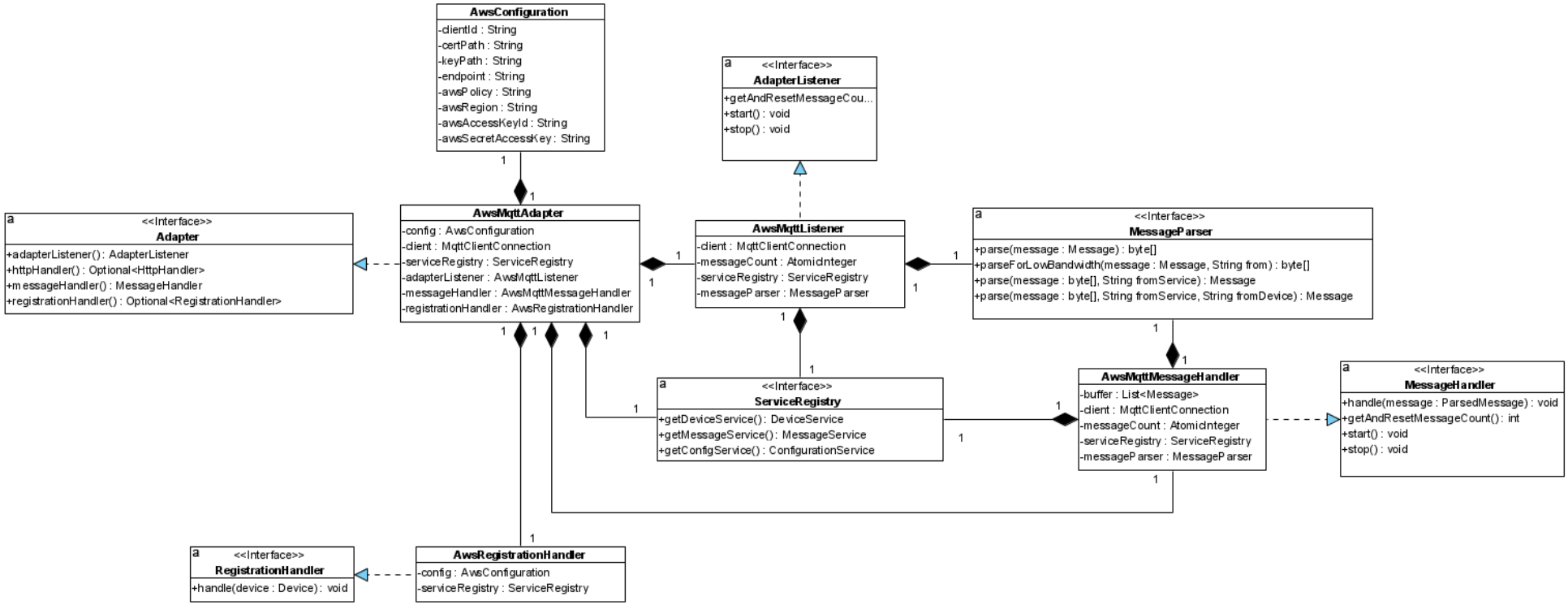


Figure 5.20: AWS MQTT adapter class diagram

Chapter 6

Implementation

There are several software components in this development, so a multi-module Maven project was created. This means that the individual components can be used by each other without any problems if this is required.

The entire structure can be seen in the following figure 6.1 and is described further in the following.

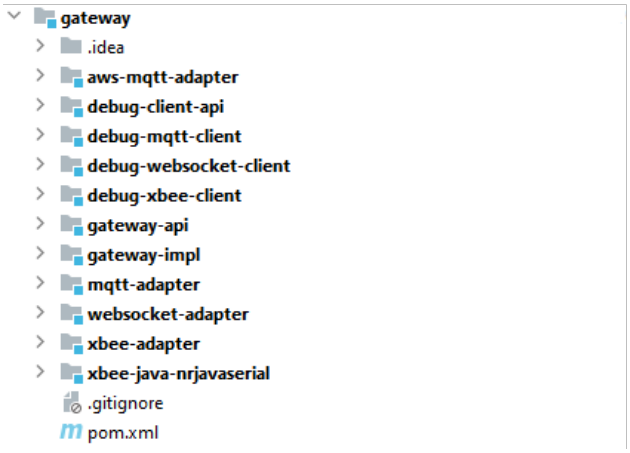


Figure 6.1: Maven project architecture

- *aws-mqtt-adapter*: Includes the adapter for the communication with AWS over MQTT. It depends on the gateway-api.
- *debug-client-api*: Includes a general API for testing purposes. It was not part of the design because this library was just created to prevent code duplication in all debug clients.
- *debug-mqtt-client*: Includes a debugging client for MQTT. It depends on the debug-client-api.
- *debug-websocket-client*: Includes a debugging client for WebSocket. It depends on the debug-client-api.
- *debug-xbee-xlient*: Includes a debugging client for ZigBee. It depends on the debug-client-api.
- *gateway-api*: Defines the complete gateway API, which was already explained in the design. It forms the basic framework for the gateway-impl module and all realised components. This project does not include any behaviour and is just for architectural purposes.
- *gateway-impl*: Defines the behaviour of the gateway while using the gateway-api module as a basic framework.

- *mqtt-adapter*: Includes the adapter for the communication with MQTT clients. It depends on the gateway-api.
- *websocket-adapter*: Includes the adapter for the communication with WebSocket clients. It depends on the gateway-api.
- *xbee-adapter*: Includes the adapter for the communication with ZigBee clients, based on XBee hardware. It depends on the gateway-api.
- *xbee-java-nrjavaserial*: Includes an interface to communicate with the XBee hardware. This project was not created in this implementation. It was used from DIGI International Inc. [Inc22a]. However it was recompiled against a more recent serial communication library, called "nrjavaserial" [Neu22], because the used "rxtx" [Jar22] library was not maintained anymore and only supported 32-bit architectures.

6.1 Gateway implementation

The gateway is split into an interface and an implementation, so replacing the actual implementation can be done without modifying other components.

The actual gateway was implemented strictly according to the design. However, some parts have not been explained in detail in the draft. These parts are explained further below.

This includes the algorithm for scheduling the messages. Since not many algorithms are compared with each other for efficiency in the thesis, a Weighted fair queueing (WFQ) approach was used here. The weights are defined statically and not calculated. The algorithm is shown in the following listing 3.

```
public class WFQSchedulingPolicy implements SchedulingPolicy<WeightedQueue<Message>[]> {
...
    @Override
    @SafeVarargs
    public final Queue<Message> select(WeightedQueue<Message>... queues) {
        Objects.requireNonNull(queues);
        ArrayBlockingQueue<Message> batch = new ArrayBlockingQueue<>(batchSize);
        while (batch.remainingCapacity() > 0
                && Arrays.stream(queues).map(Queue::size).reduce(0, Integer::sum) != 0) {
            for(WeightedQueue<Message> queue: queues){
                if(queue.isEmpty())
                    continue;
                for(int i = 0; i < queue.getWeight(); i++){
                    Message message = queue.poll();
                    if(message == null)
                        break;
                    batch.offer(message);
                }
            }
        }
        return batch;
    }
}
```

Listing 3: Clip of WFQSchedulingPolicy.java

Furthermore, the behaviour in the actual routing is not described further in the draft. Therefore, the following process is also shown as an example in this case in the following listing 4.

During routing, all connection attempts or QoS-relevant changes are handled by the *DeviceService*. Finally, the remaining packets are forwarded to the adapters based on their header or the upstream subscriber.

```
public class MessageRouterImpl implements MessageRouter {
...
    @Override
    public void handle(Queue<Message> messages) {
        for (Message message : messages) {
            Header header = message.getHeader();
            if (header.getType() == MessageType.CONNECT
                    || header.getType() == MessageType.LEGACY_CONNECT
                    || header.getType() == MessageType.DISCONNECT
                    || header.getType() == MessageType.QOS_CHANGED) {
                this.serviceRegistry.deviceService().handle(message);
            } else if (header.getTo() != null) {
                this.serviceRegistry
                        .deviceService()
                        .getDevice(header.getTo())
                        .map(Device::getAdapterId)
                        .ifPresent(adapterId -> this.routeToAdapter(adapterId, message));
            } else {
                this.serviceRegistry.configService()
                        .getConfig()
                        .getMappedAdapters()
                        .get(header.getFromService())
                        .getSubscriber()
                        .forEach(subscriber -> this.serviceRegistry
                                .adapterService()
                                .getAdapter(subscriber)
                                .map(Adapter::id)
                                .ifPresent(adapterId -> this.routeToAdapter(adapterId,
                                ↪  message)));
            }
        }
    }
...
}
```

Listing 4: Clip of MessageRouterImpl.java

Another critical point that has not been clarified in the draft is exception management. In the implementation, there are two types of exceptions, on the one hand, standard exceptions that have no further influence on the system and a so-called *ServiceException*. The *ServiceException* is further subdivided into *ServiceStartupException*, *ServiceRuntimeException* and *ServiceShutdownException*. If a service exception occurs, the component in which this exception occurred is always shut down. If this component is part of the gateway itself, the entire system is shut down. In addition, a *ServiceException* sends a mail to the administrator. This process can be seen again in the following listing 5 on the next page.

```
public class ExceptionServiceImpl implements InternalExceptionService {
...
    @Override
    public void handleException(Exception exception) {
        Objects.requireNonNull(exception);
        logger.error(ExceptionUtils.getStackTrace(exception));
        if (exception instanceof ServiceException) {
            if (!(exception instanceof ServiceShutdownException)) {
                ServiceException serviceException = (ServiceException) exception;
                if (serviceException.getService() instanceof Adapter) {
                    Adapter adapter = (Adapter) serviceException.getService();
                    this.serviceRegistry
                            .mailService()
                            .sendMail(String.format("Fatal exception in %s, shutting down
                            ↪  adapter!", adapter.id()),
                            ↪  ExceptionUtils.getStackTrace(exception));
                    this.serviceRegistry
```

```
                        .adapterService()
                        .removeAdapter(adapter);
                } else {
                    Service service = serviceException.getService();
                    if (!(service instanceof MailService)) {
                        this.serviceRegistry
                                .mailService()
                                .sendMail(String.format("Fatal exception in %s, shutting down
                                ↪  gateway!", service.id()),
                                ↪  ExceptionUtils.getStackTrace(exception));
                    }
                    logger.error("Can not recover from stacktrace, shutting down gateway!");
                    System.exit(-1);
                }
            }
        }
    }
}
```

Listing 5: Clip of ExceptionServiceImpl.java

Furthermore, the procedure for reacting to high loads was not described in the design. Therefore, two simple strategies were implemented to show how to react to such a problem modularly. The actual algorithm for this can be extended considerably, so it was deliberately implemented as a simple structure.

This algorithm stores the last ten resource measurements in a circular buffer. An average is calculated for the memory and computing load based on these measurements. This average is then combined with the configuration file to check whether it exceeds defined limits. If this is the case, the first strategy is applied. This consists of signalling to the clients that the resource utilisation is too high. This asks the client to reduce its QoS parameters if possible. If this strategy has not improved over a complete interval of measurements, gradual filters for registration, best effort and high prioritisation messages are activated. As soon as the gateway returns to the defined range, these filters will be deactivated, and it will return to regular operation. The automatic increase of QoS parameters was not considered, as this would require more complex algorithms to understand which applications need how much resource utilisation. The algorithm mentioned is also shown as an extract in the listing 6 on the following page below.

```
public class ResourceSchedulerImpl implements ResourceScheduler {
...
    @Override
    public void handle(ResourceUsageMeasurement measurement) {
        this.measurements.add(measurement);
        MessageFilter messageFilter = this.serviceRegistry.messageService().getMessageFilter();
        ResourceServiceConfig config =
        ↪  this.serviceRegistry.configService().getConfig().getMonitoring();

        double memoryUsage = this.measurements
                .stream()
                .mapToDouble(this::getCurrentMemoryConsumption)
                .average()
                .orElse(0);

        double cpuUsage = this.measurements
                .stream()
                .mapToDouble(ResourceUsageMeasurement::getCpuUsageTotal)
                .average()
                .orElse(0);

        boolean highMemoryUsage = this.isHigherThan(memoryUsage,
                config.getPrimaryMemoryThreshold(),
                config.getSecondaryMemoryThreshold(),
                config.getTertiaryMemoryThreshold());
```

```
        boolean highCpuUsage = this.isHigherThan(cpuUsage,
                config.getPrimaryCpuThreshold(),
                config.getSecondaryCpuThreshold(),
                config.getTertiaryCpuThreshold());

        if (highMemoryUsage || highCpuUsage) {
            this.timeSinceLastHighResourceConsumption = System.currentTimeMillis();
            this.serviceRegistry.deviceService()
                    .getNotPrivilegedDevices()
                    .forEach(this::publishQoSReduceMessage);
            if (this.timeSinceHighResourceConsumption == 0) {
                this.timeSinceHighResourceConsumption = System.currentTimeMillis();
            } else if (this.timeSinceHighResourceConsumption < (System.currentTimeMillis() - 10 *
            ↪  config.getInterval())) {
                messageFilter.setRegistrationFilter(config.getPrimaryMemoryThreshold() >=
                ↪  memoryUsage ||
                        config.getPrimaryCpuThreshold() >= cpuUsage);

                messageFilter.setBestEffortFilter(config.getSecondaryMemoryThreshold() >=
                ↪  memoryUsage ||
                        config.getSecondaryCpuThreshold() >= cpuUsage);

                messageFilter.setHighPriorityFilter(config.getTertiaryMemoryThreshold() >=
                ↪  memoryUsage ||
                        config.getTertiaryCpuThreshold() >= cpuUsage);
            }
        } else if (timeSinceLastHighResourceConsumption != 0
                && this.timeSinceLastHighResourceConsumption < (System.currentTimeMillis() - 10 *
                ↪  config.getInterval())) {
            this.timeSinceHighResourceConsumption = 0;
            this.timeSinceLastHighResourceConsumption = 0;
            messageFilter.setRegistrationFilter(false);
            messageFilter.setBestEffortFilter(false);
            messageFilter.setHighPriorityFilter(false);
        }
    }
...
}
```

Listing 6: Clip of ResourceSchedulerImpl.java

The last important implementation detail is the build instruction in the maven file. Finally, the bundle is assembled with all the required dependencies and stored in the local maven directory.

Essential here are the instructions on which packages are exported and imported and which dependencies are embedded. First, the API for the gateway is exported for other applications so that the adapters, for example, can use the classes in this package. All classes that are not listed as embedded dependencies must be imported. For this reason, a wildcard is also used for this. In addition, packages from *com.sun.** must be imported, as these are not automatically recognised by the build tool. Some dependencies had to be embedded and are subsequently only callable in the classpath of the exported application. The main reason was that these third-party applications were not compatible with the OSGi environment and had problems with the classloading strategies. The following listing 7 on the next page shows the important excerpt from this file.

```
...
<plugin>
    <groupId>org.apache.felix</groupId>
    <artifactId>maven-bundle-plugin</artifactId>
    <version>5.1.4</version>
    <extensions>true</extensions>
    <configuration>
        <instructions>
            <Bundle-SymbolicName>
```

```
                ${project.groupId}.${project.artifactId}
            </Bundle-SymbolicName>
            <Bundle-Name>${project.artifactId}</Bundle-Name>
            <Bundle-Version>${project.version}</Bundle-Version>
            <Bundle-Activator>
                de.hsbremen.iot.gateway.impl.Gateway
            </Bundle-Activator>
            <Export-Package>
                de.hsbremen.iot.gateway.api.*
            </Export-Package>
            <Import-Package>
                com.sun.*;resolution:=optional,
                *;resolution:=optional
            </Import-Package>
            <Embed-Dependency>nitrite; h2-mvstore; objenesis; jackson-databind;
            ↪  jackson-annotations; jasypt; podam; jcip-annotations;
            ↪  javax.annotation-api;validation-api; inline=true </Embed-Dependency>
            <Embed-Transitive>true</Embed-Transitive>
        </instructions>
    </configuration>
</plugin>
...
```

Listing 7: Clip of gateway-impl pom.xml

6.2 Adapter implementation

Based on the design, the implementation of the adapters has turned out to be very small. Only a functionality for receiving and sending messages had to be implemented. The gateway itself provides the remaining functionality. Therefore, the implementation is described below using the *xbee-adapter* as an example.

In addition to the design, the functionality for the OSGi environment has also been implemented. This is also the same for all adapters.

```
public class XbeeBundle implements BundleActivator, ServiceListener {

...
    @Override
    public void start(BundleContext bundleContext) {
        this.ctx = bundleContext;
        this.serviceReference = this.ctx.getServiceReference(ServiceRegistry.class);
        if (this.serviceReference != null) {
            ServiceRegistry serviceRegistry = this.ctx.getService(serviceReference);
            this.adapter = new XbeeAdapter(serviceRegistry);
            serviceRegistry.adapterService().registerAdapter(this.adapter);
        }
    }

    @Override
    public void stop(BundleContext bundleContext) {
        if (this.serviceReference != null) {
            ServiceRegistry serviceRegistry = this.ctx.getService(serviceReference);
            serviceRegistry.adapterService().removeAdapter(this.adapter);
                        this.ctx.ungetService(this.serviceReference);
        }
    }

    @Override
    public void serviceChanged(ServiceEvent serviceEvent) {
        switch (serviceEvent.getType()) {
            case ServiceEvent.REGISTERED:
```

```
                if (this.serviceReference != null) {
                    ServiceRegistry serviceRegistry = this.ctx.getService(serviceReference);
                    this.adapter = new XbeeAdapter(serviceRegistry);
                    serviceRegistry.adapterService().registerAdapter(this.adapter);
                }
                break;
            case ServiceEvent.UNREGISTERING:
                if (this.serviceReference != null) {
                    ServiceRegistry serviceRegistry = this.ctx.getService(serviceReference);
                    serviceRegistry.adapterService().removeAdapter(this.adapter);
                }
                break;
        }
    }
}
```

Listing 8: Clip of XbeeBundle.java

The adapter is started from the *XbeeBundle*, which is responsible for loading all required resources and providing all components for the gateway. Since the adapter is independent of OSGi, the gateway can also be used without much effort without OSGi.

```
...
    public XbeeAdapter(ServiceRegistry serviceRegistry) {
        this.serviceRegistry = serviceRegistry;
        this.xbeeMessageHandler = new XbeeMessageHandler(this);
        this.xbeeMessageListener = new XbeeMessageListener(this);
    }

    @Override
    public String id() {
        return ID;
    }

    @Override
    public void start() {
        try {
            ConfigService configService = this.serviceRegistry.configService();
            XbeeAdapterConfig xbeeAdapterConfig = configService.getConfigFromBase(ID,
            ↪  XbeeAdapterConfig.class);
            this.xBeeDevice = new XBeeDevice(xbeeAdapterConfig.getPort(),
            ↪  xbeeAdapterConfig.getBaudRate());
            this.xBeeDevice.open();
            this.xbeeMessageListener.start();
            this.xbeeMessageHandler.start();
        } catch (Exception ex) {
            this.serviceRegistry
                    .exceptionService()
                    .handleException(new ServiceStartupException(ex, this));
        }
    }

    @Override
    public void stop() {
        try {
            this.xbeeMessageHandler.stop();
            this.xbeeMessageListener.stop();
            this.xBeeDevice.close();
        } catch (Exception ex) {
            this.serviceRegistry
                    .exceptionService()
                    .handleException(new ServiceShutdownException(ex, this));
        }
    }
...
```

Listing 9: Clip of XbeeAdapter.java

As shown, only the *MessageListener* and the *MessageHandler* have to be defined on top of that. These are then responsible for sending and receiving messages via ZigBee. Apart from this, all the remaining logic is located in the gateway itself.

```
public class XbeeMessageListener implements MessageListener {
...
    @Override
    public void start() {
            ...
        xbee.addDataListener(xBeeMessage -> {
            try {
                RemoteXBeeDevice device = xBeeMessage.getDevice();
                device.readDeviceInfo(); //without reading the node id is null
                String deviceId = device.getNodeID();
                byte[] payload = xBeeMessage.getData();
                Message parsedMessage;
                if (messageParser.isLegacy(payload)) {
                    parsedMessage = messageParser.parseLegacy(payload, xbeeAdapter.id(),
                    ↪  deviceId);
                } else {
                    parsedMessage = messageParser.parse(payload, xbeeAdapter.id(), deviceId);
                }
                this.xbeeAdapter.getServiceRegistry()
                        .messageService()
                        .publish(parsedMessage);
                this.messageCount.incrementAndGet();
            } catch (Exception ex) {
                this.xbeeAdapter.getServiceRegistry()
                        .exceptionService()
                        .handleException(ex);
            }
        });
    }
...
}

public class XbeeMessageHandler implements MessageHandler {
...
    @Override
    public void run() {
            ...
        while (this.isRunning) {
            while(!this.queue.isEmpty()){
                try {
                    Message message = queue.poll();
                    byte[] parsedMessage = messageParser.parseForLowBandwidth(message);
                    if (message.getHeader().getTo() == null) {
                        xbee.sendBroadcastData(parsedMessage);
                    } else {
                        RemoteXBeeDevice remoteDevice =
                        ↪  xbee.getNetwork().discoverDevice(message.getHeader().getTo());
                        xbee.sendData(remoteDevice, parsedMessage);
                    }
                    this.messageCount.incrementAndGet();
                } catch (Exception ex) {
                    this.xbeeAdapter.getServiceRegistry()
                            .exceptionService()
                            .handleException(ex);
                }
            }
            synchronized (this) {
                this.wait();
```

```
            }
        }
        ...
    }
...
}
```

Listing 10: Clip of the XbeeMessageListener.java and XbeeMessageHandler.java

6.3 Building projects

By using Apache Maven, building the projects is very simple. For this purpose, the command mvn:install can be entered in each project. However, it is recommended to execute this command in the parent directory, as this will build all projects in the correct order.

If everything has worked correctly, the following message should appear.

```
[INFO] Reactor Summary:
[INFO]
[INFO] gateway 1.0-SNAPSHOT ............................ SUCCESS [  0.141 s]
[INFO] gateway-api 1.0-SNAPSHOT ........................ SUCCESS [  0.746 s]
[INFO] gateway-impl 1.0-SNAPSHOT ....................... SUCCESS [  8.145 s]
[INFO] mqtt-adapter 1.0-SNAPSHOT ....................... SUCCESS [  0.745 s]
[INFO] xbee-java-nrjavaserial 1.3.1 .................... SUCCESS [  2.436 s]
[INFO] xbee-adapter 1.0-SNAPSHOT ....................... SUCCESS [  0.679 s]
[INFO] websocket-adapter 1.0-SNAPSHOT .................. SUCCESS [  0.660 s]
[INFO] debug-client-api 1.0-SNAPSHOT ................... SUCCESS [  0.507 s]
[INFO] debug-mqtt-client 1.0-SNAPSHOT .................. SUCCESS [  0.328 s]
[INFO] debug-xbee-client 1.0-SNAPSHOT .................. SUCCESS [  0.282 s]
[INFO] debug-websocket-client 1.0-SNAPSHOT ............. SUCCESS [  0.342 s]
[INFO] aws-mqtt-adapter 1.0-SNAPSHOT ................... SUCCESS [  8.756 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  24.605 s
[INFO] Finished at: 2022-04-15T17:13:47+02:00
[INFO] ------------------------------------------------------------------------
```

Listing 11: Install gateway-impl command

It is essential to mention that the built projects can be found in the local Maven repository afterwards. Apache Karaf will look for them in this folder during the installation instructions. This means that the artifacts can only be deployed on the local system they are built on. For other machines, they have to be copied to the Maven directory of the respective machine. Typically the artifacts would be located in a remote repository, but this is not suitable for this thesis, as this repository would have to be maintained in the further course.

6.4 Apache Karaf setup

Installing Apache Karaf is not as easy as installing the MQTT broker, as the required resources must be downloaded directly from the manufacturer. In addition, the respective Java distribution and Maven must, of course, be installed on the computer. This means that the following steps have to be followed.

```
sudo apt update
sudo apt install default-jdk
sudo apt install maven
wget https://dlcdn.apache.org/karaf/4.3.6/apache-karaf-4.3.6.tar.gz
tar -xf apache-karaf-4.3.6.tar.gz
rm apache-karaf-4.3.6.tar.gz
```

Listing 12: Apache Karaf setup

To enable the remote session and the logging, some changes are required. First, the following lines must be commented out in the file $KARAF_HOME/etc/user.properties.

```
karaf = karaf,_g_:admingroup
_g_\:admingroup = group,admin,manager,viewer,systembundles,ssh
```

Listing 13: Apache Karaf configuration

Then, in the file $KARAF_HOME/etc/org.ops4j.pax.logging.cfg, replace everything with the following entry.

```
org.ops4j.pax.logging.log4j2.config.file=${karaf.etc}/log4j2.xml
```

Listing 14: Apache Karaf logging configuration

Finally, the following file must be placed as log4j2.xml in the folder $KARAF_HOME/etc.

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="INFO">
    <Appenders>
        <Console name="console" target="SYSTEM_OUT">
            <PatternLayout pattern="[%-5level] %d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %c{1} - %msg%n"
            ↪ />
        </Console>
        <File name="main" fileName="${sys:user.home}/.gateway/logs/main.log">
            <PatternLayout pattern="[%-5level] %d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %c{1} - %msg%n"
            ↪ />
        </File>
    </Appenders>
    <Loggers>
        <Root level="INFO" additivity="false">
            <AppenderRef ref="main" />
            <AppenderRef ref="console" />
        </Root>
    </Loggers>
</Configuration>
```

Listing 15: Apache Karaf log4j2 configuration

After that, Apache Karaf can be started via ./bin/start.

6.5 MQTT-Broker setup

The installation of the MQTT broker is straightforward. Just follow the steps below one after the other.

```
sudo apt update && sudo apt upgrade
sudo apt install -y mosquitto mosquitto-clients
sudo systemctl enable mosquitto.service
```

Listing 16: Eclipse Mosquitto setup

This should allow the Mosquitto MQTT broker to be installed on a Raspberry PI without any problems. In addition, however, the configuration must be adapted. For this purpose, the configuration must be adapted with the following command *sudo nano /etc/mosquitto/mosquitto.conf*.

```
# Place your local configuration in /etc/mosquitto/conf.d/
#
# A full description of the configuration file is at
# /usr/share/doc/mosquitto/examples/mosquitto.conf.example

pid_file /run/mosquitto/mosquitto.pid

persistence true
persistence_location /var/lib/mosquitto/

log_dest file /var/log/mosquitto/mosquitto.log

include_dir /etc/mosquitto/conf.d

listener 1883
allow_anonymous true
```

Listing 17: Eclipse Mosquitto configuration

After the adjustment, the system can be restarted, and if the start is successful, the broker should also be running on port 1883.

6.6 Xbee hardware setup

To use the Xbee hardware, it must first be configured via the XCTU tool. The devices must be connected to the PC via USB to do this. Then a scan can be started in the XCTU software. The configurations should be correct by default. After that, the selected device can be added to the software. When the device has been successfully added, it can be selected. In the further course, all configuration parameters are read from the device.

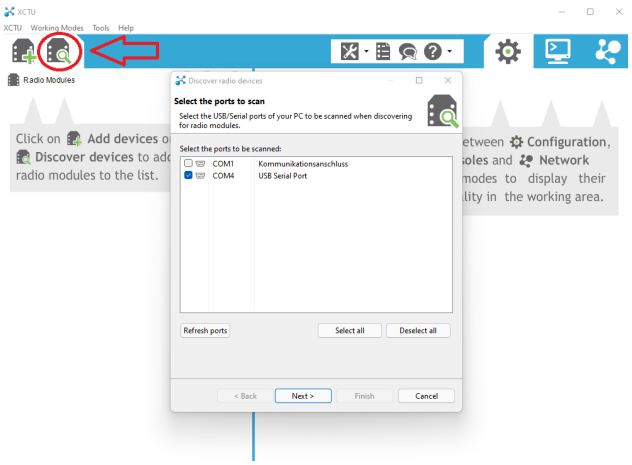


Figure 6.2: XCTU discover radio devices

In the following, the following parameters must be overwritten with the following values.

- CE Device Role: Based on type (Coordinator = Form Network [1], Router = Join Network [0])
- ID Extended PAN ID: C001BE (ID must be the same on all devices)
- SC Scan Channels: FFF
- AP API Enable: API Mode Without Escapes [1]

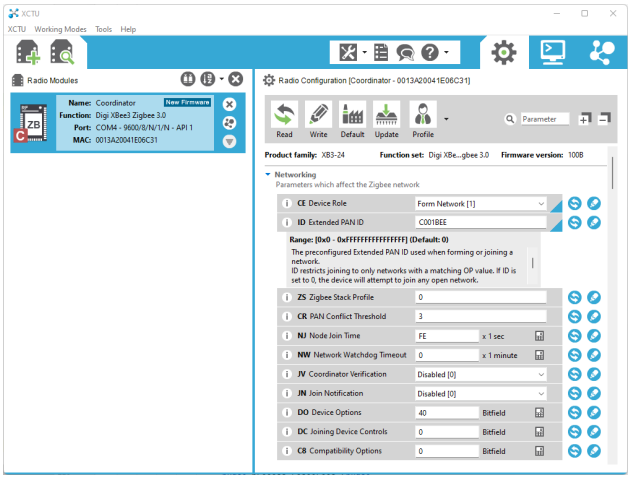


Figure 6.3: XCTU write to radio modules

If all parameters are set, they can be written to memory. For open questions, the documentation can help [Inc22b].

6.7 Amazon Web Services setup

Several steps are necessary for the gateway to connect to Amazon Web Services. On the one hand, the gateway needs rights to manage devices, and on the other hand, it must connect to AWS via MQTT. In addition, all messages should be saved to evaluate the results.

For the gateway to manage devices in AWS, it needs an access key. First, a new user must be created in AWS Identity and Access Management (IAM). Then, each user must be assigned specific rights to access certain resources in AWS. In the case of this prototype application development, the gateway user has been given administrator rights. However, these should not be used for actual deployment. After creation, one receives an access key id and a secret access key. These two values are needed to configure the gateway.

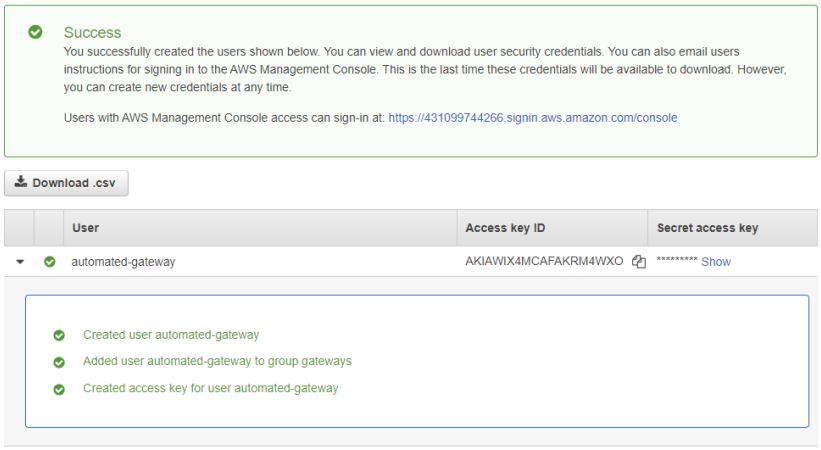


Figure 6.4: Created AWS user

In order to also communicate via MQTT, it must be created in the AWS IoT Core service. An additional certificate, group and policy must also be created for this. The certificate is required because AWS IoT Core necessarily enforces encryption via SSL. The group is used to create devices under this gateway so that they can be categorised. The policy specifies which MQTT endpoints the gateway can access. In the case of the implementation, all endpoints were selected.

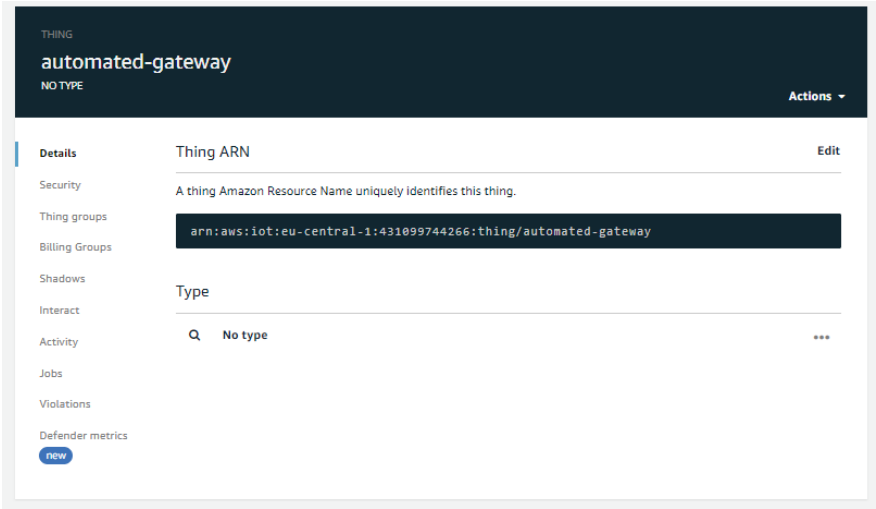


Figure 6.5: Created AWS thing

After everything has been created, the following items should be complete.

- certificate.pem.crt
- private.pem.key
- public.pem.key
- access key id
- secret access key
- gateway thing name
- gateway group arn

In addition, a permanent data store is needed for debugging the application. In this case, AWS DynamoDB is used for this purpose. For DynamoDB, only a simple table has to be created that stores the device name as the partition key and the time of receipt as the sort key.

In addition, the table can be declared as on-demand, as the resources are only used for short test periods, thus saving any costs.

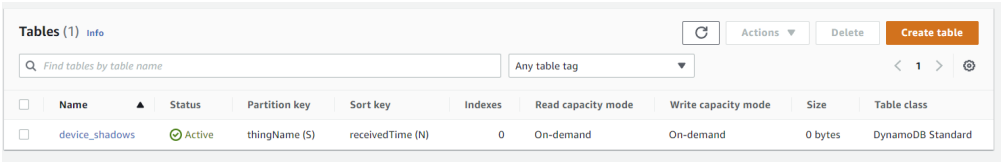


Figure 6.6: Created AWS DynamoDB table

A rule must be created in IoT Core so that the table is also filled with data. The rule can be seen in the following illustration 6.7. First, the instruction topic(3) returns the device name, then the time of receipt is documented with timeStamp(), and the sending time is also taken from the message. The values are then all stored in the table. The rule is always executed when an update message is received for a specific device.

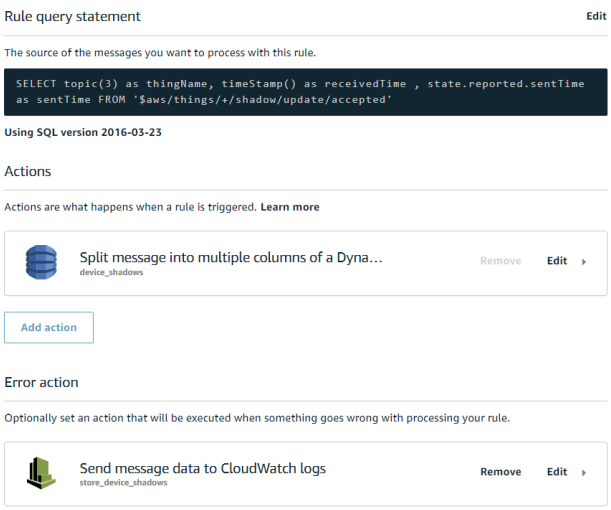


Figure 6.7: Created AWS IoT Core rule

6.8 Gateway setup

To install the gateway, Apache Karaf is also required. This must first be installed for the respective distribution. Then Apache Karaf can be started via the command line. Creating a remote session for Apache Karaf via the command line is also possible. The following is an example of this process displayed in Windows.

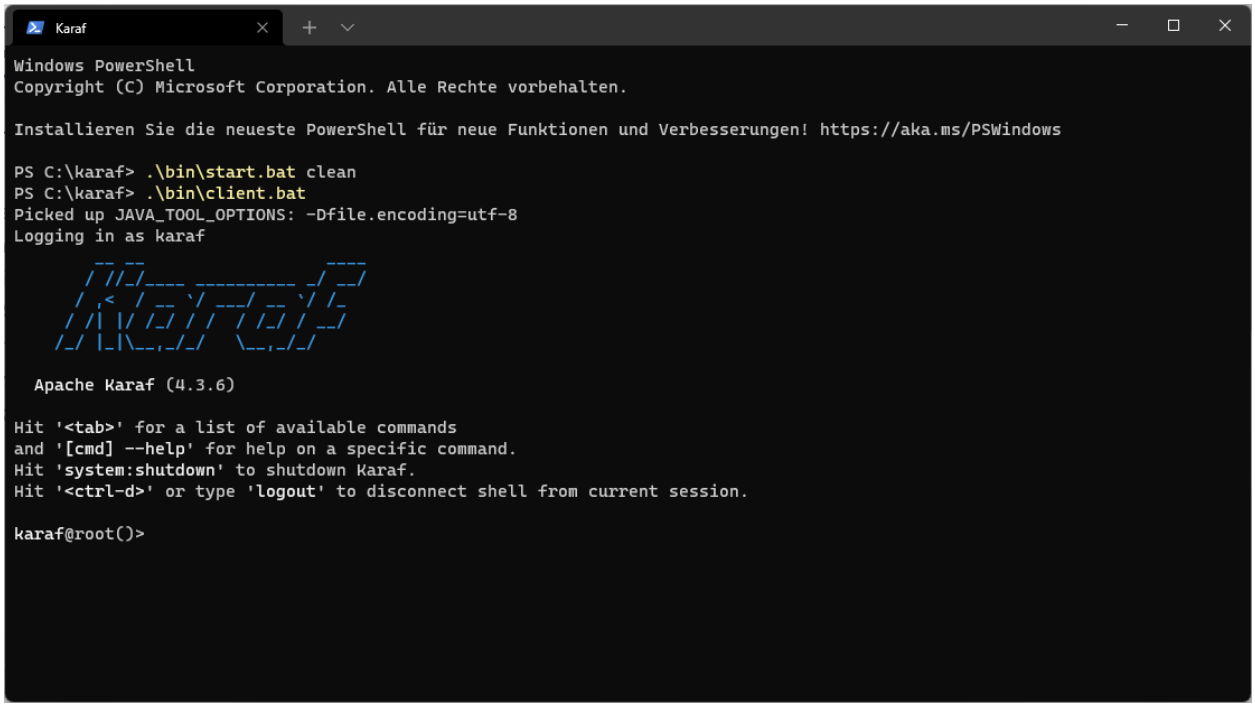


Figure 6.8: Starting apache karaf

As soon as Apache Karaf is started, the gateway can be installed in this OSGi environment. The gateway-impl artifact is provided as a feature, as all required dependencies are installed. The required artifacts could also be included in the artifact, but other adapters would not have access to them. For this reason, they must be installed in the OSGi container itself. The installation process for the feature is shown below.

```
feature:repo-add mvn:<groupId>/<artifactId>/<version>/xml/features
feature:install <artifactId>
...
feature:repo-add mvn:de.hsbremen.iot/gateway-impl/1.0-SNAPSHOT/xml/features
feature:install gateway-impl
```

Listing 18: Install gateway-impl command

6.8.1 Configuration

An error may occur here at the beginning, as the gateway expects the configuration file in "user.home/.gateway/config.yaml". If the configuration file does not exist, the gateway will create it and then shut down. The initial configuration file is empty. The format for the configuration file is to be defined in YAML. In addition, functionality has been added to mask values. Any value can be encoded in Base64, it only has to be enclosed in the brackets of the following expression *BASE64[...]*. This functionality is not intended for security purposes but only to prevent the reading of plaintext passwords in presentations.

The configuration parameters required for the gateway are shown below. In addition, the adapters will add parameters to the configuration file in the following sections.

```yaml
vertx:
  port: 8080 # The HTTP-Server port

mail:
  port: 587 # The SMTP port
  host: smtp-mail.outlook.com # The SMTP hostname or IP-address
  authEnabled: true # If auth is enabled
  tlsEnabled:  true # If TLS is enabled
  username: test@outlook.de # Username for SMTP server
  password: BASE64[dGVzdA==]# Password for username
  sendTo: test.outlook.de # Send mails to adress
  sendFrom: test.outlook.de # Send mails from adress

messaging:
  unzipIncomingMessages: true # If all incoming messages should be unzipped if possible
  messageInterceptorEnabled: true # Enables logging for incoming messages

monitoring:
  interval: 10000 # Monitoring interval in milliseconds
  primaryMemoryThreshold: 0.60 # Threshold to activate a registration filter
  secondaryMemoryThreshold: 0.80 # Threshold to activate a best effort filter
  tertiaryMemoryThreshold: 0.90 # Threshold to activate a high priority filter
  primaryCpuThreshold: 0.60 # Threshold to activate a registration filter
  secondaryCpuThreshold: 0.70 # Threshold to activate a best effort filter
  tertiaryCpuThreshold: 0.80 # Threshold to activate a high priority filter

# Defining adapters in a list
adapters:
  - adapterId: example-adapter # <artifactId> of adapter
    subscriber: # Defines the upstream default subscriber for messages that are not directed to
↪   devices.
      - example-adapter2 # <artifactId> of adapter
  - adapterId: example-adapter2
    subscriber:
      - [ ] # no subsribers is always [ ]
```

Listing 19: Example gateway configuration

6.9 Adapter setup

The process for the adapters is slightly different, as these are only individual bundles and not features. This means that they do not install any additional dependencies but include them. They are included here because no other components need these libraries or have to access them. In addition, different versions could be installed for the same libraries so that modularity is maintained.

```
bundle:install mvn:<groupId>/<artifactId>/<version>
bundle:start <id>
...
bundle:install mvn:de.hsbremen.iot/mqtt-adapter/1.0-SNAPSHOT
bundle:start mqtt-adapter
bundle:install mvn:de.hsbremen.iot/aws-mqtt-adapter/1.0-SNAPSHOT
bundle:start aws-mqtt-adapter
bundle:install mvn:de.hsbremen.iot/websocket-adapter/1.0-SNAPSHOT
bundle:start websocket-adapter
bundle:install mvn:de.hsbremen.iot/xbee-adapter/1.0-SNAPSHOT
bundle:start xbee-adapter
```

Listing 20: Install adapter commands

6.9.1 Configuration of MQTT

The configuration file must be extended to add the *mqtt-adapter* to the gateway. The required entries are shown below.

```
...
# Defining adapters in a list
adapters:
  - adapterId: mqtt-adapter
    subscriber:
      - example-adapter2
  - adapterId: example-adapter2
    subscriber:
      - [ ]

mqtt-adapter:
  serverUri: "tcp://192.168.0.70:1883" # IP-address of an MQTT broker
  cleanSession: true # If a new session should always be a clean session
  automaticReconnect: true # If automatic reconnect is enabled
```

Listing 21: Example mqtt-adapter configuration

6.9.2 Configuration of ZigBee

The configuration file must also be extended to add the *xbee-adapter*. These configurations are also shown below.

```
...

# Defining adapters in a list
adapters:
  - adapterId: xbee-adapter
    subscriber:
      - example-adapter2
  - adapterId: example-adapter2
    subscriber:
      - [ ]

xbee-adapter:
  port: "COM4" # Defines the USB-port which is connected to the xbee hardware
  baudRate: 9600 # Defines the baud rate
```

Listing 22: Example xbee-adapter configuration

6.9.3 Configuration of AWS

For the *aws-mqtt-adapter*, considerably more configurations must be stored, as this adapter requires access to AWS. These configurations are also shown below. Where the parameters come from is explained in the section for the AWS implementation 6.7 on page 105.

```
...

# Defining adapters in a list
adapters:
  - adapterId: aws-mqtt-adapter
    subscriber:
      - [ ] # No upstream subscribers for AWS, because AWS is the upstream subscribers itself.

aws-mqtt-adapter:
  mqttConnection:
    clientId: "DevGateway" # Gateway thingName in AWS
    endpoint: "a2sn3h9hqsahkf-ats.iot.eu-central-1.amazonaws.com" # AWS specific IoT Core
↪   endpoint
    certPath: "C:\\test\\.gateway\\certificate.pem.crt" # Absolute path to AWS certificate
    keyPath: "C:\\Users\\test\\.gateway\\private.pem.key" # Absolute path to AWS private key
    deviceShadowEnabled: true # Defines if AWS defined communication model should be used for
↪   upstream messages. Otherwise the gateway communication model is used.
  registrationHandler:
    region: "eu-central-1" # AWS region of application
    groupArn: "arn:aws:iot:eu-central-1:431099744266:thinggroup/DevGateway" # AWS groupArn for
↪   the gateway
    accessKeyId: "BASE64[dGVzdA==]" # AWS access key id
    secretAccessKey: "BASE64[dGVzdA==]" # AWS secret access key
```

Listing 23: Example aws-mqtt-adapter configuration

6.9.4 Configuration of WebSocket

The *websocket-adapter* does not require any further configuration. It only has to appear in the configured adapters.

```
...

# Defining adapters in a list
adapters:
  - adapterId: websocket-adapter
    subscriber:
      - example-adapter2
  - adapterId: example-adapter2
    subscriber:
      - [ ]
```

Listing 24: Example websocket-adapter configuration

6.10 DebugClient setup

To test the system, clients are needed to connect to it. However, since it is not a goal to realize a compliant client for each type of protocol, only a basic framework for testing the gateway was implemented. Furthermore, since the implementation is also not result-relevant, the implementation details are omitted and only describe how these can be used.

To start the client, just enter *java -jar <artifactId>.jar* in the command line.

However, starting the client still requires a configuration file. The configuration file must be located next to the exported jar. It is defined in YAML format and is called *config.yaml*. The following default entries are valid for all clients, but the individual clients will then extend them based on their different protocols.

```
debugDevice:
  deviceId: "testClient" #deviceId in Cloud, if now known use an new one
  maxQoS: 9 # maximum QoS level defined as int
  currentQoS: 9 # current QoS level defined as int
  privileged: false # if the device is privileged
  highPriority: false # if normal message should be high priority

debugClient:
  legacy: false # if gateway communication or simple messages should be used
  lowBandwidth: false # if device is in a low bandwidth environment
  initialDelay: 1000 # initial delay before sending the first message
  reportInterval: 100 # reporting interval in ms
```

Listing 25: Example debug-client configuration

6.10.1 DebugMqttClient configuration

To use the *debug-mqtt-client*, the configuration file must be extended. The needed configurations are shown below.

```
...
mqtt:
  serverUri: "tcp://192.168.0.70:1883" # Domaine name or IP-address of the mqtt broker
  clientId: "testClientForAwsRegistration1" # Client id for the mqtt-client, should be the same
↪  as deviceId
  gatewayHostname: "testGateway" # Hostname of the gateway
```

Listing 26: Example debug-mqtt-client configuration

6.10.2 DebugXbeeClient configuration

To use the *debug-xbee-client*, the configuration file must be extended. The needed configurations are shown below.

```
...
xbee:
  port: "COM5" # USB port
  baudRate: 9600 # Baud rate
  coordinator: "Coordinator" # name of the gateway xbee device
```

Listing 27: Example debug-xbee-client configuration

6.10.3 DebugWebSocketClient configuration

To use the *debug-websocket-client*, the configuration file must be extended. The needed configurations are shown below.

```
...
websocket:
  host: "localhost" # Websocket hostname or IP-address
  port: 8080 # HTTP server port
  uri: "/websockets?deviceId=testClient12" # URL for websocket access including the deviceId
```

Listing 28: Example debug-websocket-client configuration

Chapter 7

Verification

The following chapter deals with the verification of the defined requirements (see also section 4.2.3 on page 59 above). The requirements are divided into the different verification methods used and verified. The verification methods only prove whether the gateway complies with the specification. For error detection, additional unit tests were added to the maven project.

7.1 Test

This section deals with all requirements that have been marked with the verification method **Test**. The tests are so-called integration tests. Module tests were also carried out and can be found as JUnit tests in the Java project itself.

The following table 7.1 shows all the tests carried out and refers to the requirements and the actual test. Several times, different requirements could be proven together to combine them to avoid duplicates.

Requirement	Reference	Complies	Notes
A2		✓	
A5.3		-	Can not be verified, as this is an unused feature.
A6, A6.1, A6.2, A6.3, A6.4, A6.5, A10.1		✓	
A7, A7.1, A7.2		✓	
A5, A5.1, A5.2, A8, A8.1		✓	
A8.2		✓	
A9, A9.1, A9.3, A9.4, A9.5		✓	
A9.4		✓	
B1.x - B3.x, C1, C1.1, C1.2, C1.3, C1.4, C2, C2.1, C2.2, C2.3, C2.4		✓	
B4.1, B4.2, B4.3, B4.4, C1.5		✓	

Table 7.1: Test verified requirements

Based on the table, almost all requirements were verified. Only requirement **A5.3** could not be verified, as this functionality was not needed in the end.

7.1.1 Configuration

This test scenario tests the configuration of the system. Therefore the requirement **A2** can be verified.

Instructions:

1. Start the gateway locally.
2. Compare the logged configuration properties with the config.yaml file.

Expected result:

1. The logged configuration properties match the config.yaml file.

Since the configuration file matches the logged content, the **A2** request can be considered verified. However, the match has not been listed here because the complete example would use too much space.

7.1.2 Monitoring

This test scenario tests the monitoring of the system. The requirements **A6**, **A6.1**, **A6.2**, **A6.3**, **A6.4**, **A6.5**, and **A10.1** can be verified together.

Preconditions:

1. Gateway is started locally.
2. AWS-MQTT-Adapter is initialised.
3. MQTT-Adapter is initialised.
4. WebSocket-Adapter is initialised.
5. ZigBee-Adapter is initialised.
6. debug-mqtt-client is running.
7. debug-xbee-client is running.
8. debug-websocket-client is running.

Instructions:

1. Access "http://localhost:8080/gateway/resources" to review resource measurements.

Expected result:

1. The gateway returns a JSON with the measured resources. The received and sent messages should match based on the report interval.


```
// 20220408125430
// http://localhost:8080/gateway/resources

{
  "interval": 10000,
  "systemTime": 1649415268623,
  "cpuUsageTotal": 0.14397596030849158,
  "cpuUsageProcess": 0.0021533037295654077,
  "heapMemoryUsage": {
    "init": 16777216,
```

```
    "used": 73039616,
    "committed": 80252928,
    "max": 259522560
  },
  "nonHeapMemoryUsage": {
    "init": 163840,
    "used": 61961552,
    "committed": 63930368,
    "max": -1
  },
  "messageServiceUsage": {
    "queueSize": 0,
    "sentMessageCount": 400,
    "receivedMessageCount": 400
  },
  "adapterUsages": [
    {
      "adapterId": "websocket-adapter",
      "sentQueueSize": 0,
      "sentMessageCount": 0,
      "receivedMessageCount": 100
    },
    {
      "adapterId": "xbee-adapter",
      "sentQueueSize": 0,
      "sentMessageCount": 0,
      "receivedMessageCount": 100
    },
    {
      "adapterId": "mqtt-adapter",
      "sentQueueSize": 0,
      "sentMessageCount": 0,
      "receivedMessageCount": 100
    },
    {
      "adapterId": "aws-mqtt-adapter",
      "sentQueueSize": 0,
      "sentMessageCount": 400,
      "receivedMessageCount": 0
    }
  ]
}
```


Listing 29: Response of resource measurement endpoint

With the correctly received JSON document, requirements **A6**, **A6.1**, **A6.2**, **A6.3**, **A6.4**, **A6.5**, **A10.1** can be considered verified.

7.1.3 Resource scheduling

This test scenario tests the resource scheduling of the system. The requirements **A7**, **A7.1** and **A7.2** can be verified together.

Preconditions:

1. Gateway is started locally
2. MQTT-Adapter is initialised

Instructions:

1. Start a CPU stress test and observe log entries (CPU-Z was used in this test scenario).
2. Wait until RegistrationFilter and BestEffort filter is activated.
3. Start a debug-mqtt-client with BEST_EFFORT priority.
4. Restart the debug-mqtt-client with HIGH_PRIORITY priority.

Expected result:

1. The gateway should report that a high resource consumption was detected.
2. The BEST_EFFORT message should be denied.
3. The HIGH_PRIORITY message should be treated normally.

```
[INFO ] 2022-04-08 13:43:18.207 [pool-9-thread-1] ResourceSchedulerImpl - High resource
↪  consumption detected!
[INFO ] 2022-04-08 13:43:18.209 [pool-9-thread-1] ResourceSchedulerImpl - Activating
↪  RegistrationFilter, no more registrations allowed!
[INFO ] 2022-04-08 13:43:18.209 [pool-9-thread-1] ResourceSchedulerImpl - Activating BestEffort,
↪  no more best effort messages allowed!
[INFO ] 2022-04-08 13:54:24.811 [MQTT Call: Marvin-Desktop] MessageServiceImpl - Incoming message:
↪  Message(header=Header(fromService=mqtt-adapter, type=UPDATE, priority=BEST_EFFORT,
↪  compressed=false, from=testClientForAwsRegistration1, to=null), payload=[123, 34, 115, 116,
↪  97, 116, 101, 34, 58, 123, 34, 114, 101, 112, 111, 114, 116, 101, 100, 34, 58, 123, 34, 115,
↪  101, 110, 116, 84, 105, 109, 101, 34, 58, 49, 54, 52, 57, 52, 50, 54, 48, 54, 52, 125, 125,
↪  125])
[INFO ] 2022-04-08 13:43:36.370 [MQTT Call: Marvin-Desktop] MessageFilterImpl - Denied message:
↪  Registration filter is enabled!

...
[INFO ] 2022-04-08 13:54:24.811 [MQTT Call: Marvin-Desktop] MessageServiceImpl - Incoming message:
↪  Message(header=Header(fromService=mqtt-adapter, type=UPDATE, priority=HIGH_PRIORITY,
↪  compressed=false, from=testClientForAwsRegistration1, to=null), payload=[123, 34, 115, 116,
↪  97, 116, 101, 34, 58, 123, 34, 114, 101, 112, 111, 114, 116, 101, 100, 34, 58, 123, 34, 115,
↪  101, 110, 116, 84, 105, 109, 101, 34, 58, 49, 54, 52, 57, 52, 50, 54, 48, 54, 52, 125, 125,
↪  125])
```

Listing 30: Log entries of the resource scheduling test scenario

According to the log, the requirements **A7**, **A7.1** and **A7.2** can be considered verified.

7.1.4 Connection management

This test scenario tests the connection management of the system. The requirements **A8** and **A8.1** can be verified together. For requirement **8.2**, there is an additional test scenario in the following. The requirements **A5**, **A5.1**, and **A5.2** can be verified because the log entry includes the saved metadata.

Preconditions:

1. Gateway is started locally.

Instructions:

1. Start an unregistered debug-mqtt-client in non legacy mode.
2. Start an unregistered debug-mqtt-client in legacy mode.

Expected result:

1. The gateway should report that a new device was created on a connection attempt in both cases. This also means that the gateway detects if a legacy client already exists in his local database.

```
[INFO ] 2022-04-08 14:18:33.257 [MQTT Call: Marvin-Desktop] MessageServiceImpl - Incoming message:
↪  Message(header=Header(fromService=mqtt-adapter, type=CONNECT, priority=HIGH_PRIORITY,
↪  compressed=false, from=testClientForAwsRegistration1, to=null), payload=[123, 34, 109, 97,
↪  120, 81, 111, 115, 34, 58, 57, 44, 34, 99, 117, 114, 114, 101, 110, 116, 81, 111, 115, 34, 58,
↪  57, 44, 34, 112, 114, 105, 118, 105, 108, 101, 103, 101, 100, 34, 58, 102, 97, 108, 115, 101,
↪  125])
[INFO ] 2022-04-08 14:18:33.263 [ForkJoinPool.commonPool-worker-19] DeviceFactoryImpl - Created
↪  device from message Message(header=Header(fromService=mqtt-adapter, type=CONNECT,
↪  priority=HIGH_PRIORITY, compressed=false, from=testClientForAwsRegistration1, to=null),
↪  payload=[123, 34, 109, 97, 120, 81, 111, 115, 34, 58, 57, 44, 34, 99, 117, 114, 114, 101, 110,
↪  116, 81, 111, 115, 34, 58, 57, 44, 34, 112, 114, 105, 118, 105, 108, 101, 103, 101, 100, 34,
↪  58, 102, 97, 108, 115, 101, 125])
[INFO ] 2022-04-09 16:58:57.875 [ForkJoinPool.commonPool-worker-19] NitriteDeviceRepository -
↪  Saved device Device(deviceId=testClientForAwsRegistration1, state=CONNECTED, maxQos=9,
↪  currentQos=9, privileged=false, adapterId=mqtt-adapter, adapterInterfaces={}) to local
↪  database
[INFO ] 2022-04-08 14:18:33.300 [ForkJoinPool.commonPool-worker-19] MessageServiceImpl - Incoming
↪  message: Message(header=Header(fromService=DeviceService, type=CONNECT_ACK,
↪  priority=HIGH_PRIORITY, compressed=false, from=DeviceService,
↪  to=testClientForAwsRegistration1), payload=null)

...
[INFO ] 2022-04-08 14:18:52.449 [ForkJoinPool.commonPool-worker-19] MessageServiceImpl - Incoming
↪  message: Message(header=Header(fromService=DeviceService, type=DISCONNECT_ACK,
↪  priority=HIGH_PRIORITY, compressed=false, from=DeviceService,
↪  to=testClientForAwsRegistration1), payload=null)
[INFO ] 2022-04-08 14:21:23.577 [MQTT Call: Marvin-Desktop] MessageServiceImpl - Incoming message:
↪  Message(header=Header(fromService=mqtt-adapter, type=LEGACY_CONNECT, priority=HIGH_PRIORITY,
↪  compressed=false, from=testClientForAwsRegistration3, to=null), payload=null)
[INFO ] 2022-04-08 14:21:23.578 [ForkJoinPool.commonPool-worker-5] DeviceFactoryImpl - Created
↪  device from message Message(header=Header(fromService=mqtt-adapter, type=LEGACY_CONNECT,
↪  priority=HIGH_PRIORITY, compressed=false, from=testClientForAwsRegistration3, to=null),
↪  payload=null)
```

Listing 31: Log entries of the connection management test scenario

According to the log, the requirements **A8** and **A8.1** can be considered verified. Also, the saved metadata is included. Therefore the requirements **A5**, **A5.1**, and **A5.2** can also be verified.

7.1.5 Quality of Service management

This test scenario tests the quality of service management of the system. The quality of service management is a part of the connection management, but it must be tested separately. For this reason, requirement **8.2** is verified with this test.

Preconditions:

1. The Gateway is started locally.
2. The mqtt-adapter is initialised.
3. A debug-mqtt-client is running.

Instructions:

1. Start a CPU stress test and observe log entries (CPU-Z was used in this test scenario).

Expected result:

1. The gateway should report that a high resource consumption was detected.
2. The gateway should send a QOS_REDUCE message to available clients.

```
[INFO ] 2022-04-08 15:11:11.162 [pool-8-thread-1] ResourceSchedulerImpl - High resource
↪  consumption detected!
[INFO ] 2022-04-08 15:11:12.138 [pool-8-thread-1] MessageServiceImpl - Incoming message:
↪  Message(header=Header(fromService=ResourceService, type=QOS_REDUCE, priority=HIGH_PRIORITY,
↪  compressed=false, from=Marvin-Desktop, to=testClient12), payload=null)
```

Listing 32: Log entries of the QoS test scenario

According to the log, the requirement **A8.2** can also be considered verified.

7.1.6 Exception handling

This test scenario tests the exception handling of the system. As a result, the requirements **A9**, **A9.1**, **A9.3**, **A9.4**, and **A9.5** can be verified. In order to verify requirement **A9.2**, another scenario must additionally be carried out.

Preconditions:

1. The gateway is started locally.

Instructions:

1. Start the mqtt-adapter with a faulty configuration file.

Configuration:

Expected result:

1. The gateway should report that there was an service startup exception.
2. The gateway should send an email to the configured administrator.
3. The gateway should shut down the adapter.

```
[ERROR] 2022-04-09 13:47:04.244 [pipe-bundle:start 88] ExceptionServiceImpl -
↪  de.hsbremen.iot.gateway.api.exception.ServiceStartupException:
↪  de.hsbremen.iot.gateway.api.exception.ConfigCheckException: ConfigCheck failed on
↪  'MqttAdapterConfig' because field 'serverUri' is null!
...
Caused by: de.hsbremen.iot.gateway.api.exception.ConfigCheckException: ConfigCheck failed on
↪  'MqttAdapterConfig' because field 'serverUri' is null!
...
[INFO ] 2022-04-09 13:47:05.785 [pipe-bundle:start 88] MqttHandler - MqttHandler successfully
↪  shut down!
```

Listing 33: Truncated log entries of exception handling test scenario

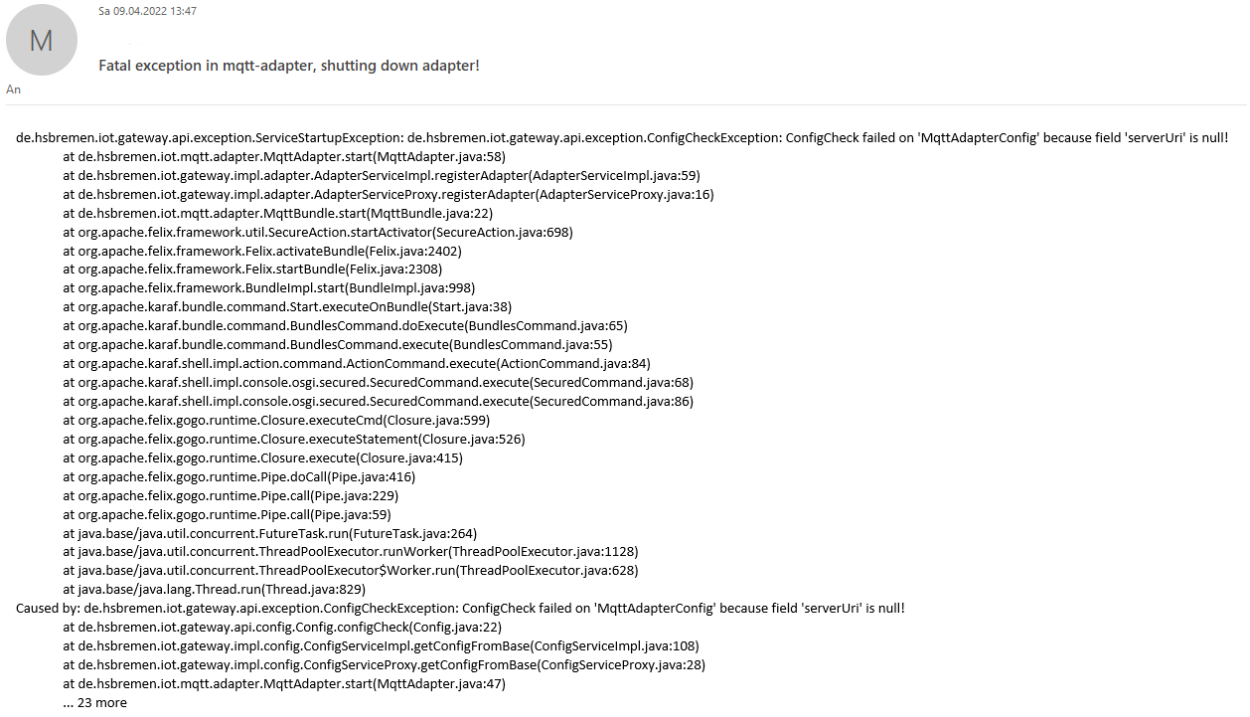


Figure 7.1: Received notification of the exception handling test scenario

Based on the email received, the log entered, and the adapter shut down, requirements **A9**, **A9.1**, **A9.3**, **A9.4**, and **A9.5** can be considered verified.

7.1.7 Fatal exception handling

In addition to the previously mentioned requirements for exception handling, requirement **A9.4** must be tested separately.

Configurations:
The required parameters must be commented out of the configuration file to run this test scenario. In this case, the parameters for Vert.X were commented out.

```
#vertx:
#  port: 8080
...
```

Listing 34: Fatal exception handling configuration changes

Instructions:

1. Start the gateway with a faulty configuration file.

Expected result:

1. The gateway should report that there was an service startup exception.
2. The gateway should shut down itself.

```
[ERROR] 2022-04-09 14:10:38.139 [features-3-thread-1] ExceptionServiceImpl -
↪  de.hsbremen.iot.gateway.api.exception.ServiceStartupException: The ConfigService could not be
↪  started!
...
[ERROR] 2022-04-09 14:10:38.161 [features-3-thread-1] ExceptionServiceImpl - Can not recover from
↪  stacktrace, shutting down gateway!
```

Listing 35: Truncated log entries of the fatal exception handling test scenario

Based on the log entries and the fact that the gateway is not running anymore, the requirement **A9.4** can also be seen as verified.

7.1.8 Adapter

Since the mqtt-adapter, xbee-adapter and websocket-adapter have the exact requirements, they can be tested together. This means that all requirements from the range **B1.x** - **B3.x** can be tested with this test scenario. Additionally, this scenario can also verify the requirements **C1**, **C1.1**, **C1.2**, **C1.3**, **C1.4**, **C2**, **C2.1**, **C2.2**, **C2.3** and **C2.4** because we are using the implemented clients to verify this behaviour. The following scenario explains the procedure as an example for the mqtt-adapter. The settings are the same for all adapters except for the xbee-adapter, where the low bandwidth must also be activated.

Preconditions:

1. The Gateway is started locally.
2. The associated adapter is initialised.

Configurations:
To run this test scenario, one parameter is required. Therefore the roundTrip settings should be enabled.

```
...
debugClient:
  legacy: false
  lowBandwidth: false # true for ZigBee
  roundTrip: true
  initialDelay: 1000
  reportInterval: 1000
```

Listing 36: Adapter configuration changes

Instructions:

1. Start the associated client.

Expected result:

1. As the client saves messages to a human-readable format, the entries of sent.csv and received.csv can be compared.

```
Priority,From,To,SentTime,ReceivedTime,TimeInBetweenMs
BEST_EFFORT,testClient,testClient,1649522100747,-
BEST_EFFORT,testClient,testClient,1649522101745,-
BEST_EFFORT,testClient,testClient,1649522102744,-
BEST_EFFORT,testClient,testClient,1649522103736,-
BEST_EFFORT,testClient,testClient,1649522104743,-
BEST_EFFORT,testClient,testClient,1649522105747,-
BEST_EFFORT,testClient,testClient,1649522106741,-
BEST_EFFORT,testClient,testClient,1649522107750,-
```

Listing 37: Sent messages of the adapter test scenario

```
Priority,From,To,SentTime,ReceivedTime,TimeInBetweenMs
BEST_EFFORT,testClient,testClient,1649522100747,1649522100772,25
BEST_EFFORT,testClient,testClient,1649522101745,1649522101765,20
BEST_EFFORT,testClient,testClient,1649522102744,1649522102766,22
BEST_EFFORT,testClient,testClient,1649522103736,1649522103757,21
BEST_EFFORT,testClient,testClient,1649522104743,1649522104765,22
BEST_EFFORT,testClient,testClient,1649522105747,1649522105768,21
BEST_EFFORT,testClient,testClient,1649522106741,1649522106763,22
BEST_EFFORT,testClient,testClient,1649522107750,1649522107774,24
```

Listing 38: Received messages of the adapter test scenario

Since the messages look the same, it can be verified that the gateway receives the messages, extracts them and sends them back to the correct client. For this reason, the above requirements can be considered verified.

7.1.9 AWS-MQTT-Adapter

The AWS MQTT adapter requires additional testing, as we do not have a direct client that sends and receives data. Therefore, the following test scenario can be used to verify the requirements **B4.1**, **B4.2**, **B4.3**, and **B4.4**. Additionally, the requirement **C1.5** can also be verified with this scenario, as we verify that Amazon Web Services received the testing messages.

Preconditions:

1. The Gateway is started locally.
2. The aws-mqtt-adapter is initialised.
3. Any possible adapter is initialised (in this case the mqtt-adapter was used).

Instructions:

1. Start the associated client (in this case mqtt-client).
2. Compare the created sent.csv with entries in DynamoDB.

Expected result:

1. The client reported sent messages in sent.csv should match the entries in DynamoDB.

```
Priority,From,To,SentTime,ReceivedTime,TimeInBetweenMs
BEST_EFFORT,testClient,null,1649520759424,-
BEST_EFFORT,testClient,null,1649520760416,-
BEST_EFFORT,testClient,null,1649520761423,-
```

Listing 39: Sent messages of the aws-mqtt-adapter test scenario

```
thignName, receivedTime, sentTime
testClient,1649513558911,1649520759424
testClient,1649513559908,1649520760416
testClient,1649513560929,1649520761423
```

Listing 40: Received messages of the aws-mqtt-adapter test scenario

Since the data in the payload has been correctly processed, this means that the message has been correctly received, transformed and sent by the adapter. For this reason, the requests can be considered verified.

7.1.10 AWS-MQTT-Adapter registration

The aws-mqtt-adapter has an additional requirement. This deals with the registration of devices in AWS itself. Therefore, the following test scenario deals with requirement **B4.5**.

Preconditions:

1. The Gateway is started locally.
2. The aws-mqtt-adapter is initialised.
3. Any possible adapter is initialised (in this case the mqtt-adapter was used).

Instructions:

1. Start the associated client (in this case mqtt-client) with a new device identifier.

Expected result:

1. The device is registered in AWS IoT Core

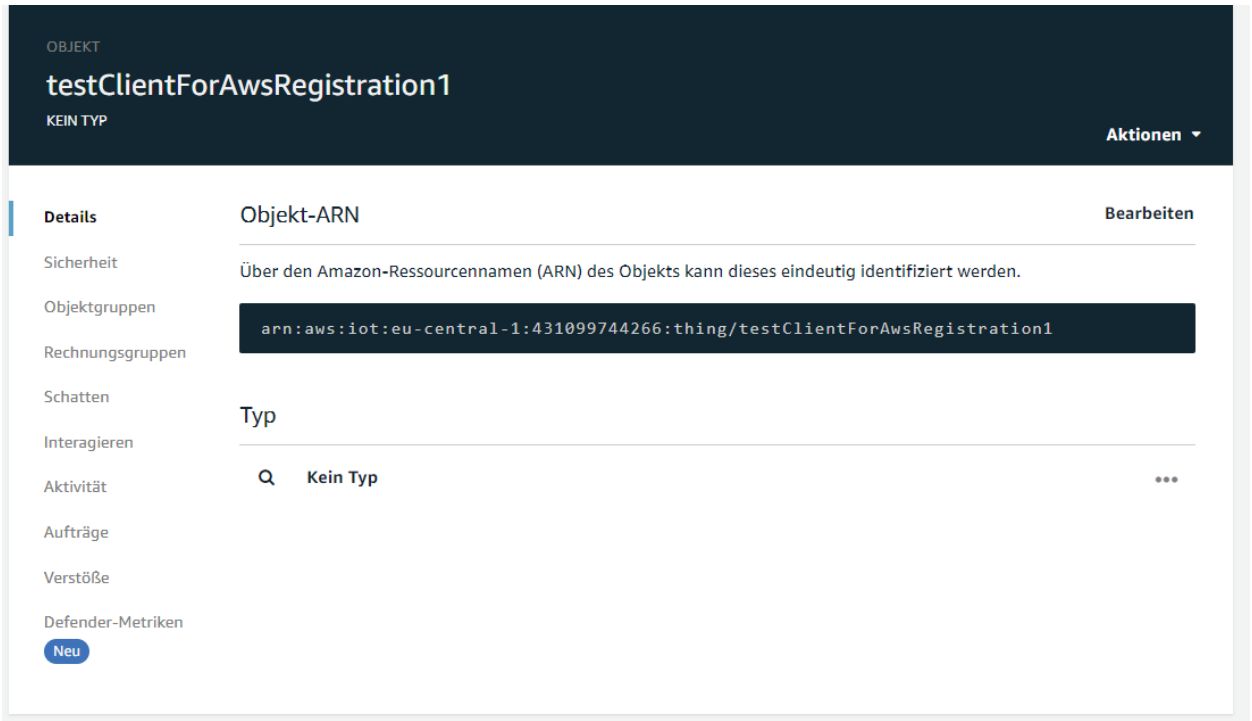


Figure 7.2: Automatic registered device of the aws-mqtt-adapter registration test scenario

Since the device has been automatically registered in Amazon Web Service, this requirement can also be considered verified.

7.2 Inspection

The following requirements were evidenced by an inspection of the relevant documents or source code.

Requirement	Reference	Complies	Notes
A1, A1.1, A1.2, A1.3	7.2.1	✓	
A3.3		✓	

Table 7.2: Inspection verified requirements

This means that all specified requirements, which were marked with Inspection, have been verified.

7.2.1 Inspection of GatewayConfig

By looking at the source code more specifically, by looking at the GatewayConfig.java and ConfigServiceImpl.java file, the requirements **A1, A1.1, A1.2 and A1.3** can be verified. For example, the following listing shows that all specified configurations are present and loaded.

```
@Getter
@Setter
public class GatewayConfig extends Config {

    private MailConfig mail;

    private VertxConfig vertx;

    private List<AdapterConfig> adapters;

    private MessageServiceConfig messaging;

    private ResourceServiceConfig monitoring;

    ...
}

public class ConfigServiceImpl implements InternalConfigService {
...
    @Override
    public void start() {
        try {
            Path configPath = Paths.get(CONFIG_PATH);
            if (Files.notExists(configPath)) {
                if (Files.notExists(configPath.getParent()))
                    Files.createDirectory(configPath.getParent());
                Files.createFile(configPath);
                throw new ConfigCheckException("ConfigCheck failed because path '%s' is unknown,
                ↪  please create config first!", configPath);
            }
            this.configAsString = Files.readString(configPath);
            if (this.configAsString.isEmpty()) {
                throw new ConfigCheckException("ConfigCheck failed because config.yaml is empty,
                ↪  please fill out config first!");
            }
            this.config = this.getConfig(GatewayConfig.class);
            logger.info("Loaded config: {} {}", System.lineSeparator(), this.configAsString);
            logger.info("ConfigService successfully started!");
        } catch (Exception e) {
            this.serviceRegistry.exceptionService()
                    .handleException(new ServiceStartupException("The ConfigService could not be
                    ↪  started!", e, this));
        }
```

```
    }

        @Override
    public <T extends Config> T getConfig(Class<T> tClass) throws ConfigException {
        Yaml yaml = new Yaml(new CustomClassLoaderConstructor(tClass.getClassLoader()),
        ↪  this.representer);
        T t = yaml.loadAs(this.configAsString, tClass);
        t.configCheck();
        t.postConstruct();
        return t;
    }
...
}
```

Listing 41: Clip of the GatewayConfig.java and MailConfig.java

7.2.2 Inspection of XbeeAdapterConfig

The source code can also be viewed to check how the individual adapters are configured. The configuration interface is reused and combined with an independent configuration class. This allows the YAML file to be read into the format of this new class. The following listing also shows the verification of requirement **A3.3**.

```
@Getter
@Setter
@ToString
public class XbeeAdapterConfig extends Config {

    private String port;

    private int baudRate;

}

public class XbeeAdapter implements Adapter {
        ...
    @Override
    public void start() {
        try {
            ConfigService configService = this.serviceRegistry.configService();
            XbeeAdapterConfig xbeeAdapterConfig = configService.getConfigFromBase(ID,
            ↪  XbeeAdapterConfig.class);
            this.xBeeDevice = new XBeeDevice(xbeeAdapterConfig.getPort(),
            ↪  xbeeAdapterConfig.getBaudRate());
            this.xBeeDevice.open();
            this.xbeeMessageListener.start();
            this.xbeeMessageHandler.start();
        } catch (Exception ex) {
            this.serviceRegistry
                    .exceptionService()
                    .handleException(new ServiceStartupException(ex, this));
        }
    }
    ...
}

public class ConfigServiceImpl implements InternalConfigService {
        ...
    @Override
    public <T extends Config> T getConfigFromBase(String identifier, Class<T> tClass) throws
    ↪  ConfigException {
        Yaml yaml = new Yaml(new CustomClassLoaderConstructor(tClass.getClassLoader()),
        ↪  this.representer);
```

```
        Map<String, Object> configAsMap = yaml.load(this.configAsString);
        T t = yaml.loadAs(yaml.dump(configAsMap.get(identifier)), tClass);
        t.configCheck();
        t.postConstruct();
        return t;
    }
    ...
}
```

Listing 42: Clip of the ConfigServiceImpl.java, XbeeAdapter.java and XbeeAdapterConfig.java

7.3 Review of Design

The following requirements were attempted to be evidenced by inspection of the referenced documentation. Accordingly, all evidenced requirements have been ticked to indicate this.

Requirement	Documentation reference	Complies	Notes
A3, A3.1, A3.2, A4, A4.1, A4.2, A4.3, A4.4, A4.5	[Fou22c]	✓	Provided by Apache Karaf
A10	5.6 on page 87	✓	Provided by ResourceService-HttpHandler
B1, B2, B3, B4	5.7 on page 89	✓	Provided by MQTT, ZigBee, WebSocket and AWS-MQTT adapters
C1.5	[Cor22a]	✓	Provided by AWS IoT Core

Table 7.3: Review of Design verified requirements

This means that all specified requirements, which were marked with Review of Design, have been verified.

Chapter 8

Evaluation

The following chapter evaluates the results achieved against the defined objectives. In addition, the results are classified in the current state of research. Finally, an explanation is given for each objective of how it was achieved.

8.1 Protocol conversion

Protocol conversion is one of the biggest problems in IoT, as there is no standardised interface for communication due to heterogeneity. Therefore, many manufacturers use their standards or fall back on one of the many freely available standards.

To solve this problem, the thesis is based on three scientific publications mentioned in the chapters 2.1.3 on page 12, 2.1.4 on page 13 and 2.1.5 on page 14.

First, a uniform interface for protocols is made available. Separate OSGi bundles can implement this interface. The implemented bundle can now be installed subsequently to the existing gateway and thus enables communication via a new protocol. This makes the actual gateway independent of the protocols used and can be easily extended shown in figure 5.2 on page 70.

This approach alone would be sufficient to send messages only to the cloud. However, this does not make more complex communication processes possible. For this purpose, the approach was supplemented by another, in that the messages are sent through a fixed defined format shown in figure 5.6 on page 77. This format now enables addressing individual devices and the definition of cross-system methods or even the prioritisation of these. In addition, the data format is not necessary, so incompatible devices can still send messages to the cloud. Unfortunately, however, more complex message flows are then not possible.

These two exemplary approaches can show how the problem of protocol conversion can be solved. However, the data format used so far is still a very rudimentary and not fully developed version. In addition, it must be noted that points such as access security have not yet been taken into account, so this problem must also be solved. Nevertheless, this approach shows how one could solve the problems through a firmly defined interface. This means that a general IoT standard could solve the heterogeneity of the protocols.

8.2 Device management

Device management is a topic that increases in complexity with growing depth. For this reason, the proposed approach only solves the problem fundamentally.

In the proposed approach, each gateway uses a database of all connected devices. The gateway can detect whether devices have already been registered or not from this database. When an unregistered device connects to the gateway, a registration process is automatically performed. This process can then notify the various adapters of the new device so that the adapters are responsible for registering the device in subsequent systems.

This simple approach allows devices to be registered in the cloud without much effort, as the adapter does all the work. This means that there is no need for configuration in the cloud itself. However, this approach also brings with it a significant problem, as no precise security mechanisms have been worked out. Therefore, providing this approach with security mechanisms is essential to allow only specific devices to communicate. In addition, this does not automate the configuration on the device side. Therefore, the concept would have to be supplemented with predefined device configurations for specific device types. The last step would be to extend the concept to include device discovery so that devices do not have to know where the gateway is but are automatically integrated into the communication instead.

8.3 Middleware abstraction

Middleware abstraction is an additional problem to the already existing heterogeneity of protocols. Nowadays, there are many cloud providers, and in some cases, even several applications are used simultaneously. Moreover, these cloud providers always differ in their interfaces, so no consistent communication is possible.

Therefore, in this thesis, the existing adapter proposal is supplemented to solve this problem. The adapter allows the protocols and the cloud-specific interfaces to be translated. Of course, this presupposes that these can be covered by the message format and the internal communication in the gateway.

In the prototype example, the registration requests are thus further sent to AWS and processed. In addition, the standard messages from the gateway are extracted and adapted to the device shadow in IoT Core. This can optionally be turned off so that the message format from the gateway is used again.

So this proposal shows what an accurate middleware abstraction can look like. Furthermore, it does not require much effort to extend the already presented approach.

8.4 Resource management

In order for *Resource management* to be automated, several parameters must, of course, be monitored. For this reason, the following metrics are measured in an interval in the proposed prototype.

- CPU utilisation of the system
- CPU utilisation of the process
- Heap memory usage
- Non-heap memory usage
- Throughput of messages

Based on the related work, this is only a tiny part of the metrics, but this part can be measured without much effort. In addition to the measurements, these are used to prioritize messages and adjust QoS levels. Adjusting QoS levels can only be done thanks to the unified message format, which can be used

to define such messages. This allows, for example, to lower or raise the intervals of message exchange from the gateway. Of course, devices that do not implement this format can not be managed.

8.5 Traffic optimization

Traffic optimisation is not an easy topic to deal with. There are many different approaches to increase throughput, and they all need to be evaluated and assessed for this use case. For this reason, only straightforward methods were used initially, which were also used in the scientific publication in chapter 2.1.2 on page 11.

For this purpose, the incoming messages were first segmented into two queues. Then, the messages are placed in this queue and processed based on priorities. The exact algorithm used is a simple version of WQF with fixed weights. After selecting messages, they are submitted to a separate queue for the respective adapter. The influence of this approach can be observed in the following measurements.

However, a significant problem could already be identified in the measurements. When sending the messages, the scheduling algorithm is always faster than the actual adapter. This means that not the internal queue is filled, but the adapter's queue. If this queue is now full, prioritised messages are queued first internally, but the long queue of the adapter still delays them. While this can be considered fair, it can also cause problems for critical messages. The problem is visually illustrated in the following figure 8.1.

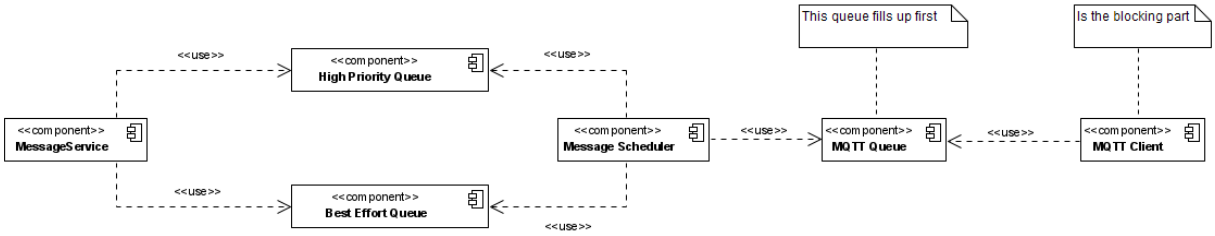


Figure 8.1: Issues in the scheduling process presented visually

In addition, compression of messages was introduced, but this is initially only used for protocols such as ZigBee, as this protocol cannot send large payloads. Here, Gzip is used with the highest available compression rate.

8.6 Analysis of Measurements

In order to more accurately evaluate the performance of the gateway, several test procedures were applied, and measurements were taken on them. The gateway's underload behaviour was tested in more detail, mainly whether crashes occur or how the response time behaves.

For this, loads were simulated on the MQTT and WebSocket protocols. Unfortunately, this was not possible for ZigBee since this protocol does not allow high data transfer rates and, therefore, only consumes low resources.

8.6.1 Gateway behaviour on high load

In the first test scenario, the load behaviour of the gateway was examined. The gateway was monitored for one hour and confronted with high data rates for this purpose. In each test, the message had a size of 579 bytes. The test was repeated several times, each time doubling the data rate until the gateway lost reliability. It is important to note that only one client was ever used for the measurements here. Furthermore, only WebSocket allows several connections to be made simultaneously, influencing the system among the available protocols.

A summary of the results for the test scenario introduced can be found in the table below. Based on this table, it can be said that the gateway can cope with 4000 messages per second via MQTT and 8000 messages per second via WebSocket. As already mentioned, however, these are only single client connections.

Messages/s	CPU system	CPU process	Heap memory	Non Heap memory	Response Time	Test successfull
			Baseline			
-	0.6%	0.35%	49.48 MB	56.23 MB	-	✓
			MQTT			
1000	18%	18%	40.18 MB	61.94 MB	5.25 ms	✓
2000	27%	27%	76.05 MB	52.93 MB	5.9 ms	✓
4000	54%	54%	55.75 MB	52.95 MB	12.19 ms	✓
8000	43%	42%	168.92 MB	53 MB	212.36463 s	-
			WebSocket			
1000	10%	8%	46.37 MB	55.08 MB	14.74 ms	✓
2000	17%	13%	47.43 MB	55.24 MB	15.67 ms	✓
4000	24%	19%	46.2 MB	55.08 MB	18.84 ms	✓
8000	43%	34%	45.48 MB	55.19 MB	91.82 ms	✓

Table 8.1: Average measured values of the evaluation

Baseline

To investigate the influence on the gateway, a measurement of the resources in an idle state was initially created. The results can be seen in the following figure 8.2. There was an average CPU utilisation of 0.67%, of which 0.35% came from the process itself. Furthermore, the average memory consumption is 39.48 MB on the heap and 56.23 MB outside the heap. Based on these values, the gateway hardly influences the system when idle.

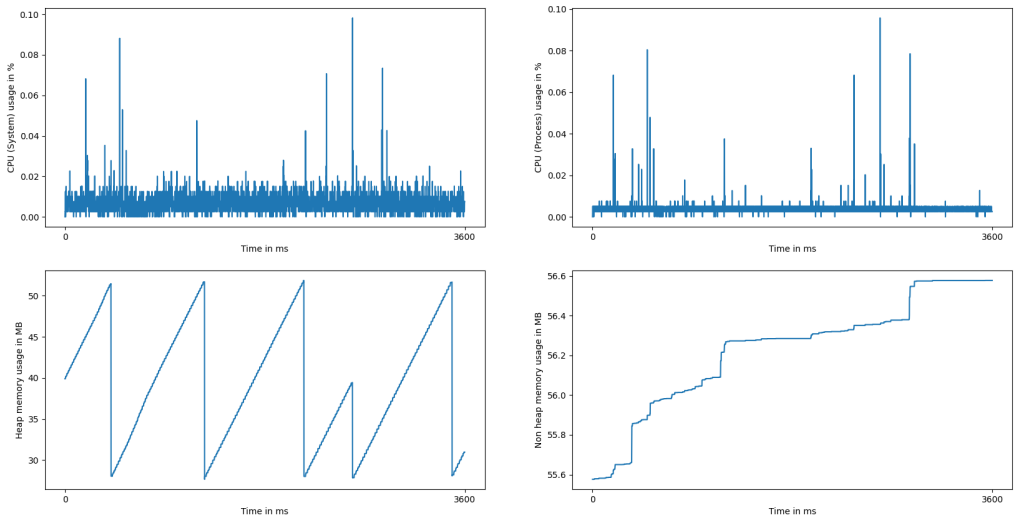


Figure 8.2: Resource consumption of the gateway in idle state

MQTT

A load of 1000 messages per second was initially simulated, starting with MQTT. The response times of the gateway can be found in the figure 8.3 and the resource utilization in the figure 8.4.

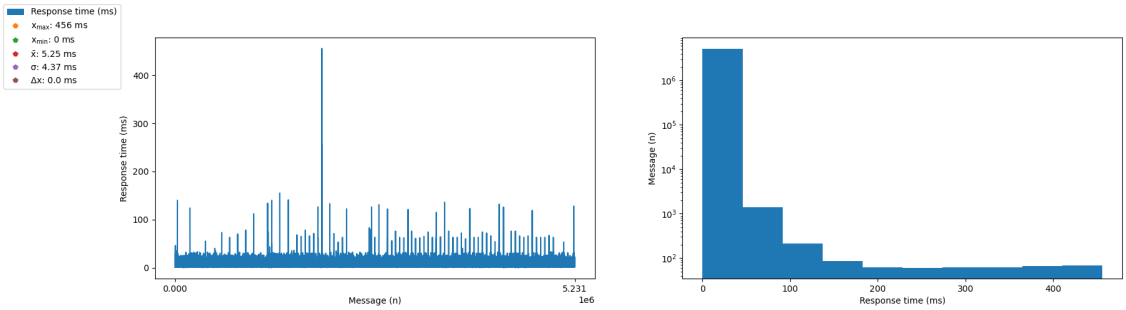


Figure 8.3: Response time of MQTT with an average load of 1000 messages per second.

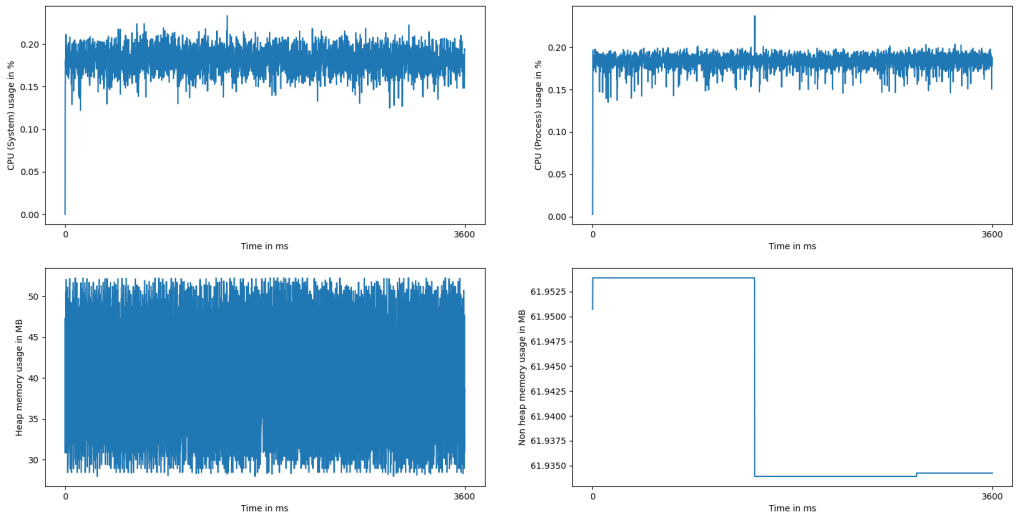


Figure 8.4: Resource consumption of MQTT with an average load of 1000 messages per second

It can be seen that the average response time of 5.25 ms is very low. There are a few swings, but these could also be due to the client or MQTT broker. Unfortunately, it is not possible to see the problem with this measurement. Additionally, just 20% of the CPU power is used. The memory usage of a maximum of 110 MB is also shallow. The conclusion is that 1000 messages per second is not yet a big task for the gateway.

Doubling the load still does not change much. This can be seen in the following figures 8.5 and 8.6.

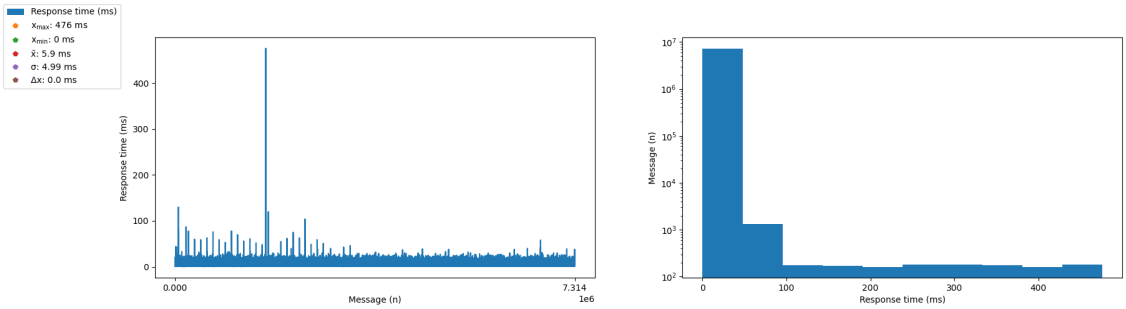


Figure 8.5: Response time of MQTT with an average load of 2000 messages per second

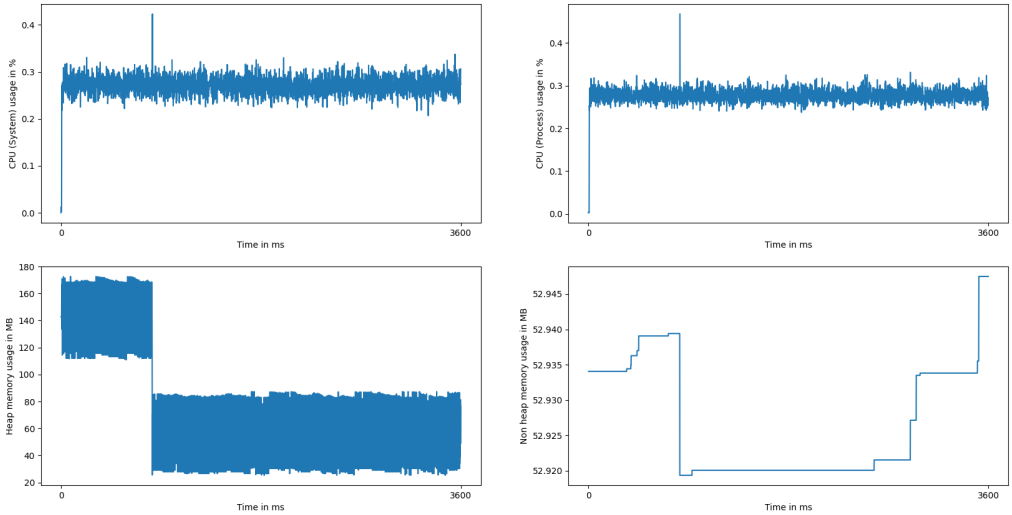


Figure 8.6: Resource consumption of MQTT with an average load of 2000 messages per second

Overall, the average response time has increased by just 0.45 ms. The CPU utilization has increased by just 10%. At first, it could be assumed that memory utilization has increased significantly. However, the garbage collector becomes active in the later course so that just a slight increase in memory can be seen.

With the next doubling of the load, there is already a clearer slope to be seen. The values for this can be seen in the following figures 8.7 and 8.8.

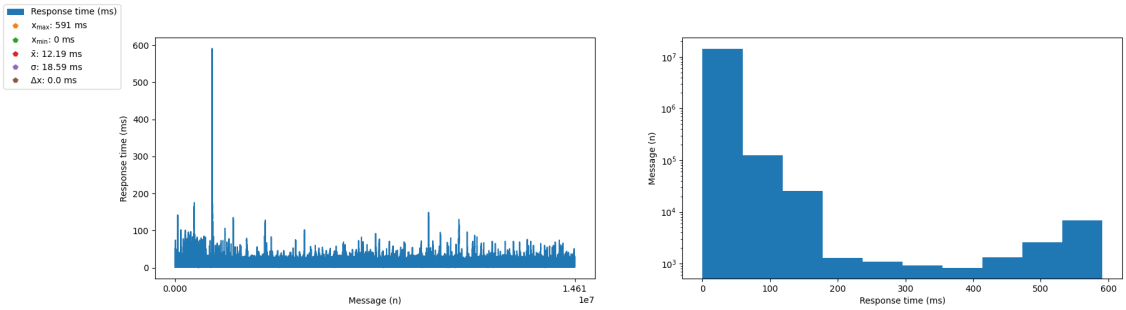


Figure 8.7: Response time of MQTT with an average load of 4000 messages per second

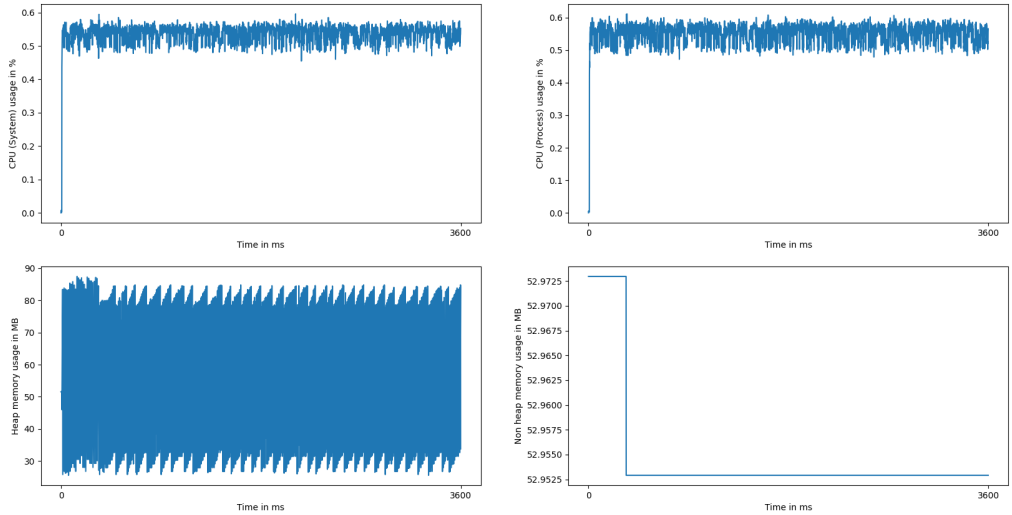


Figure 8.8: Resource consumption of MQTT with an average load of 4000 messages per second

The increase in load has doubled the response time and the CPU utilisation. However, the memory consumption is still in the same range. Nevertheless, the load and the response time are still within an acceptable range.

With the increase to 8000 messages per second, everything looks different. The gateway loses much reliability and is overloaded. The values for this can be seen in the following figures 8.9 and 8.10.

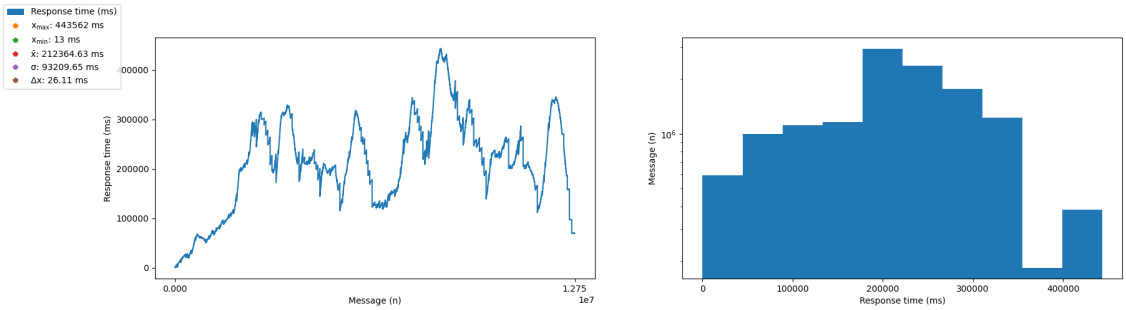


Figure 8.9: Response time of MQTT with an average load of 8000 messages per second

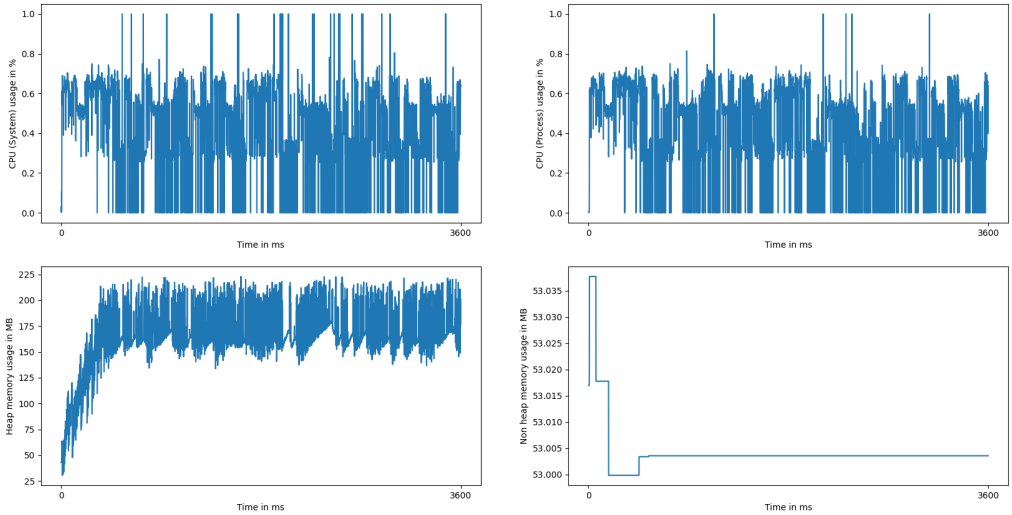


Figure 8.10: Resource consumption of MQTT with an average load of 8000 messages per second

The response time rises to an unacceptable level of 212.36463 seconds on average. In addition, the CPU reaches a load of 100%, and the memory increases threefold. The gateway is having problems, as well as the broker itself. The measurements had to be repeated several times because the broker interrupted the connections. In addition, not all sent messages arrived but were lost between the broker and the gateway.

One can now assume that a load of up to 4000 messages per second is acceptable based on MQTT measurements.

WebSocket

An initial load of 1000 messages per second was also simulated for WebSocket. The response time can be seen in the figure 8.11 and the resource consumption can be seen in the figure 8.12.

The response time via WebSocket is fundamentally higher than that of MQTT. While MQTT had an average response time of 5.9 ms, WebSocket had a response time of 14.74 ms. However, the average resource consumption is very similar to MQTT measurements. The average CPU utilisation is 10%, and the process itself consumes 8%. On average, 46.37 MB of heap memory and 55.08 non-heap memory are used.

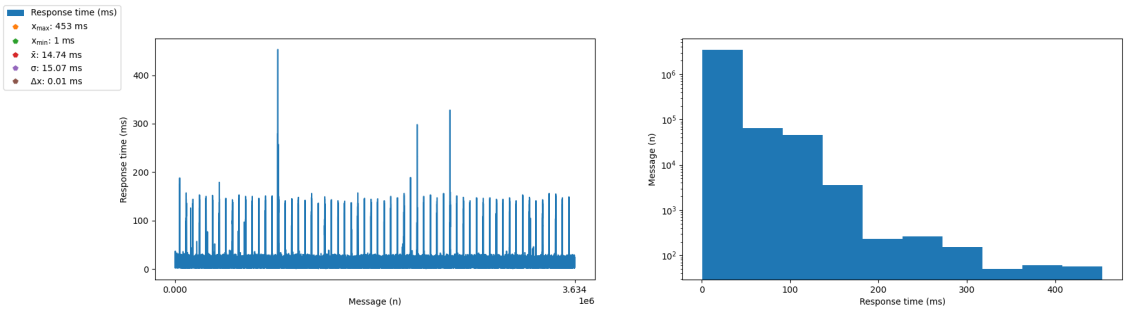


Figure 8.11: Response time of WebSocket with an average load of 1000 messages per second

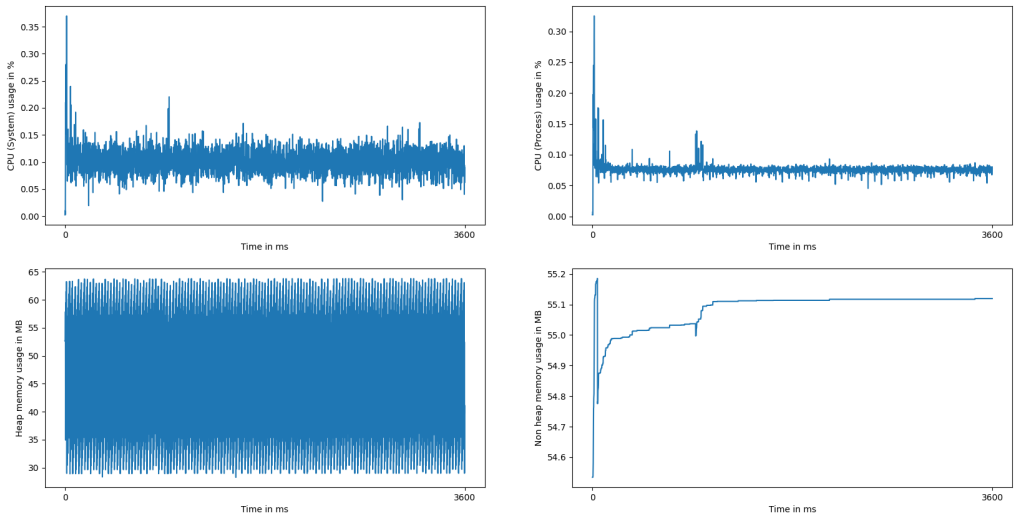


Figure 8.12: Resource consumption of WebSocket with an average load of 1000 messages per second

With the increase to 2000 messages per second, there is only a tiny increase in response time and resource consumption in this case. The measurements are shown in the following figures 8.13 and 8.14. The average response time is 15.67 ms. The actual resource consumption increases to an average of 17% CPU utilisation, 13% of which comes from the process itself. The memory consumption has not changed significantly and remains roughly in the same range.

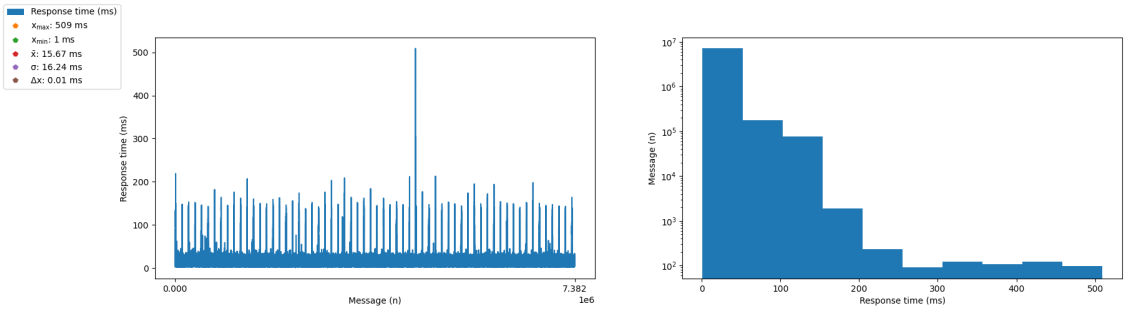


Figure 8.13: Response time of WebSocket with an average load of 2000 messages per second

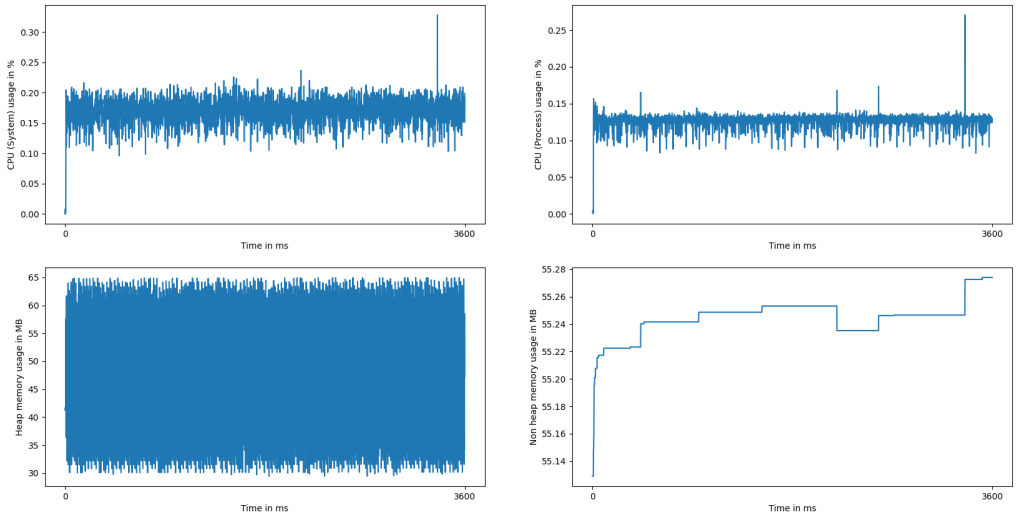


Figure 8.14: Resource consumption of WebSocket with an average load of 2000 messages per second

With the next increase to 4000 messages per second, a smaller increase in response time and resource consumption again. The measurements are shown in the following figures 8.15 and 8.16. The response time increases to an average of 18.84 ms. The CPU utilisation increases to 24%, but only 19% of this is from the process itself. The memory utilisation is in the same range here.

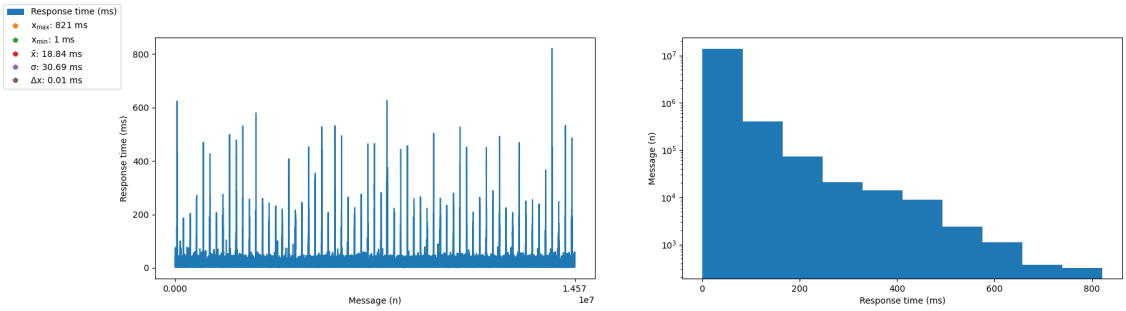


Figure 8.15: Response time of WebSocket with an average load of 4000 messages per second

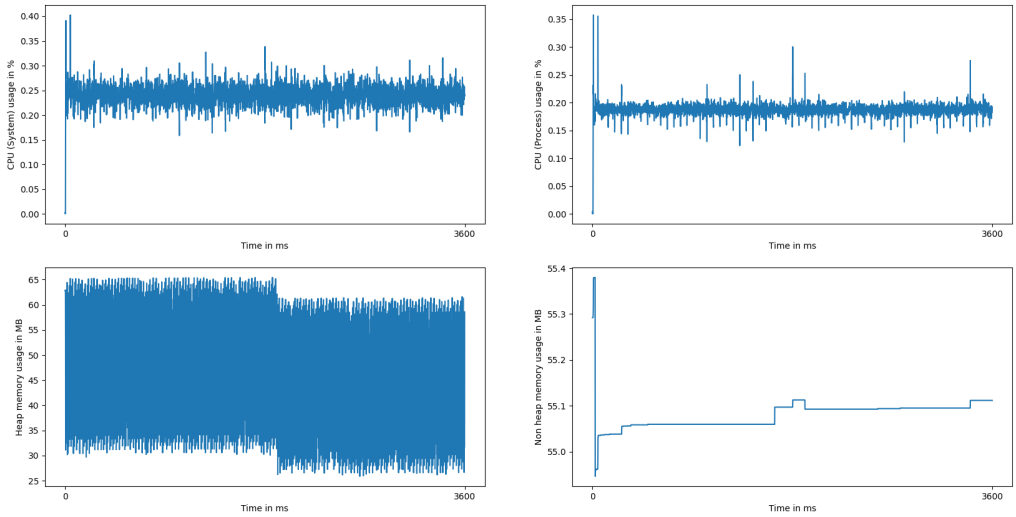


Figure 8.16: Resource consumption of WebSocket with an average load of 4000 messages per second

With the increase to 8000 messages per second, transmission via MQTT was no longer possible. This still seems to be just about possible with WebSocket based on the measurement. The measurements are shown in the following figures 8.17 and 8.18. The response time increases to an average of 91.82 ms. The CPU load also seems to increase to only 43%, whereby the process itself generates 34%. The memory utilisation also remains roughly in the same range here.

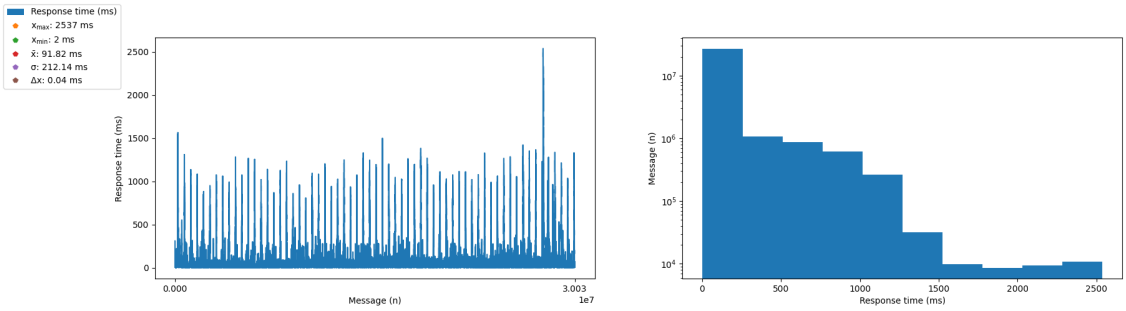


Figure 8.17: Response time of WebSocket with an average load of 8000 messages per second

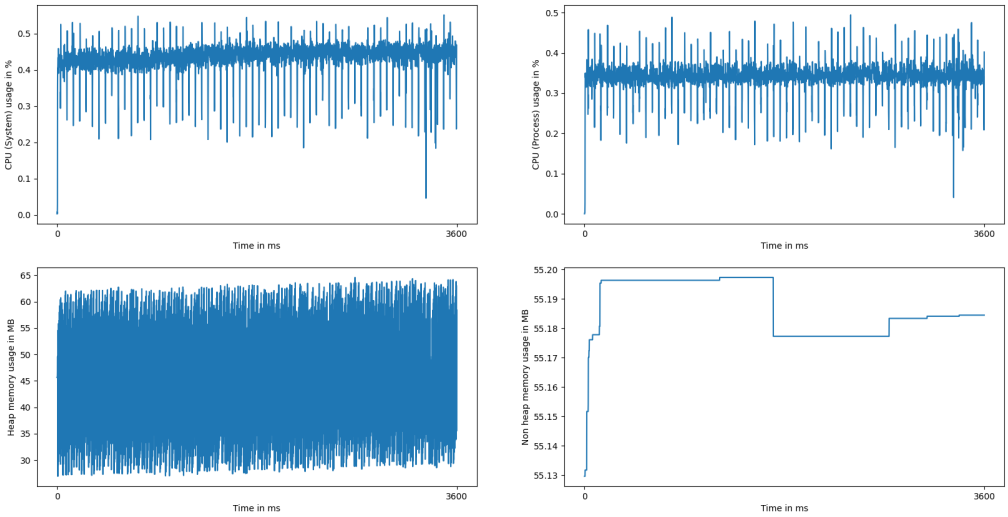


Figure 8.18: Resource consumption of WebSocket with an average load of 8000 messages per second

A further doubling of the message load was not possible because, on the one hand, the bandwidth of the Raspberry PI is exceeded, and, on the other hand, loads of over 8000 messages per second regularly led to crashes. As a result, not one measurement could be executed entirely for more than an hour. For this reason, loads of up to 8000 messages per second are acceptable for WebSocket.

8.6.2 Gateway behaviour of priority messages

In addition to the normal load behaviour, the behaviour of high priority messages was investigated. Two clients each simulated a load of 1000 messages per second. One of these clients only sent high-priority messages, and the other sent normal messages. The response times are shown in the following figures 8.19 and 8.20.

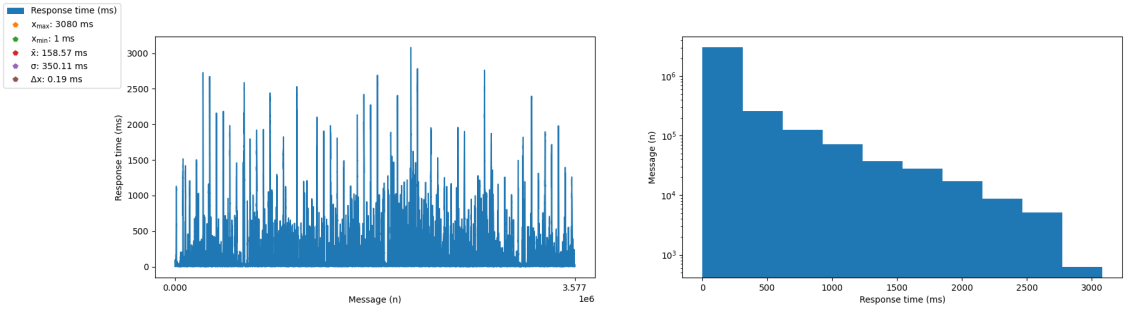


Figure 8.19: Response time of WebSocket with an average load of 2000 messages per second (best effort)

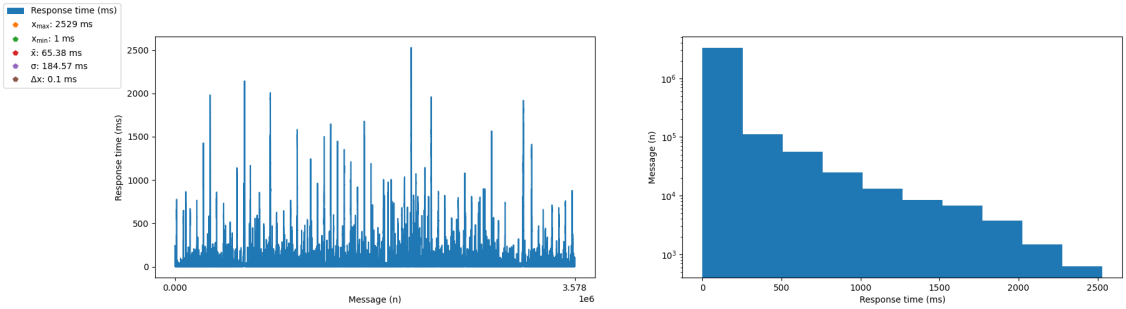


Figure 8.20: Response time of WebSocket with an average load of 2000 messages per second (high priority)

It can be seen that the response time of high priority messages is, on average, significantly lower than the normal ones. While high priority messages have an average response time of 65.38 ms, regular messages have an average response time of 158.57 ms. Interestingly, the response times here seem to be significantly higher than the previous measurements. This probably has to do with the fact that multiple connections are now used. For this reason, further measurements are needed to investigate the influence of multiple clients on the response time.

8.6.3 Gateway behaviour on long time periods

In addition to the previous measurements, the MTTF and MTBF should be measured. Unfortunately, the period of two measured months was too small. Therefore, there were no critical failures and only one connection failure caused by a router restart. For this test, three clients were simulated on the test system: one debug-xbee-client with one message per second and one debug-websocket-client with 1000 messages per second and one debug-mqtt-client also with 1000 messages per second.

Chapter 9

Conclusion

IoT is a fast-growing area and is thus becoming increasingly important. Unfortunately, strong growth is also associated with many different problems. While a great deal of research has already been done on the topic of IoT in the fields of cloud computing, fog computing, and edge computing, little seems to have happened in the field of the gateway in particular.

The different problems are highly diverse. However, based on extensive research, the main problems can be found in the areas of heterogeneity, scalability, QoS, security and privacy, intelligence, and robustness. The exact definitions of these problems can be found in chapter 2.1.1 on page 9. On the other hand, many problems arise in managing the devices as they continue to increase.

A concept that can be called an automatic gateway is required to solve this problem. For this reason, this thesis proposes a prototype to take the first step toward an automatic gateway. Since not all of the problems mentioned can be solved within a simple Master Thesis, only a part of them has been dealt with. However, this part forms the basic functionalities that must be available for such a gateway to be called an automatic gateway.

The mentioned prototype deals with the problems of protocol conversion, device management, middleware abstraction, resource management and traffic optimisation. Protocol conversion, device management and middleware abstraction try to solve the problem of heterogeneity in the gateway environment. Resource management and traffic optimisation are oriented to the problem of scalability and partly to robustness. However, based on the functionalities mentioned and the literature researched, the proposed gateway cannot yet be classified as a fully automatic gateway but only as a semi-automatic gateway. There are still some steps to take for a fully automated gateway, as many points have not been addressed in this thesis.

The prototype mentioned was implemented in Java for the Raspberry Pi but can be executed on all JVM-supported systems. In addition, the prototype is based on the OSGi standard to enable modularisation of the individual components.

Five components were developed for the actual gateway. The components include the gateway and several modules to communicate via different protocols. These modules were implemented for MQTT, ZigBee (XBee) and WebSocket protocols. In addition, a module was developed to communicate with AWS IoT Core as an end consumer. The end consumer represents the real world more accurately and shows what a middleware abstraction can look like.

In addition to the modules, a different message format was developed to achieve these goals. The message format is still optional, as senders and receivers do not have to use it. This format only enables more complex message exchange between the individual protocols. Without this format, all other messages are treated as upstream messages and forwarded to the defined end consumer, which in this case is AWS.

Debug clients were also developed to test and evaluate the system. However, these clients do not implement the mentioned message format entirely and are only intended for testing the application.

In order to demonstrate the feasibility of the above approach, an evaluation was also carried out. First,

the stated goals were compared with the achieved results and evaluated. Open problems were also identified, which are also discussed in the outlook. Furthermore, several performances and load tests were performed to understand:

1. How much delay is caused by the system design, and can the system communicate in near real-time?
2. How many messages can the gateway handle simultaneously, and what impact does a high message count have on the gateway?
3. What impact the prioritisation of messages has on response time?
4. How long does the gateway run without errors, or more precisely, what are the MTTF and MTBF?

The measurements have shown that a message load of 4000 messages per second can be considered acceptable, independent of the protocol. While even 8000 messages were still within the realm of possibility for WebSocket, this does not apply to MQTT. Nevertheless, the response time increased to almost 100ms here as well. Measurements with more messages per second could not be performed because the connections were constantly interrupted. However, the measurements do not yet consider how many clients can connect simultaneously, for example. With MQTT, this mainly depends on the broker, but the clients connect directly to the gateway with WebSocket. ZigBee could not perform precise load measurements since this protocol does not allow high data rates. In addition, prioritising messages improved the response time by approximately 242%. However, it should be noted that the response time increased significantly to 65.38 ms and 158.57 ms due to the use of two clients. A total load of 4000 messages per second was simulated via WebSocket. In addition to the performance measurements, the runtime was measured over two months. No critical errors occurred, so it can be assumed that the design mentioned runs relatively reliably. Of course, even more, extended periods should be measured. For this test, a load of 1000 messages per second was simulated on each protocol and a load of only one message per second on ZigBee.

The proposed gateway forms a modularised solution to solve the mentioned problems. The gateway can be easily extended to support additional protocols and consumers. Device management is also automated, reducing a significant administrative burden. Also, some aspects are supported to increase the scalability and robustness of the system. Nevertheless, the approach needs to be further optimised to improve performance. Also, some issues have not yet been addressed in this thesis, which is necessary for productive use. In particular, the problem of security still needs to be addressed, as it has not been dealt with here at all.

9.1 Outlook

The following chapter deals on the one hand with which work packages can still be dealt with in the mandatory objectives and on the other hand with which other topics are still open.

9.1.1 Mandatory objectives

In the following section, open points in the mandatory goals are explained and presented.

Protocol conversion: The thesis presented a possibility to convert the protocols, whereby a unique message format was developed. However, this format is still very rudimentary, so it must be further developed. On the one hand, the header size is too large because it uses the device names for addressing. In addition, different mechanisms could be implemented, such as protocol-independent encryption. This would enable encryption from the device to the end consumer. Also, some protocols like ZigBee support only small payloads so that a mechanism for splitting messages can be implemented. These are only two suggestions for improvement, but there are many more. Nevertheless, the approach's potential is great because it could be used to establish a standard for the IoT world that can act independently of protocols.

Device management: For device management, a concept for automatic registration and administration was presented. Here, the registrations are forwarded to the consumers and entered into the gateway. While this process reduces the configuration effort significantly, several extension possibilities exist. For example, devices could be discovered and configured through device discovery. This would reduce the configuration of devices completely before starting them. In addition, further mechanisms are missing to configure and manage devices subsequently via the gateway. This point could be even more attractive in a multi gateway environment. These examples still show considerable potential for improvement in the device management environment.

Middleware abstraction: The middleware abstraction can be solved well by the proposed gateway. However, developing a separate module for each consumer to translate the communication is still necessary. It would also be possible to develop the developed data format further as a standard so that cloud providers, for example, can use it. This would completely resolve the issue of middleware abstraction since a generally applicable standard would apply. However, user-defined solutions would still be possible due to the concept. For this reason, there are still possibilities for improvement, but they are challenging to implement.

Ressource management: Resource management is divided into monitoring and scheduling. While some parameters are already included in the monitoring, some are still missing. For example, the bandwidth, jitter and roundtrip delay could also be measured. This way, not only internal parameters could be taken into account, but also external ones.

The scheduling process itself is still very simply implemented. It can already be used to reduce the QoS parameters of devices and block messages. In principle, there are still many possibilities for expansion, for example, mechanisms could be built to increase the prioritisation of processes. For example, individual protocols could be prioritised if the load on the respective interface increases. For a network of individual gateways, these could even exchange information to distribute loads. These are only two examples, but the potential in this environment is still huge.

Traffic optimization: Simple techniques were also used for traffic optimisation. On the one hand, a high priority and best-effort queue were implemented. On the other hand, a simple version of the WFQ algorithm with fixed weights was used for scheduling. This has already led to improvements in the latencies of high-priority messages. However, the current design also has a problem because each protocol adapter has a queue to process the messages. This queue is the first to fill up with high loads so that high-priority messages can be delayed. The exact explanation of the problem was explained in more detail in the evaluation. For this reason, this problem should be eliminated.

In addition, different algorithms need to be investigated to find out which one is best suited. It would also be possible to exchange the algorithm through the configuration. This is possible without any problems due to the modular design.

9.1.2 Open topics

The following section explains open questions that have not been addressed in this thesis but emerged as necessary from the literature research.

Other drivers and protocols: The thesis only concentrates on a sub-area of the protocols, as it is mainly about a proof of concept. However, other protocols are also interesting to investigate, as there are specific protocols that may also create new problems. In particular, the connection to OPC-UA should be considered here. Many companies in the industry use this standard to transfer data to the cloud. The integration would mean a big step towards connecting intelligent industrial devices.

Security and Privacy: In principle, security is a critical topic in the Internet of Things. However, since gateways are the data entry points to the network, they are exposed to even more security risks. The individual security threats in this regard have already been presented and introduced in chapter 2.1.5 on page 14, for example. Therefore, the topic of security and privacy is a topic that cannot be neglected. For this reason, work must first be done on this topic before the prototype presented can be used productively.

Intelligence: The gateway should be smart enough to recognise which data should be forwarded and which can be processed locally. This could lead to significantly lower latency and reduce the load on the network.

Robustness: Since a gateway is to be used to exchange messages with the cloud, it can also happen that this gateway fails. Therefore, a redundant mechanism would be helpful to in order to be able to continue to function even in the event of failures.

Workload balancing: The planned gateway should distribute the available resources as automatically as possible to remain operable. However, hardware resources are always limited, which will always become a bottleneck at some point. Therefore, it makes sense to consider several gateways to make more computing power available. Of course, these gateways must then be selected by the devices based on suitable parameters.

Bibliography

[AG18] Siemens AG. *PROFINET – der Puls des digitalen Unternehmens*. https://assets.new.siemens.com/siemens/assets/api/uuid:007470ec-18d0-4de0-9eef-5fd4b49744b0/-dffa-b10530-00-profinet-broschuere72dpi.pdf?ste_sid=bf3ced01c1d351f4075bce0e7a94382c. [Online; accessed 29-April-2022]. 2018.

[Ala18] Tanweer Alam. "A reliable communication framework and its use in internet of things (IoT)". In: *CSEIT1835111| Received* 10 (2018), pp. 450–456.

[Ama21] Amazon. *Raspberry Pi 3 Model B+*. 2021. URL: https://www.amazon.de/Raspberry-1373331-Pi-Modell-Mainboard/dp/B07BDR5PDW.

[Aut22] The ThingsBoard Authors. *ThingsBoard IoT Gateway Documentation*. 2022. URL: https://thingsboard.io/docs/iot-gateway/ (visited on 02/01/2022).

[Bos+21] Marcin Bosk, Marija Gajić, Susanna Schwarzmann, Stanislav Lange, and Thomas Zinner. "HTBQueue: A Hierarchical Token Bucket Implementation for the OMNeT++/INET Framework". In: *arXiv preprint arXiv:2109.12879* (2021).

[BS21] Gunjan Beniwal and Anita Singhrova. "A systematic literature review on IoT gateways". In: *Journal of King Saud University-Computer and Information Sciences* (2021). Elsevier.

[Cal+19] Francisco L de Caldas Filho, Rodrigo L Rocha, Cláudia JB Abbas, Lucas MC E Martins, Edna Dias Canedo, and Rafael T de Sousa. "Qos scheduling algorithm for a fog iot gateway". In: *2019 Workshop on Communication Networks and Power Systems (WCNPS)*. IEEE. 2019, pp. 1–6.

[Com22] openHAB Foundation e.V. openHAB Community. *openHab documentation*. 2022. URL: https://www.openhab.org/docs/ (visited on 02/01/2022).

[Cor22a] Amazon Web Services IoT Core. *AWS IoT Core- Developer Guide*. 2022. URL: https://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html (visited on 01/28/2022).

[Cor22b] Oracle Corporation. *Java Authentication and Authorization Service (JAAS) Reference Guide*. https://docs.oracle.com/javase/8/docs/technotes/guides/security/jaas/JAASRefGuide.html. [Online; accessed 24-May-2022]. 2022.

[CPU22] CPUID. *CPU-Z System information software*. https://www.cpuid.com/softwares/cpu-z.html. [Online; accessed 24-May-2022]. 2022.

[DSA15] Pratikkumar Desai, Amit Sheth, and Pramod Anantharam. "Semantic gateway as a service architecture for iot interoperability". In: *2015 IEEE International Conference on Mobile Services*. IEEE. 2015, pp. 313–319.

[Elh+18] Sakina Elhadi, Abdelaziz Marzak, Nawal Sael, and Soukaina Merzouk. "Comparative study of IoT protocols". In: *Smart Application and Data Analysis for Smart Cities (SADASC'18)* (2018).

[FA18] Qiang Fan and Nirwan Ansari. "Towards workload balancing in fog computing empowered IoT". In: *IEEE Transactions on Network Science and Engineering* 7.1 (2018), pp. 253–262.

[Fel04] Joachim Feld. "PROFINET-scalable factory communication for all applications". In: *IEEE International Workshop on Factory Communication Systems, 2004. Proceedings.* IEEE. 2004, pp. 33–38.

[Fil+17] Francisco L de Caldas Filho, Lucas MC e Martins, Ingrid Palma Araújo, Fábio LL de Mendonça, João Paulo CL da Costa, and Rafael T de Sousa Júnior. "Design and evaluation of a semantic gateway prototype for IoT networks". In: *Companion Proceedings of the10th International Conference on Utility and Cloud Computing*. 2017, pp. 195–201.

[Fou17] OPC Foundation. *Unified Architecture. Part 1: Overview and Concepts*. https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-1-overview-and-concepts/. [Online; accessed 29-April-2022]. 2017.

[Fou22a] Eclipse Foundation. *The extensible open source Java/OSGi IoT Edge Framework*. 2022. URL: https://www.eclipse.org/kura/ (visited on 02/01/2022).

[Fou22b] OPC Foundation. *Unified Architecture.* https://opcfoundation.org/about/opc-technologies/opc-ua/. [Online; accessed 29-April-2022]. 2022.

[Fou22c] The Apache Software Foundation. *Apache Karaf Container 4.x - Documentation*. 2022. URL: https://karaf.apache.org/manual/latest/ (visited on 02/14/2022).

[Fou22d] The Apache Software Foundation. *Maven - Introduction*. 2022. URL: https://maven.apache.org/what-is-maven.html (visited on 02/14/2022).

[Fou22e] EdgeX Foundry. *EdgeX Foundry Documentation*. 2022. URL: https://docs.edgexfoundry.org/2.1/ (visited on 02/01/2022).

[Guo+13] Shang Guoqiang, Chen Yanming, Zuo Chao, and Zhu Yanxu. "Design and implementation of a smart IoT gateway". In: *2013 IEEE international conference on green computing and communications and IEEE internet of things and IEEE cyber, physical and social computing*. IEEE. 2013, pp. 720–723.

[Has+19] Vikas Hassija, Vinay Chamola, Vikas Saxena, Divyansh Jain, Pranav Goyal, and Biplab Sikdar. "A survey on IoT security: application areas, security threats, and solution architectures". In: *IEEE Access* 7 (2019), pp. 82721–82743.

[Inc20] Digi International Inc. *Digi XBee® 3 ZigBee®*. 2020. URL: %5Curl%7Bhttps://www.digi.com/resources/documentation/digidocs/pdfs/90001539.pdf%7D (visited on 01/05/2022).

[Inc21a] DIGI International Inc. *DIGI XBEE 3 ZIGBEE MESH KIT*. 2021. URL: https://www.digi.com/resources/library/data-sheets/ds-xbee3-zigbee-mesh-kit.

[Inc21b] DIGI International Inc. *DIGI XBEE 3 ZIGBEE MESH KIT*. 2021. URL: https://www.digi.com/products/embedded-systems/digi-xbee/digi-xbee-tools/xctu.

[Inc22a] DIGI International Inc. *XBee Java Library*. https://github.com/digidotcom/xbee-java. [Online; accessed 22-February-2022]. 2022.

[Inc22b] DIGI International Inc. *XBee Java Library*. https://www.digi.com/resources/documentation/digidocs/PDFs/90001438.pdf. [Online; accessed 22-February-2022]. 2022.

[Jar22] Trent Jarvi. *nrjavaserial*. http://rxtx.qbang.org/wiki/index.php/Main_Page. [Online; accessed 22-February-2022]. 2022.

[Jet22] Jetbrains. *What is IntelliJ IDEA?* https://www.jetbrains.com/idea/features/. [Online; accessed 22-February-2022]. 2022.

[KC18] Byungseok Kang and Hyunseung Choo. "An experimental study of a reliable IoT gateway". In: *ICT Express* 4.3 (2018), pp. 130–133.

[KKC17] Byungseok Kang, Daecheon Kim, and Hyunseung Choo. "Internet of everything: A large-scale autonomic IoT gateway". In: *IEEE Transactions on Multi-Scale Computing Systems* 3.3 (2017), pp. 206–214.

[Liu+10] Yao Liu, Peng Ning, Huaiyu Dai, and An Liu. "Randomized differential DSSS: Jamming-resistant wireless broadcast communication". In: *2010 Proceedings IEEE INFOCOM*. IEEE. 2010, pp. 1–9.

[Liu09] Chienyuan Liu. "The design of home care assistant system by the ZigBee technology". In: *Life Science Journal* 6.2 (2009).

[Mat20] Prof. Dr.-Ing. Jasminka Matevska. *V02 Anforderungs-Engineering F4 TI Matevska SOFTW2 SoSe 20*. 2020.

[Min+14] Dong Min, Zeng Xiao, Bi Sheng, Huang Quanyong, and Pan Xuwei. "Design and implementation of heterogeneous IOT gateway based on dynamic priority scheduling algorithm". In: *Transactions of the Institute of Measurement and Control* 36.7 (2014), pp. 924–931.

[mqtt-v5.0] Andrew Banks, Ed Briggs, Ken Borgendale, and Rahul Gupta. *MQTT Version 5.0*. OASIS Standard. `https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html`. Latest version: `https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html`. 7March 2019.

[MT+16] Mustafa El Gili Mustafa, Samani A Talab, et al. "The effect of queuing mechanisms first in first out (FIFO), priority queuing (PQ) and weighted fair queuing (WFQ) on network's routers and applications". In: *Wireless Sensor Network* 8.05 (2016), p. 77.

[Neu22] NeuronRobotics. *nrjavaserial*. `https://github.com/NeuronRobotics/nrjavaserial`. [Online; accessed 22-February-2022]. 2022.

[Res22] Copyright © Eclipse Foundation. All Rights Reserved. *Eclipse Mosquitto™*. `https://mosquitto.org/`. [Online; accessed 22-February-2022]. 2022.

[RFC6202] Salvatore Loreto, P Saint-Andre, Sd Salsano, and G Wilkins. "Known issues and best practices for the use of long polling and streaming in bidirectional http". In: *Internet Engineering Task Force, Request for Comments* 6202.2070-1721 (2011), p. 32.

[RFC6455] Ian Fette and Alexey Melnikov. *The websocket protocol*. 2011.

[Sal12] Mohammed A Abdala& Alaa Mohammed Salih. "Design and Performance Analysis of Building Monitoring System with Wireless Sensor Networks". In: *Iraqi Journal of Science* 53.4 (2012), pp. 1097–1102.

[Sar18] Vasil Sarafov. "Comparison of iot data protocol overhead". In: *Proceedings of the Seminars of Future Internet (FI) and Innovative Internet Technologies and Mobile Communication (IITM)*. Vol. 720. 2018.

[SB13] Michael H Schwarz and Josef Börcsök. "A survey on OPC and OPC-UA: About the standard, developments and investigations". In: *2013 XXIV International Conference on Information, Communication and Automation Technologies (ICAT)*. IEEE. 2013, pp. 1–6.

[Ser22] Amazon Web Services. *AWS - Webseite*. 2022. URL: `https://aws.amazon.com/de/` (visited on 01/28/2022).

[SM17] Dipa Soni and Ashwin Makwana. "A survey on mqtt: a protocol of internet of things (iot)". In: *International Conference On Telecommunication, Power Analysis And Computing Techniques (ICTPACT-2017)*. Vol. 20. 2017.

[SP12] Nisha Ashok Somani and Yask Patel. "Zigbee: A low power wireless technology for industrial applications". In: *International Journal of Control Theory and Computer Modelling (IJCTCM)* 2.3 (2012), pp. 27–33.

[STP13] Victor Santos, Joao Pedro Trovao, and P.G. Pereirinha. "Wireless Communications Usage in Smart Grids Home Area Network: a Survey". In: Sept. 2013.

[Wik22a] Wikipedia. *Gzip — Wikipedia, The Free Encyclopedia*. `http://en.wikipedia.org/w/index.php?title=Gzip&oldid=1084110597`. [Online; accessed 26-April-2022]. 2022.

[Wik22b] Wikipedia. *I²C — Wikipedia, The Free Encyclopedia*. `http://en.wikipedia.org/w/index.php?title=I%C2%B2C&oldid=1087567666`. [Online; accessed 24-May-2022]. 2022.

[Wik22c] Wikipedia. *Modbus — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Modbus&oldid=1086616226. [Online; accessed 24-May-2022]. 2022.

[Wik22d] Wikipedia. *MoSCoW-Priorisierung — Wikipedia, The Free Encyclopedia*. http://de.wikipedia.org/w/index.php?title=MoSCoW-Priorisierung&oldid=220293338. [Online; accessed 22-February-2022]. 2022.

[Wik22e] Wikipedia. *OSGi — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=OSGi&oldid=1085199168. [Online; accessed 15-May-2022]. 2022.

[Wik22f] Wikipedia. *Verfügbarkeit — Wikipedia, The Free Encyclopedia*. http://de.wikipedia.org/w/index.php?title=Verf%C3%BCgbarkeit&oldid=206844047. [Online; accessed 19-February-2022]. 2022.

[Wik22g] Wikipedia. *Watchdog timer — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Watchdog%20timer&oldid=1083223779. [Online; accessed 24-May-2022]. 2022.

[Wik22h] Wikipedia. *WebSocket — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=WebSocket&oldid=1062054519. [Online; accessed 04-January-2022]. 2022.

Appendix A

XBee 3 ZigBee Mesh Kit

DIGI XBEE 3 ZIGBEE MESH KIT

Explore hands-on mesh networking and connectivity with Digi XBee 3 RF modules

The **Digi XBee® 3 Zigbee Mesh Kit** offers a great way to learn how to use Digi XBee 3 RF modules for device connectivity and Zigbee-based mesh networking. Zigbee is one of the most popular open standard mesh networking protocols, specifically designed for low-data rate and low-power applications. With simple examples and step-by-step guidance, you can quickly assemble kit components to create reliable, low-power device communications and sensor networks.

Mesh networking is a powerful way to route data. Range is extended by allowing data to hop from node to node, and reliability is increased by "self-healing," the ability to create alternate paths when one node fails or a connection is lost.

Digi XBee 3 Zigbee Modules Included in the Kit

Digi XBee 3 and XBee 3-PRO Zigbee modules are ideal for applications in the energy and controls markets where time-to-market and reliability are critical. With Digi's extensive and easy-to-use DIGI XBee API framework, customers can get their Zigbee product to market faster than any other module available in the industry. Features like binding and multicasting also allow for simple integration for building automation applications.

The Kit includes:

- ✓ 3 Digi XBee Grove Development Boards
- ✓ 3 Digi XBee ZigBee SMT Modules, programmable to 802.15.4 and DigiMesh
- ✓ 3 Micro-USB cables
- ✓ 3 antennas
- ✓ Comprehensive web and video-based instruction

NUMBER	DESCRIPTION
XK3-Z8S-WZM	Digi XBee 3 Zigbee Mesh Kit, worldwide

Our modules are available in the popular Digi XBee through-hole, surface mount and now micro-mount form factors, providing customers the flexibility to substitute one Digi XBee technology for another with minimal development time and risk. Using the long-range Digi XBee 3-PRO variant, customers can get up to two miles (3200 meters) LoS range.

SPECIFICATIONS	Digi XBee® 3 Zigbee 3.0	Digi XBee® 3 PRO Zigbee 3.0
PERFORMANCE		
TRANSCEIVER CHIPSET	Silicon Labs® EFR32MG SoC	
DATA RATE	RF 250 Kbps, Serial up to 1 Mbps	
INDOOR/URBAN RANGE	60 m (200 ft)	90 m (300 ft)
OUTDOOR/RF LINE-OF-SIGHT RANGE	1200 m (4000 ft)	3200 m (2 miles)
TRANSMIT POWER	+8 dBm	+19 dBm
RECEIVER SENSITIVITY (1% PER)	-103 dBm Normal Mode	
FEATURES		
SERIAL DATA INTERFACE	UART, SPI, I²C	
CONFIGURATION METHOD	API or AT commands, local or over-the-air (OTA)	
FREQUENCY BAND	ISM 2.4 GHz	
FORM FACTOR	Micro, Through-Hole, Surface Mount	
INTERFERENCE IMMUNITY	DSSS (Direct Sequence Spread Spectrum)	
ADC INPUTS	(4) 10-bit ADC inputs	
DIGITAL I/O	15	
ANTENNA OPTIONS	Through-Hole: PCB Antenna, U.FL Connector, RPSMA Connector SMT: RF Pad, PCB Antenna, or U.FL Connector Micro: U.FL Antenna, RF Pad, Chip Antenna	
OPERATING TEMPERATURE	-40° C to 85° C (-40° F to 185° F)	
DIMENSIONS (L X W X H)	Through-Hole: 2.438 x 2.761 cm (0.960 x 1.087 in) SMT: 2.199 x 3.4 x 0.305 cm (0.866 x 1.33 x 0.120 in) Micro: 13 x 19 x 2 mm (0.533 x 0.76 x 0.087 in)	
PROGRAMMABILITY		
MEMORY	1 MB / 128 KB RAM	
CPU/CLOCK SPEED	HCS08 / up to 50.33 MHz	
NETWORKING AND SECURITY		
PROTOCOL	Zigbee 3.0	
ENCRYPTION	128/256 bit AES	
RELIABLE PACKET DELIVERY	Retries/Acknowledgements	
IDS	PAN ID and addresses, cluster IDs and endpoints (optional)	
CHANNELS	16 channels	
POWER REQUIREMENTS		
SUPPLY VOLTAGE	2.1 to 3.6 V	
TRANSMIT CURRENT	40 mA @ 8 dBm	135 mA @ 19 dBm
RECEIVE CURRENT	15 mA	
POWER-DOWN CURRENT	1.7 micro Amp @ 25° C (77° F)	
REGULATORY APPROVALS		
FCC, IC (NORTH AMERICA)	Yes	Yes
ETSI (EUROPE)	Yes	No

It's the easy and fast way to build a wireless mesh network using Digi XBee 3 modules. To learn more visit hub.digi.com.

877-912-3444 | 952-912-3444

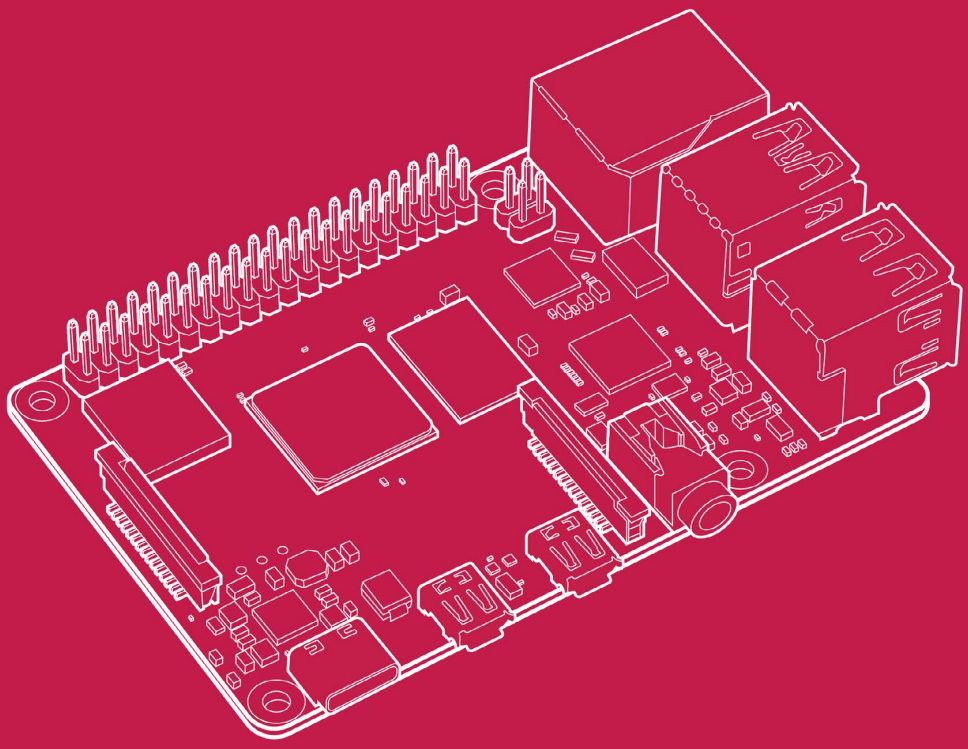
© 1996-2021 Digi International Inc. All rights reserved. All other trademarks are the property of their respective owners.

91004234
A3/1021

Appendix B

Raspberry Pi 4 Computer Model B

Raspberry Pi 4 Computer
Model B

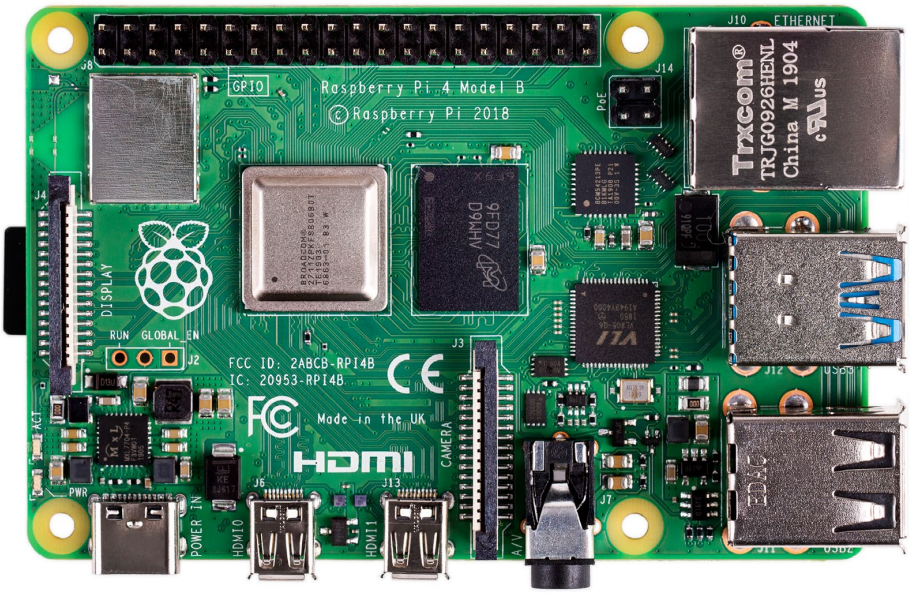


Published in January 2021
by Raspberry Pi Trading Ltd.

Raspberry Pi

www.raspberrypi.org

Overview



Raspberry Pi 4 Model B is the latest product in the popular Raspberry Pi range of computers. It offers ground-breaking increases in processor speed, multimedia performance, memory, and connectivity compared to the prior-generation Raspberry Pi 3 Model B+, while retaining backwards compatibility and similar power consumption. For the end user, Raspberry Pi 4 Model B provides desktop performance comparable to entry-level x86 PC systems.

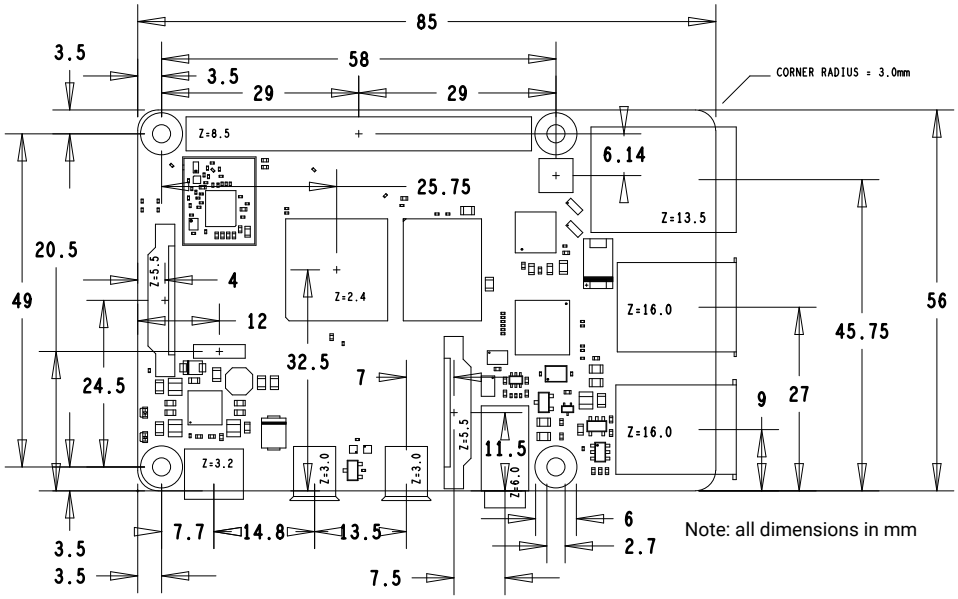
This product's key features include a high-performance 64-bit quad-core processor, dual-display support at resolutions up to 4K via a pair of micro-HDMI ports, hardware video decode at up to 4Kp60, up to 8GB of RAM, dual-band 2.4/5.0 GHz wireless LAN, Bluetooth 5.0, Gigabit Ethernet, USB 3.0, and PoE capability (via a separate PoE HAT add-on).

The dual-band wireless LAN and Bluetooth have modular compliance certification, allowing the board to be designed into end products with significantly reduced compliance testing, improving both cost and time to market.

Specification

Processor:	Broadcom BCM2711, quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz
Memory:	1GB, 2GB, 4GB or 8GB LPDDR4 (depending on model) with on-die ECC
Connectivity:	2.4 GHz and 5.0 GHz IEEE 802.11b/g/n/ac wireless LAN, Bluetooth 5.0, BLE Gigabit Ethernet 2 × USB 3.0 ports 2 × USB 2.0 ports.
GPIO:	Standard 40-pin GPIO header (fully backwards-compatible with previous boards)
Video & sound:	2 × micro HDMI ports (up to 4Kp60 supported) 2-lane MIPI DSI display port 2-lane MIPI CSI camera port 4-pole stereo audio and composite video port
Multimedia:	H.265 (4Kp60 decode); H.264 (1080p60 decode, 1080p30 encode); OpenGL ES, 3.0 graphics
SD card support:	Micro SD card slot for loading operating system and data storage
Input power:	5V DC via USB-C connector (minimum 3A[1]) 5V DC via GPIO header (minimum 3A[1]) Power over Ethernet (PoE)–enabled (requires separate PoE HAT)
Environment:	Operating temperature 0–50°C
Compliance:	For a full list of local and regional product approvals, please visit https://www.raspberrypi.org/documentation/hardware/raspberrypi/conformity.md
Production lifetime:	The Raspberry Pi 4 Model B will remain in production until at least January 2026.

Physical Specifications



Note: all dimensions in mm

WARNINGS

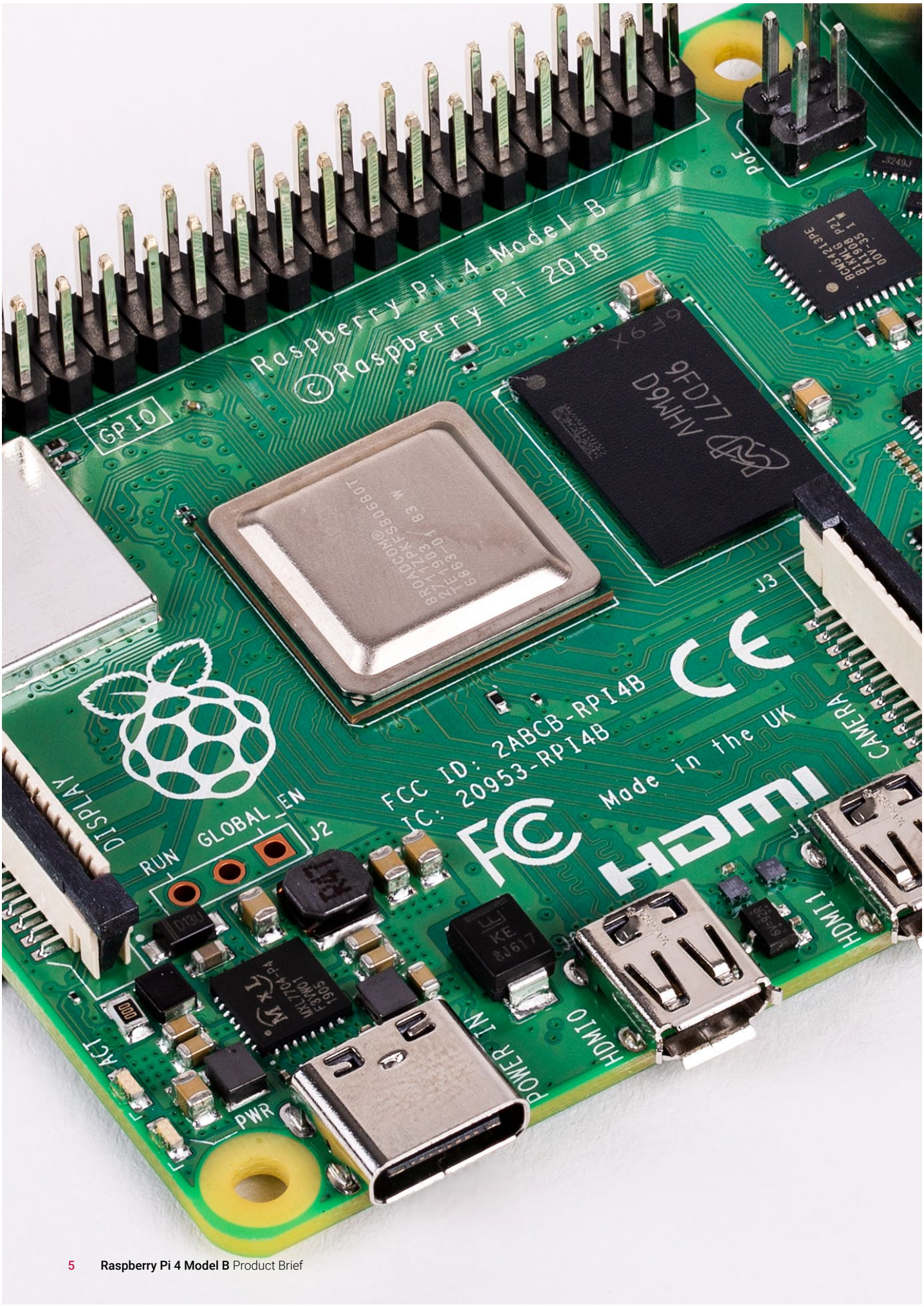
- This product should only be connected to an external power supply rated at 5V/3A DC or 5.1V/ 3A DC minimum[1]. Any external power supply used with the Raspberry Pi 4 Model B shall comply with relevant regulations and standards applicable in the country of intended use.
- This product should be operated in a well-ventilated environment and, if used inside a case, the case should not be covered.
- This product should be placed on a stable, flat, non-conductive surface in use and should not be contacted by conductive items.
- The connection of incompatible devices to the GPIO connection may affect compliance and result in damage to the unit and invalidate the warranty.
- All peripherals used with this product should comply with relevant standards for the country of use and be marked accordingly to ensure that safety and performance requirements are met. These articles include but are not limited to keyboards, monitors and mice when used in conjunction with the Raspberry Pi.
- Where peripherals are connected that do not include the cable or connector, the cable or connector must offer adequate insulation and operation in order that the relevant performance and safety requirements are met.

SAFETY INSTRUCTIONS

To avoid malfunction or damage to this product please observe the following:

- Do not expose to water, moisture or place on a conductive surface whilst in operation.
- Do not expose it to heat from any source; Raspberry Pi 4 Model B is designed for reliable operation at normal ambient room temperatures.
- Take care whilst handling to avoid mechanical or electrical damage to the printed circuit board and connectors.
- Avoid handling the printed circuit board whilst it is powered and only handle by the edges to minimise the risk of electrostatic discharge damage.

[1] A good quality 2.5A power supply can be used if downstream USB peripherals consume less than 500mA in total.


Raspberry Pi 4 Model B
©Raspberry Pi 2018
GPIO
PoE
9FD77
D9WHV
J3
FCC ID: 2ABCB-RPI4B
IC: 20953-RPI4B
Made in the UK
CAMERA
DISPLAY
RUN
GLOBAL_EN
J2
HDMI1
HDMI0
POWER IN
ACT
PWR

The Adopted Trademarks HDMI, HDMI High-Definition Multimedia Interface, and the HDMI Logo are trademarks or registered trademarks of HDMI Licensing Administrator, Inc. in the United States and other countries.

MIPI DSI and MIPI CSI are service marks of MIPI Alliance, Inc.
Raspberry Pi and the Raspberry Pi logo are trademarks of the Raspberry Pi Foundation

www.raspberrypi.org

Raspberry Pi

Appendix C

Use case template

Use case	ID / identifier of the use case
Name	Name of the use case
Initiating actor	The system participant that triggers the use case
Other actors	Other system participants involved in the use case
Brief description	A brief summary of the process
Precondition	Precondition Fact that must be ensured before the start of the use case. Preconditions are not checked in a use case, but are taken as given.
Postcondition	Postcondition Facts that are assured with the end of the use case.
Procedure	Schematic description of the individual processes and participation of the actors in these processes. In the sequence of events, only the expected normal course of events is described.
Alternatives	Deviating processes for a correct flow
Exceptions	Deviating processes resulting from an error case
Used use cases	Embedded use cases that are used, extended, or inherited by this use case.
Special requirements	Here are requirements that go beyond preconditions, which make it possible for the use case to function in the first place, as well as special requirements that the use case must fulfill (e.g., safety requirements).
Assumptions	Decisions still to be made in the context of a continued architecture definition, which must be anticipated here. If these assumptions are not met, the use case must generally be completely redesigned.
Open topics	Aspects that still need to be clarified during the realization of the use case, but are not seen as a decision of the framework architecture.
References	References to laws and other sources on which the the use case relies on or implements.
Data requirements	Data involved in the use case, as well as and the requirements for the design of the use case.
Non-functional requirements	Requirements for the use case that do not affect the (logical) flow, but have a strong influence on the realization.

Table C.1: Use case template [Mat20, p. 70-72]

Appendix D

Scope of delivery

In summary, all components of the work are listed. This ensures that Bremen University of Applied Sciences has all existing components listed and assigned.

In addition to the thesis, Bremen University of Applied Sciences will receive a memory stick with the following folder structure.

- architecture: Contains a Visual Paradigm project with all the Unified Modeling Language (UML) diagrams produced.
- expose: Contains the already submitted exposé for this Master's thesis.
- gateway: Contains the complete code for all Maven modules.
- thesis: Contains all LateX files, as well as all illustrations for the thesis.
- data: Contains the raw test data described in the evaluation chapter.

In addition to the memory stick, 3 Raspberry Pi's with the following allocation will be distributed. All Raspberry Pi's can be reached via SSH using the same username and password. The username is always **pi** and the password is **root**.

- Raspberry Pi 03 (mqtt-broker): Contains the installed MQTT broker, which is also started at system start.
- Raspberry Pi 06 (testing-environment): Contains the test environment with all available test clients installed. The test clients are in the user's home directory.
- Raspberry Pi 07 (gateway): Contains the installed gateway, Apache Karaf is stored in the home directory and can be started from there. The application is located in the local Maven directory.

Also included is the Digi Xbee 3 ZigBee Mesh Kit. The device allocation and configuration are also listed below.

- XBee 3 1 (Coordinator): Is configured as a coordinator and must be connected to the gateway.
- XBee 3 2 (Router1): Is configured as a router and can be connected to the test environment.
- XBee 3 2 (Router2): Is configured as a router and can be connected to the test environment.