Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Analysis of Transformation Capabilities between Communication Types of Cloud Application Components

Mustafa Akilli

Course of Study: INFOTECH

Examiner: Prof. Dr. Dr. h. c. Frank Leymann

Supervisor: Karoline Saatkamp, M.Sc.

Commenced: October 2, 2017

Completed: March 28, 2018

CR-Classification: C.2.0, I.6.0

Abstract

The components of a cloud application and their relationships can be defined and modelled by standards such as the Topology and Orchestration Specification for Cloud Applications (TOSCA) language to minimise the deployment, management, and portability efforts. However, changing internal or external conditions leads to updates for the model. Sometimes the components running on the same host may split into different hosts. Recent studies on splitting the components into several hosts focus on the matching capabilities with available hosting offerings in the new environment, yet the required changes regarding the communication are not considered. After splitting topologies, the new environment may not support the previously established communication. In this case, the communication between the components must be updated and transformed. The splitting process should be aborted if a transformation is not possible. In this thesis, transformation capabilities between communication types of cloud application components are analysed. The communication protocols are first categorised according to characteristics. Next, the possibility of the transformation between those communication types and the challenges and limitations are researched. Finally, a prototype that detects whether the transformation is needed when the environment changes and offers decision support for redistributing the components is implemented.

Contents

List of Figures

List of Tables

1 Introduction

Cloud computing has changed the way enterprises use information technology (IT). It reforms IT by providing convenient, on-demand, and scalable network access to cost-efficient operations [BSW14]. Investing in one's own IT infrastructures and applications and managing every step of the entire cloud application has become archaic or obsolete due to the developments and advances in cloud computing. Infrastructures, platforms, and services from providers such as Amazon, Google, Microsoft, or IBM are available. The management, maintenance, configuration, and deployment, as well as the portability of such offerings, would be considerably important regarding useful and promising cloud services. However, several parts of the application would have to be replaced or reimplemented in case the cloud provider were to change [BBKL14].

Current cloud applications are complex composite applications that are comprised of different individual components, each providing a distinct piece of functionality [BBKL14]. The unique components should be handled concerning deployment, configuration, and their communication with other components [BBLS12]. Nevertheless, management becomes tedious and error-prone if there is lack of standardisation that defines the components and their relationships [BSW14].

Several technologies, e.g. Chef and Docker and standards, e.g. TOSCA (Topology and Orchestration Specification for Cloud Applications), have been published to enable developing portable cloud applications and the automation of their deployment and management [SBKL17]. The TOSCA standard enables the description of application deployments in a portable and vendor-independent manner. It facilitates an automated deployment of cloud applications [BBKL14; OAS13]. In addition, TOSCA defines service templates that contain topology templates providing a structural description of the application by the components and relationships [BBLS12].

Using TOSCA, components and their communications can be modelled. However, sometimes the model must be adapted because of changing internal and external conditions, e.g. optimizing cost, reduction of employees, or newly available technologies. Saatkamp et al. [SBKL17] have presented a method for splitting a deployment model following a manually specified distribution on the business layer. The method also enables automatically deploying the resulting model without the need for manual intervention. However, the paper focuses on the matching of capabilities with available hosting offerings in the new environment. For this reason, the required changes concerning communication are not examined. The components running on the same host may be split to different hosts. This may even require changing the entire communication, as the new environment may not support the communications before splitting.

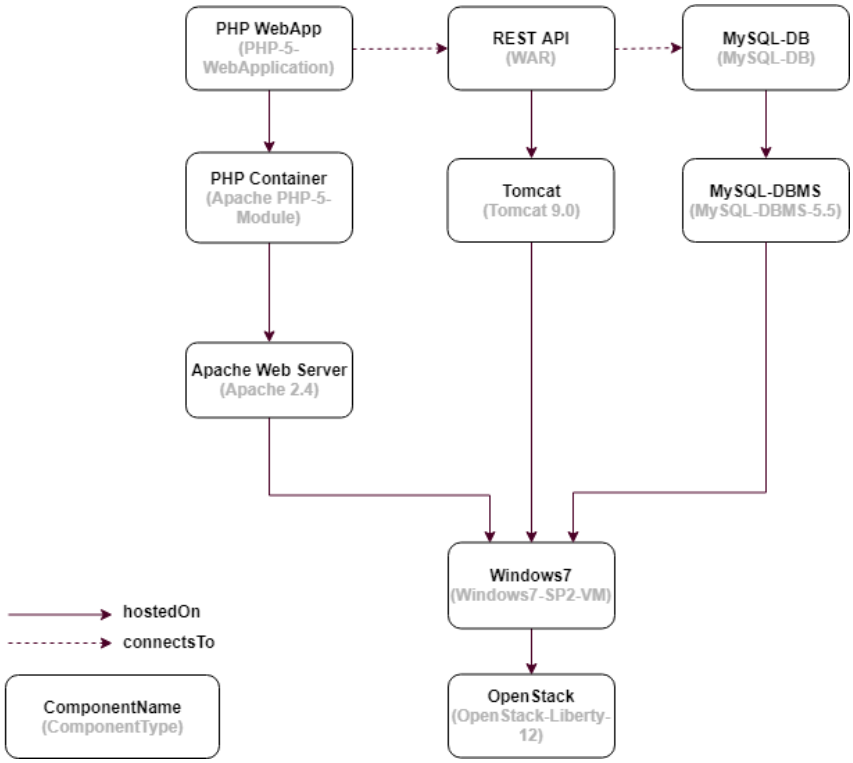

Figure 1.1: A topology model for an on-premise hypervisor (adapted from [SBKL17])

Figure 1.1 shows a topology model describing the deployment of an application. Both the PHP WebApp and the Representational State Transfer (REST) API (in Java) are on the localhost. Between the PHP WebApp and the REST API, a Hypertext Transfer Protocol (HTTP) connection shall be established. Now imagine a case in which the REST API with its Tomcat Server is split to a different environment such as a private cloud. The new environment may forbid or block the HTTP connections because of the security reasons (firewall) and may allow communication only through a gateway that has just a message broker (such as RabbitMQ). In this case, the new environment will only allow messaging such as the Advanced Message Queuing Protocol (AMQP) or Simple Text Orientated Messaging Protocol (STOMP) communication protocols: HTTP will not be permitted. In this scenario, when the topology is split, the communication between the components will no longer be available, and a transformation is needed.

This thesis aims to analyse transformation capabilities between communication types of cloud application components. The results are then used to complement the automation of the splitting and deployment process. With this objective, different communication types and technologies must be considered, and their transformation capabilities must be analysed. Furthermore, it is necessary that challenges and limitations of such transformations are examined. Finally, a prototype that detects whether the transformation is needed must be implemented.

This thesis is structured in the following way:

Chapter 2 – Fundamentals discusses the fundamentals of this thesis. This includes the messaging solutions, the Internet of things (IoT), and Internet Protocol Suite.

Chapter 3 – Related Work explains the previous research related to this thesis regarding classification of communications and transformation of communications.

Chapter 4 – Categorization and Characteristics of Communication Types categorizes the communications according to their characteristics and features.

Chapter 5 – Transformation of Communication Types analyzes the transformation capabilities of the selected communication types. This also includes the limitations and challenges of the transformations.

Chapter 6 – Implementation of Prototype for Detection of Transformation presents the prototype which detects whether the transformation is needed or not when the environment changes.

Chapter 7 – Conclusion and Future Work summarizes the contributions of this thesis and recommends related topics for future research.

2 Fundamentals

2.1 Internet Protocol Suite

According to four-layer internet architecture, the layers are, from top to bottom, 'application layer,' 'transport layer,' 'internet layer,' and 'physical layer' [Bra89].

- ***The application layer*** enables applications to have standardised data exchange and hides all the low-level details. Examples are HTTP, File Transfer Protocol (FTP), and Simple Mail Transfer Protocol (SMTP).
- ***The transport layer*** is responsible for maintaining end-to-end communications across the network. The connection of the processes is handled in this layer. The most well-known examples include the Transport Control Protocol (TCP) and User Datagram Protocol (UDP).
- ***The network layer*** (or the internet layer) provides connectivity and delivers packets for the network. The network layer protocols are the Internet Protocol (IP) and the Internet Control Message Protocol (ICMP).
- ***The physical layer*** (or link layer) provides the delivery of the data across a physical link in a network according to the above layers that define the receiver. The example protocols for this layer consists of Ethernet for local area networks (LANs) and the Address Resolution Protocol (ARP).

All the communication protocols and types examined in this thesis are higher than the transport layer, i.e. in the application layer. The layers that are below the application layer are out of the scope of this thesis. For this thesis, the only critical point regarding the lower layers is to know the differences between TCP and UDP, as they may change the characteristics of the communication type. Both UDP and TCP are transport layer protocols. While UDP supports only an unreliable channel, TCP maintains flow control, connection establishment, and reliable transmission of data. It guarantees that the packet or frame will be delivered free or errors and unchanged in a reliable channel, while the UDP does not guarantee anything, as fast delivery is the primary objective.

2.2 IoT Interaction Levels

In recent years, the IoT has gained importance and attention. This is mainly because of the advance of the technology that results in decreasing sensor and device sizes, energy

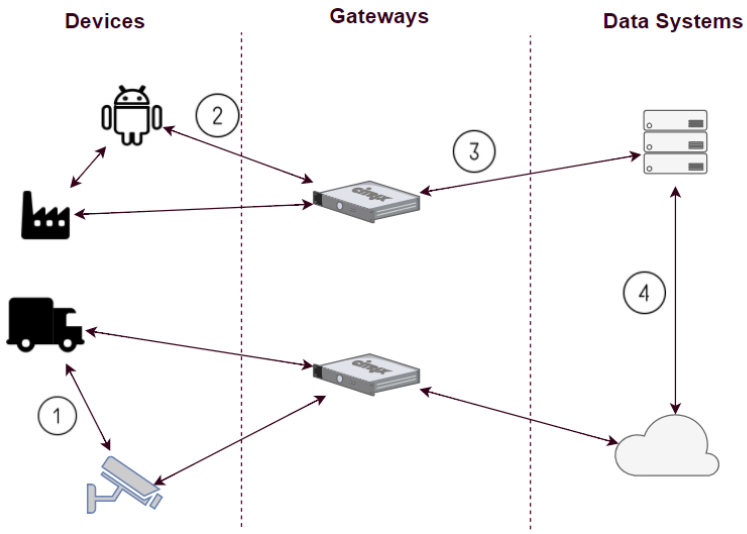

Figure 2.1: IoT communication overview (adapted from [Sol17])

consumption, or cost of chips and sensors [RBF+16]. Since IoT is a large topic itself and this thesis is about communications, only the overview of the interaction levels in the IoT is provided as background information.

In figure 2.1, an overview of the communication patterns in IoT can be seen. In an IoT architecture, there are four different kinds of connections based on the data movement: device-to-device, device-to-gateway, gateway-to-server, and server-to-server [Sol17].

Device-to-device level: Sensors or industrial devices sometimes need to send data to other devices to work together. In such situations, the communication and the information delivery should be fast and in real time since the timeliness of the data is vital. For instance, in a factory, industrial devices or sensors must work together to complete a task, such as creating the skeleton of a car. Data Distribution Service (DDS) or Constrained Application Protocol (CoAP) is the popular choice at this level.

Device-to-gateway level: Most of the sensors or devices are used for gathering information, and they need to send the data to an aggregating gateway device. The primary duty of gateways is that they can analyse, aggregate or manipulate the information coming from low-level devices and forward this data to the central servers or services. Gateways also send notifications and commands such as 'restart,' 'shut down,' or 'get information' from servers to devices. They can also send the data directly to the databases to be processed by servers later. Another usage is that gateways may be utilised as a firewall. Some devices may not have direct access from the outside, or it is not desired that they be accessed directly. In such cases, the access is done by gateways. Message Queuing Telemetry Transport (MQTT) is mainly used at this level.

Gateway-to-server: Gateway devices have no difficulties with the computing capabilities because they are way more powerful than sensors. Therefore, they can use complex protocols with reliable networks and QoS. As a result, the choice of the communication protocol does not depend on the capabilities of the device but on the QoS requirements, such as security, availability, or reliability. Since the power consumption does not limit the devices, HTTP, AMQP, or Extensible Messaging and Presence Protocol (XMPP) can be used at this level.

Server-to-server: Within the secure servers or data-centres, the selection of the communication protocol depends on the need for integration with existing applications, availability, recoverability, and throughput. An example would be a climate service transmitting messages to the news service. Here, interoperability is vital, as the services may have completely different implementations. For this reason, AMQP or HTTP can be used. In this example, the news service might be implemented in Microsoft's .NET, and the climate service could be developed in Oracle's Java environment.

2.3 Messaging

According to Hohpe and Woolf [HW04], messaging is a new way of communication that provides asynchronous, program-to-program, and reliable delivery. The authors also stated that applications, programs, or services communicate by sending frames (packets) of data (called messages) to each other. Queues or channels are the way of connecting the sender and receiver to forward messages. Queues save and deliver the messages and create all the complicated connections across multiple computers. Moreover, they can be used concurrently with various applications. A sender or producer is an application or program that generates and sends a message by pushing it to the message queue. In contrast, a receiver or consumer is a program that receives a message by reading (and deleting) it from the channel or queue [HW04].

As a result of communicating asynchronously, applications are loosely coupled in messaging systems. This also leads to more reliable communication since the two programs do not have to be running simultaneously. The programs can focus more on what they need to send instead of how they carry their data because the messaging system handles this for the applications [HW04].

2.3.1 Messaging Patterns

Messaging can be done commonly in four different ways. The interaction patterns between components are the following:

Fire-and-Forget: This type of interaction is the most basic version. The sender initiates the conversation by sending data or a message, and the recipient receives and processes the messages. A simpler interaction would be no interaction at all. For this reason, the fire-and-forget pattern is the loosest of all possible couplings among all the other patterns [HW04]. The sender does not worry about the result and replies and does not have to know something about the receiver. Since there is no conversation history between the sender and the receiver, this pattern is stateless. However, the simplicity and loose coupling nature of this model has disadvantages. Since there is no reply or feedback from the recipient, error handling is impossible. The fire-and-forget pattern must have guaranteed delivery in case messages are lost or dropped [HW04]. An example can be seen in figure 2.2.

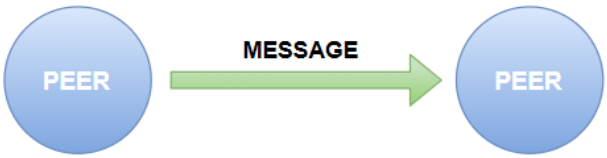

Figure 2.2: Fire and Forget pattern (adapted from [Wah16])

Request-Response: When the communication is not one way and both sides of the communication want to be involved in the conversation, then a different approach must be followed. Hohpe and Woolf [HW04] stated that request-response has two participants: the requestor dispatches a request message and expects a reply message, and the replier receives the request message and responds with a reply message. When the request and the response are synchronous, then there is no problem, as the requestor will be blocked until a reply arrives, and there will be no further processing. However, if messaging is used, due to the asynchronous nature of the conversation, the requestor should consider strategies such as 'messaging correlation ID' to keep track of the replies and match the answers and requests, as response messages may potentially arrive in a different order than the requests were sent [HW04]. In addition, there should be a separate queue for replies. An example is depicted in figure 2.3.

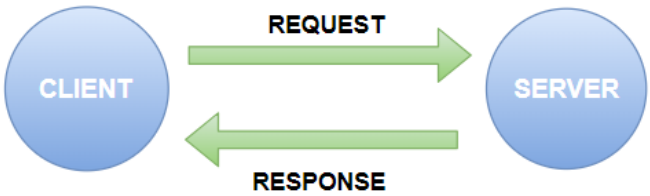

Figure 2.3: Request-Response pattern (adapted from [Wah16])

Multicasting: Sometimes devices or applications need to send the same message to many different devices or applications at the same time. However, sending messages one by one can be less efficient, and there should be a way to address the same messages to multiple points simultaneously. The most appropriate pattern is the multicasting communication pattern. In this pattern, the originator sends the messages via an intermediary node such as a router or broker. This node then shares it to multiple receivers. Since the sender does not distribute the message by itself and just sends the messages to one point, multicasting can save bandwidth dramatically. Indeed, the producer does not need to know the receivers who will acquire the messages. The most basic example of this pattern would be distributing information to many other components [Wah16]. The visualisation of this pattern can be seen in figure 2.4.

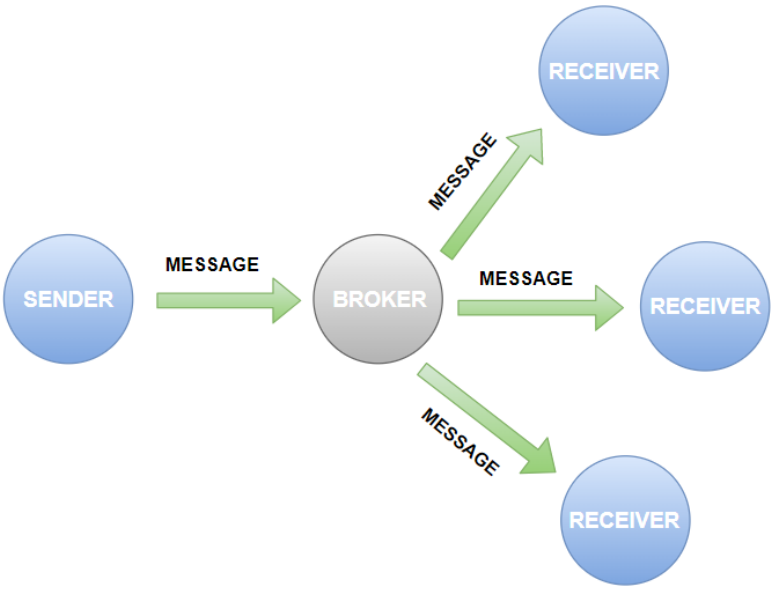

Figure 2.4: Multicasting pattern (adapted from [Wah16])

Publish-Subscribe: The need for publish-subscribe is a solution to the problem of when the application in an architecture wants to send messages to the programs that are interested in the particular topic. The publish-subscribe communication pattern is the modified version of the multicasting pattern. The only and core difference is that messages are also saved on the intermediary node [Wah16]. The messages will be forwarded automatically to the subscribers of that channel or queue. There are options to set such items as the latest message is stored, the last N (a number defined by configuration) messages stored, or all the messages are stored. Publishers and subscribers are loosely coupled to each other, and they do not have to aware of each other. An example of this pattern is demonstrated in figure 2.5.

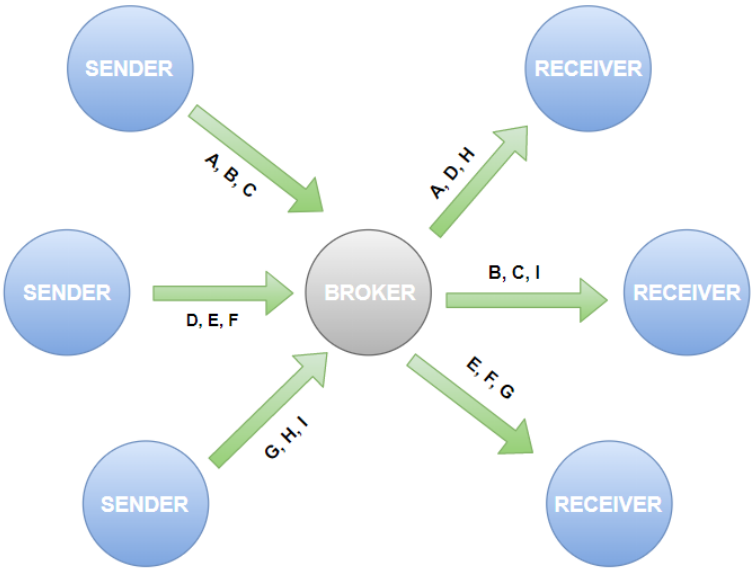

Figure 2.5: Publish-Subscribe pattern (adapted from [Wah16])

2.3.2 Messaging Middleware

'Messaging middleware' is also known as 'messaging servers' or 'message providers.' The implementations of messaging servers generally include 'message queue,' 'message broker,' and 'message router.' These concepts can all be used interchangeably.

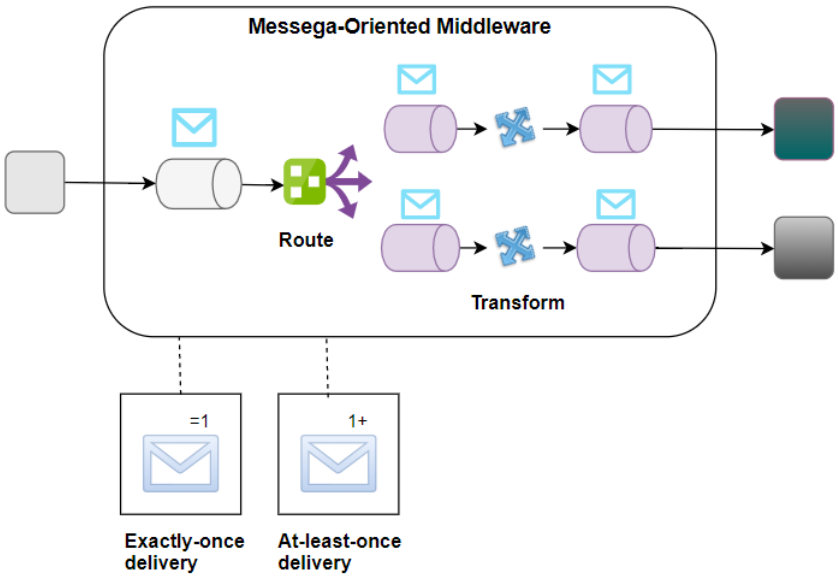

Figure 2.6: Messaging-oriented middleware (adapted from [FLR+14])

Messaging middleware is software or hardware infrastructure that offers the management of the message transfer and traffic between distributed systems. Message-oriented middleware (MOM) supports application and programs that are developed on different platforms and eliminates the intricacy and difficulty of implementing software that can communicate with different operating systems and network protocols. In this way, the complexity of the underlying layers, such as operating systems and network protocols, can be hidden by MOM, and a more straightforward interface can be provided to the application developers [FLR+14; Hop03]. In figure 2.6, a general overview of the MOM is shown.

The main function of middleware is to obtain incoming messages from different software applications and platforms and then forward and distribute those messages to desired locations. The following points are the other important operations that can be performed by MOM: [Cur04; HW04]

- transforming messages to a different format;
- collaborating with external services and databases to extend a message or store it;
- operating message aggregation, breaking down messages into small frames messages and sending them to their recipients, and then merging the responses into one message to be returned to the requestor;
- executing or invoking web services to obtain information;
- offering content- and topic-based message routing using the publish-subscribe pattern;
- and handling of errors and events.

A messaging queue offers an asynchronous communication protocol, i.e. the originator of the message and the consumer of the message do not have to contact with the message queue at the same time. Messages placed into the queue are saved until the recipient retrieves them. Message queues can be configurable. The configurable properties may include the message maximum data size that can be sent in a message or the number of messages that can remain or be stored [HW04].

A message broker is an intermediary application that converts the message from one format (sender) to the other message format (receiver). A message broker is an architectural model for message verification and validation, transformation, and routing. It negotiates communication among applications, reduces the mutual awareness that applications have to exchange messages correctly and efficiently, and causes them to be loosely coupled [HW04].

A message router finds the desired destinations to forward messages based on given predicates and configurations. It also decouples the destination of a message from the originator and provides central or main control over message traffic [HW04].

2.3.3 Messaging APIs

Messaging APIs are only a standard or interface; they do not consider the communication protocol and how the communication is performed. They are just APIs to form messages. The interoperability of messaging APIs is limited to their environment. They need a message queue (MQ) implementation or provider to work. These MQs can be external or built in.

The most well-known and popular messaging technologies are Java Messaging Service (JMS) by Oracle and .NET Messaging Service (NMS) by Microsoft. Both JMS and NMS can support point-to-point (fire-and-forget or request-response) and publish-subscribe patterns. JMS is designed to operate in Oracle's Java applications, and NMS is intended for Microsoft's .NET applications. Unless a message service provider or MQ is specified in the configuration, they use their built-in MQs. For JMS, the default MQ is OpenMQ, which is integrated with the GlassFish application server by Oracle, and for NMS, it is MSMQ, included in the .NET framework.

ActiveMQ is not an Oracle product, but it is a native provider for JMS. One can write normal JMS application, and the message queue can be specified in the configuration files. If MQ is not specified explicitly, then Oracle's OpenMQ (default) will be used, and the communication protocol (also called wire-level protocol, or application-level protocol, or transport protocol) will be Oracle's specific protocol. If the ActiveMQ is used instead (if installed), then the communication protocol will be ActiveMQ's specific protocol, which is OpenWire, unless it is explicitly set to use other supported communication protocol by ActiveMQ (it can support AMQP and OpenWire).

To form messages, client libraries are needed. Client libraries are only needed for messaging APIs such as NMS or JMS but also needed by AMQP, STOMP, or other kinds of protocols to make their packets. These client libraries can be provided directly by the founder team of the protocol, can be developed by third parties, or can be provided by their messaging queues.

For instance, to use RabbitMQ, the client libraries can be downloaded and used to form messages. In Java, instead of importing JMS libraries, RabbitMQ libraries can be imported and used for messaging. In this way, the communication protocol will be the native protocol of RabbitMQ, which is AMQP instead of OpenWire or other protocols. If the developer wants to use JMS over RabbitMQ which has a JMS support, then he or she does not need to include the RabbitMQ library. The developer will use the JMS library from the Java platform, and in the configuration files, the message queue needs to be set to RabbitMQ. The communication protocol will then be the native protocol of RabbitMQ, which is AMQP, unless a different protocol is specified since RabbitMQ can support other protocols. In a similar manner, if a develop wants to use RabbitMQ in .NET Apps, then RabbitMQ client libraries will be included in the C# code, and messaging can then be used via specified transport protocols (by default AMQP for RabbitMQ).

However, for STOMP, for instance, there is no need to use the message queue client libraries because there are online external libraries for most of the languages. The libraries can be included, and messages can be formed by using these external libraries, and message

queues for STOMP can then be used as middleware only. For example, HornetQ has only Java client and has support for STOMP. To use STOMP messaging with HornetQ in Python language, 'STOMP.py' can be downloaded and imported in the Python application. HornetQ will then be used only for message middleware, a server, or a broker.

An overview of the message brokers is given in section 4.2.1. In that section, the information about the supported protocols and client libraries of message broker implementation can be found.

3 Related Work

This thesis has reviewed previous work in three different categories: the characteristics of the communication types, the classification of them and the transformation between them. The related work for the characteristics of the communications includes mainly the official documents of the respective communication protocol. That is why there will be no section for official documents, and those papers are not discussed here. Individual references can be found in chapter 4. In the classification and comparison of the communications section 3.1, the papers discuss the different advantages and disadvantages, and a comparison of the protocols is provided. In section 3.2, the articles that address how to transform communication protocols and communication patterns are explained.

3.1 Classification and Comparison of the Communication Types

The papers explained in this section are focus on the survey and the comparison of the communication types regarding advantages, disadvantages, performance, and scope. Some papers compare the low-level layers of the communication protocols more, while other papers only consider the most important three to four communication protocols. However, no paper compares and classifies almost all of the most common application layer protocols as was done in this work. Nevertheless, all the papers together provide a sufficient overview of the differences of communication protocols.

Karagiannis et al. [KCVA15] created a survey of application layer protocols of the IoT. The survey offers a comparison of MQTT, XMPP, AMQP, HTTP, and CoAP. Since there are not so many papers that review possible alternatives of communication protocols with pros and cons, the work of Karagiannis et al. [KCVA15] is useful compared to the other surveys. The authors included protocols from different levels of an IoT scenario. In addition, they argued the suitability of these protocols for the IoT by considering reliability, security, and energy consumption aspects. Yassein, Shatnawi, and Al-Zoubi [YSA16] also have a large survey of application-level protocols. They have included DDS and WebSocket protocols in addition to those in the paper by Karagiannis et al. [KCVA15]. They made a comparison among these protocols based on transport layer used, architecture, communication model, security and quality of service. They have also addressed the weaknesses and strengths of each protocol. Because it is a comprehensive comparison between the existing protocols, it is helpful to select the suitable protocol for the existing environments and applications for the developers and researchers. Thota and Kim [TK17] similarly discussed and analysed the

advantages, disadvantages, efficiency, usage, and requirements of the IoT communication models. However, their paper only considers two major protocols, namely CoAP and MQTT. Thota and Kim [TK17] thought that these protocols are lightweight concerning the operation and data transfer and that they can therefore fit many IoT environments. Their research includes the outcomes, anomalies, and benefits of each of these protocols. However, Al-Fuqaha, Guizani, and Mohammadi [AGM15] have provided a more detailed comparison of the popular communication protocols. Their work contains the AMQP, DDS, XMPP, CoAP, and MQTT protocols. Al-Fuqaha, Guizani, and Mohammadi [AGM15] not only compared these protocols but also explained the relationship between communication protocols and different engineering disciplines, such as big data analysis. They also added example use cases of the protocols. Richards [Ric11] compared AMQP with JMS and explained the place and importance of the AMQP among messaging technologies and messaging APIs. In the article, he provided examples to demonstrate the importance of interoperability and AMQP. Schneider [Sch13a] also compared XMPP, MQTT, DDS and AMQP, explaining the features and characteristics of these protocols.

Although many papers have compared the characteristics and peculiarities of the communication protocols, several papers have compared the protocols concerning their efficiency. Sasirekha et al. [SSCK17] compared the performance of CoAP, HTTP, and MQTT. The authors compared the efficiency in terms of response time, latency, and throughput. In the same way, the paper by Mijovic, Shehu, and Buratti [MSB16] compared the application layer protocols based on their efficiency and performance. They compared CoAP, WebSocket, and MQTT. The performance, regarding protocol efficiency and average round-trip time (RTT), was evaluated experimentally. The authors also assessed the same scenarios with various configurations and settings: first with a LAN configuration in which the access point (AP) and the server were on the same LAN and then with a more realistic IoT configuration in which the AP was connected to a remote server via the Internet. Fysarakis et al. [FAS+16] compared the efficiency and performance of machine-to-machine (M2M) protocols such as CoAP, MQTT, and Extensible Access Control Markup Language (XACML). They evaluated the selected protocols in the context of designing and implementing an application that requires various M2M interactions, such as a policy-based access control framework for IoT devices. The authors' evaluation considered each protocol's intrinsic characteristics, features, and performance.

Some papers compare and classify the communication protocols according to their IoT level. These IoT levels are device-to-device, device-to-gateway, gateway-to-server, and server-to-server, as mentioned in chapter 2. For instance, Sethi and Sarangi [SS17] researched in a more general way and provided general information about the IoT and its layers, protocols, and technologies. Compared to similar survey papers in IoT communication protocols, the paper made more comprehensive and detailed work in its coverage and exhaustively covered most major protocols and technologies. The book by Rayes and Salam [RS16] comprehensively described an end-to-end IoT architecture. The authors explained the architecture comprised of devices, network, computer, storage, platform, and applications along with management and security components. In chapter 2 of their book, they provided a sufficient overview of protocol layers from the lower layers (such as IP)

to the application layer and classified the IoT protocols in the layers. FocusGroup [Foc14] from the International Telecommunication Union (ITU)[1] also published a technical report. In their report, they described the IoT layers and protocols and provided examples. The group also analysed the API and protocol requirements for the M2M service layer. Salman and Jain [SJ16] created a detailed overview of all the layers of communication protocols. In their comprehensive survey of protocols for IoT, they explained datalink layer, network routing standards, the network encapsulation layer, the session layer, and management standards for the IoT.

In addition, there are papers that compare and classify the communication protocols regarding lower layers or layers closer to the hardware level. Patil and Lahudkar [PL16] further researched the MAC layer issues of the IoT protocols. Although the paper is about the MAC layer issues, the authors also provided a comparison of MQTT, AMQP, HTTP, and CoAP. However, the overview was more about efficiency and communication channels of the protocols. Weyrich, Schmidt, and Ebert [WSE14] also made a comparison of the lower levels of the M2M and IoT communication protocols and offered an overview of the lowest layer of the M2M and IoT protocols. These lowest layers include the technologies such as LTE, WLAN, Bluetooth, Zigbee, RFID, and NFC.

3.2 Transformation Between Communication Types

The papers provided in this section are generally about how to convert from one specific protocol to the other one or the conversion of communication patterns, such as request-reply to publish-subscribe. The main difference of the papers in this section compared to the work of this thesis is that they do not have an overview of the transformation of the all-important application layer communication protocols. Moreover, the papers do not include information about the possible losses, challenges, and difficulties of the transformation.

The related works regarding the transformation between communication types can be divided into two categories: (1) papers that analyse the conversion between two specific communication protocols such as MQTT to HTTP and (2) articles that research the transformation of the communication patterns such as request-response to publish-subscribe.

The official guideline from Internet Engineering Task Force (IETF)[2] by Castellani, Loreto, and Rahman [CLR17] describes how to map between CoAP and HTTP in detail. It is a comprehensive manual and guideline for the developers who need conversions between CoAP and HTTP, because the IETF defines the standards of both protocols. Castellani, Fossati, and Loreto [CFL12] also analysed the HTTP-CoAP conversion in their paper. They discussed how to map CoAP to HTTP and what the currently open issues are, especially in the deployment and security areas. There are also HTTP-CoAP proxies that are developed by different groups and organisations, such as DEI-Telecommunications Group[3]. Another

[1] International Telecommunication Union - https://www.itu.int/en/Pages/default.aspx
[2] Internet Engineering Task Force (IETF) - https://www.ietf.org/
[3] Proxy developed by DEI-Telecommunications Group - http://telecom.dei.unipd.it/iot

example is jCoAP[4], a Java Library that implements the CoAP. Developers of jCoAP has also introduced their CoAP-HTTP proxy.

Some of the proxies have been built as part of frameworks with a wider scope. For example, Californium[5] provides the central framework with the protocol implementation to create the IoT applications, and they have their CoAP-HTTP proxy.

Vasters [Vas17] explained how to create conversions between MQTT and AMQP. He first gave an overview of both protocols and introduced their header fields and then highlighted their similarities and differences. In addition, Vasters [Vas17] stated why such conversion is needed. After this introductory section, he described how to perform the transformation from MQTT to AMQP and AMQP to MQTT. However, in his paper, he did not provide an overview of the difficulties and limitations of such transformations. He just explained the process of conversion, i.e. how to map incoming MQTT packets to AMQP packets or vice versa. Vasters [Vas17] also did not evaluate such transformation from the characteristics point of view. For example, AMQP has more features than MQTT has. He did not explain how to transform those missing features in MQTT when transforming AMQP to MQTT.

Desai, Sheth, and Anantharam [DSA14] proposed and implemented the Semantic Gateway as Service (SGS), which provides transformation between common messaging protocols (XMPP, CoAP, and MQTT) via a multi-protocol proxy architecture. In their paper, they analysed IoT architecture to ensure interoperability between systems, which utilises established communication and data standards. They explained first why such transformations are needed because of different and various IoT scenarios. After that, they introduced their gateway implementation, which can receive different types of messages such as XMPP, MQTT or CoAP and then performs the necessary conversions and finally outputs the message in the desired protocol. In their system, there are three main components. The chief component is 'multi-protocol proxy,' which receives input messages. This component has all the necessary brokers or message servers. By the help of the 'message store' and 'topic router' components, the necessary subscriptions can be done. These components will also publish to the subscribed topics if there is new information or message, and the saved message can be retrieved. Although they explain how this gateway works in a high-level way, the authors did not fully explain how to make such conversions and transformations, especially the transformations that involve XMPP. In their paper, CoAP to MQTT conversion is illustrated with examples as well as how to transform from request-response to publish-subscribe. However, there are no transformation details about XMPP. The authors also did not analyse the possible losses, limitations, and challenges.

Luoto and Systä [LS18] described how a system designed with RESTful architecture could be reimplemented by using MQTT. They analysed how designs based on Internet architectures can be implemented once more on top of MQTT. In their paper, they first showed how to map concepts of HTTP to MQTT, stating that there could be many different ways to implement those concepts using MQTT. They provided examples of two different ways

[4]jCoAP - https://code.google.com/archive/p/jcoap/
[5]Californium - http://people.inf.ethz.ch/mkovatsc/californium.php

and selected one of them to show. In contrast to other articles mentioned above, Luoto and Systä [LS18] explained how to transform from HTTP to MQTT thoroughly. Furthermore, they demonstrated not only packet-wise mapping but also concept-wise mapping. They showed how publish-subscribe could be implemented using the request-response pattern. The authors described the difficulties and challenges for this transformation, but they did not include limitations.

There are also papers regarding mapping or transforming the concepts of the protocols rather than analysing the transformation between two specific protocols. For instance, the paper by Cugola, Migliavacca, and Monguzzi [CMM07] researched adding replies to the publish-subscribe pattern. The authors attempted to implement the request-response pattern using publish-subscribe. The main focus of Cugola, Migliavacca, and Monguzzi [CMM07] was to minimise the impact on the positive characteristics of the publish-subscribe model, and their scope included distributed applications. In a similar manner, Heimbigner [Hei00] researched how to implement query-advertise using publish-subscribe. Query-advertise is a system that has three main concepts: advertisements, queries, and responses. The system is used for a peer-to-peer search engine. Since query-advertise is similar to the request-response pattern and obviously needs a response, the research results and the stated algorithm can be used for implementations of request-response using publish-subscribe. In the article by Eugster et al. [EFGK03], the different types of the publish-subscribe pattern are explained. The authors did not include any transformation between those types. However, the research can be a basis for implementing publish-subscribe using different patterns because the authors described the main characteristics and common denominator underlying the different types of publish-subscribe patterns.

4 Categorization and Characteristics of Communication Types

In this chapter, communication methods and technologies are explained. The communication types are categorised according to their characteristics. There are many ways to classify them, as a communication type can have multiple characteristics. There is consequently no clear separation of communication types. However, there is one characteristic that differentiates communication types. This thesis categorises communication types into two main categories, namely one-directional or bidirectional communication protocols. One-directional communication types are naturally synchronous and can be made asynchronous by extension. A request can start the conversation, and the other side can only send a response to that request. In contrast, bidirectional systems are naturally asynchronous. Both sides of the communication can send a message anytime to each other. These two categories are divided into subcategories.

4.1 One-directional Communication

One-directional communication is one of the basic methods of communication in the world of computing. One computer or device sends requests, and the other sends back the response to the request. It is mainly used in client-server (or master-slave) types of architecture where clients submit their requests and servers replies [HW04]. According to Waher [Wah16], client-server type is used in the well-known communication type HTTP and is the basis for service-oriented architecture, web services, and REST. For a client-server or master-slave structure, this model is the right decision. Waher [Wah16] also stated that one drawback of this communication type is that it is difficult to implement when both sides want to request information from each other. In this case, it is difficult to define the server and client clearly. If the sensor is the client and the middleware is a server, then the sensor can send its data anytime. However, the middleware cannot obtain information when it wants. If the sensor is selected as the server and the middleware as the client, the middleware can receive data send requests and obtain the data whenever it needs to, but the sensor might not send its report when necessary [Wah16]. Furthermore, in case of failure, it is the responsibility of the client to detect the failure and send the request again. Since this type is naturally synchronous and the requests are blocking requests, it has different problems too. Rodríguez-Domínguez et al. [RBN+12] stated that this is not well suited for mobility support because the coupling between senders and receivers may

lead to unexpected and undesirable situations. For example, a server may no longer be accessible in a system, and makes requesters blocked forever while waiting for replies.

The most well-known communication methods of this type are HTTP and Remote Rrocedure Call (RPC). While this kind of communication is naturally synchronous, ModBus is an asynchronous type, as it supports message correlation and the requests do not blocking requests. The HTTP can be made asynchronous using call-back functions, but RPC works only in synchronous mode, meaning that the requests will always be blocked. However, if the programming language supports threads, the process can be completed in a separate thread so that it is blocked but the main thread is not. Nevertheless, this does not change the synchronous characteristic of RPC type.

4.1.1 HTTP and REST

Hypertext Transfer Protocol is an application layer protocol exchanging or transferring contents, such as hypertext, in a request-response pattern [FR14]. The communication is established over TCP/IP connections with the default port TCP 80. When a browser needs resources and information from a web server, the browser sends an HTTP request to the server. Then the server replies with an HTTP response, as seen in figure 4.1.

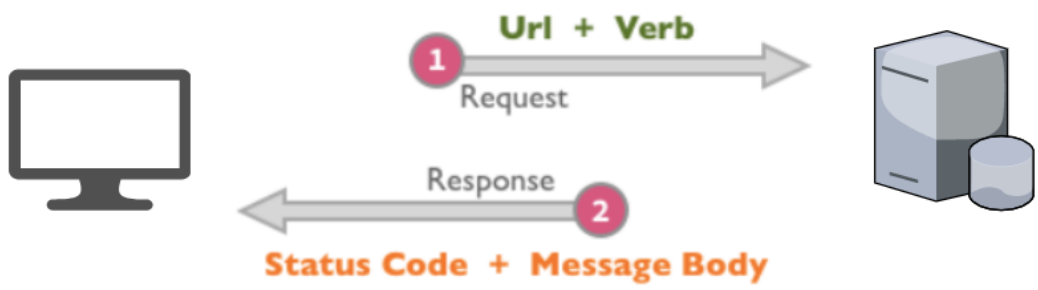

Figure 4.1: HTTP communication (adapted from [Pod13])

When HTTP is used to communicate between web servers and browsers, it can have features such as 'cookies' and 'caching.' It also supports many different data types. As long as both sides know, any data type of any size can be sent: XML, JSON, binary, image, HTML, et cetera. The most significant advantage of HTTP is that it is interoperable, cross platform, cross language, and ubiquitous. This means that everybody can speak HTTP, and it is easier to develop a Web service on top of HTTP, and existing Web servers can be configured to send specific HTTP requests to the engine [Tho01]. Although HTTP has such features, there are also disadvantages. The communication is not bidirectional, and it is synchronous. These drawbacks can be eliminated by using HTTP2 and WebSocket, which are discussed later. Hypertext Transfer Protocol Secure (HTTPS) encrypts the HTTP exchange to add

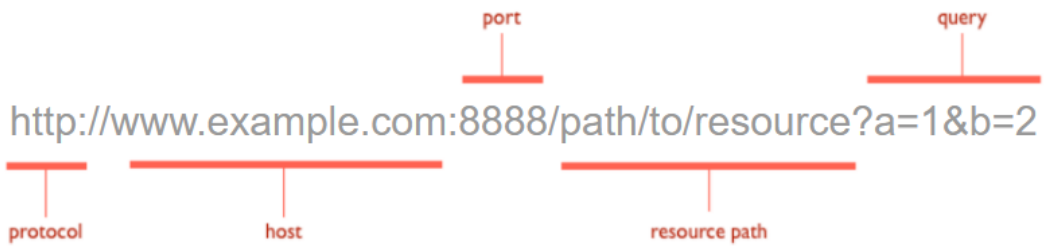

Figure 4.2: Example URI (adapted from [Pod13])

important security using TSL(SSL) and provides a secure, encrypted data transport layer. Switching from HTTP to HTTPS is easy and requires no significant changes [GT02].

However, HTTP is not employed only between web servers and browsers, although this is how it was meant to be used. Hypertext Transfer Protocol can also be utilised to message passing. For instance, SOAP messages between web services are mostly carried by HTTP. The SOAP message, in XML, is format written into the payload of the HTTP.

Hypertext Transfer Protocol defines four basic operations (verbs) that include 'GET,' 'POST,' 'PUT,' and 'DELETE' [Tho01]. These are the essential parts of the REST developed by Fielding [Fie00]. Representational State Transfer is an approach that leverages HTTP and is not a substitute to it [RA13]. This approach means that the main item in the architecture should be the resource. Create, read, update, and delete (CRUD) operations should be performed for each action applied to a resource. Four verbs of HTTP are convenient to use as CRUD operations (POST for create, GET for read, PUT for update, and DELETE for delete) [Mas11]. In REST design, every verb or operation is used with a Uniform Resource Identifier (URI), which identifies the resource. An example can be seen in figure 4.2.

4.1.2 Remote Procedure Call (RPC)

The client triggers an RPC by sending requests to a known remote server to perform a specific procedure with given parameters. The remote server then replies a response to the client, and the client application can continue to process. The requests are blocking requests; that is why while the server is executing the operation, the client application is blocked [BN84]. There are many RPC implementations. Some RPC technologies use HTTP as underlying transport protocols, some of them use socket programming or their protocols.

- ***XML-RPC or JSON-RPC*** is an RPC protocol that uses XML (or JSON) syntax in messages, and HTTP is used as a transport protocol.
- ***Microsoft .NET Remoting*** offers RPC facilities for distributed systems that run in the Windows environment. WCF has taken its place.

- ***Java Remote Method Invocation (RMI)*** is an API, which implements RPC in Java with object-oriented programming. Object references can be used for efficiency. Passing object reference is more useful than to passing complete large objects. This API does not use HTTP but instead uses remote references layers, stub, and skeletons on top of the TCP layer.

According to Hohpe and Woolf [HW04], it may seem natural to use RPCs since they look like calling local functions. However, they do not perform the same. First, RPCs are slower and can fail because of network connections. It is an undesirable situation when multiple applications communicate over an enterprise, and an entire system is down by a failure of one application. In addition, the design may have problems, as each application should know details of the other applications, even if this is only details about their interfaces. These are not the only disadvantages of RPC. Kukreja and Garg [KG14] stated that in case the procedure that was initially built to run locally is called remotely, may have references to the global variable, this would fail, and method would not work. Such a problem is the same as that of the pointer variables and is as difficult to handle. Moreover, exception handling is not same as local calls. Another disadvantage of RPC is firewalls. By default, most of the firewalls enable port 80 for HTTP requests. However, RPC calls may be blocked because of firewalls and require further firewall configurations.

4.1.3 ModBus

Modbus Protocol is a messaging system developed by Modicon in 1979. It is used to settle communication between master-slave (client-server) industrial devices. It has been used and implemented by many vendors on various devices to transfer and register data between control devices [Mod04]. Modbus is used in multiple master-slave applications to monitor and program devices, to communicate between industrial devices and sensors, and to monitor field devices using PCs and Human-machine interactions. Modbus is not only an industrial protocol; infrastructure, transportation, and energy applications also make use of its assets [Mod04]. Although Modbus is used mainly in industry and is out of the scope of this thesis, it is included to show different communication protocols and diversity.

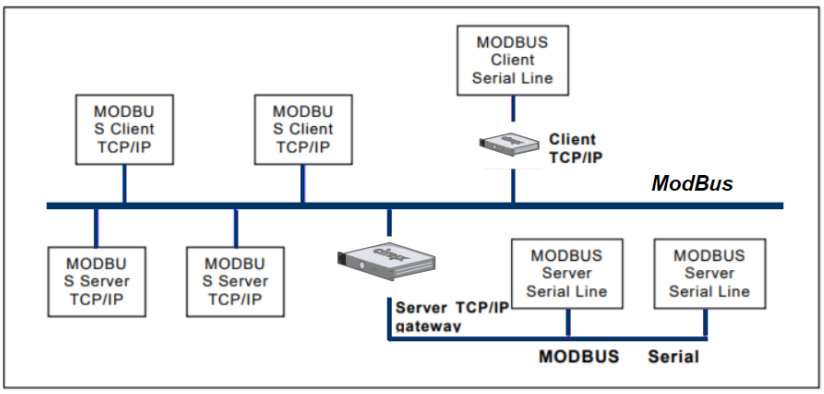

Figure 4.3: Modbus overview (adapted from [Mod06])

The original Modbus was a serial communication that used the RS232 port in the devices. After TCP/IP became a widely used protocol, the TCP/IP variant of Modbus was developed [Mod06]. In figure 4.3, devices using the original Modbus version and devices using the TCP/IP variant can be seen.

Modbus, unlike others, does not need to be synchronous. It has message correlation, and the requests are not blocking. When the master device polls the slave, if the slave cannot answer, it replies with an empty acknowledge (ack) so that master device is not blocked. By checking using message correlation, the master can then understand which response is for which request.

In the figures 4.4 and 4.5, the original header of the Modbus protocol and the header of Modbus TCP/IP can be seen. The first header has checksum, but there is no error checksum in the second header since this already included in the lower layer, that is the TCP layer.

The MODBUS Application Protocol header (MBAP) has four fields. They are the 'Transaction Identifier' (for synchronisation between messages of server and client), 'Protocol Identifier' (0 for Modbus/TCP), 'Length Field' (number of following bytes), and 'Unit Identifier' (identification of a remote slave connected on a serial line or other buses). Supported functionalities are basic functionalities for industrial devices. There are about 16 different commands, and they are mainly read/write device or read/write registers and records commands. Detailed information regarding supported function codes can be found here [Mod04].

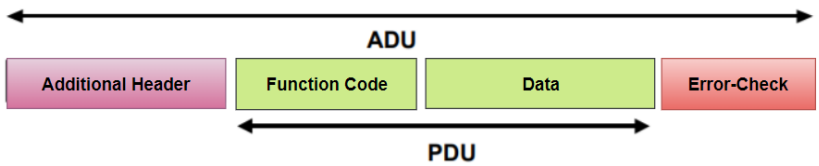

Figure 4.4: Modbus RS232 (original) message (adapted from [Mod06])

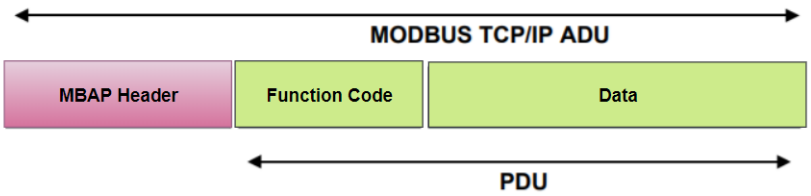

Figure 4.5: Modbus TCP/IP message (adapted from [Mod06])

4.2 Bidirectional Communication

In this communication type, both parties can send messages, and the requests are not blocking; thus, they are naturally asynchronous. These communication types are further divided into two groups. Some communication types in this section require messaging middleware to work; however, others require no middleware.

4.2.1 Communication Protocols with Middleware

Message Queue	Languages/Platforms (or Client Libraries)	Supported WireLevel Protocols (or Application Transport protocol)	Native Protocol
RabbitMQ	Almost all	AMQP, STOMP, MQTT, HTTP, WebSocket	AMQP
ActiveMQ	Almost all	OpenWire, AMQP, STOMP, MQTT, XMPP, HTPP, WebSocket	OpenWire
Apache Qpid	Java, C/C++, Python	AMQP	AMQP
IBM MQ	C, C++, Visual Basic, Java, Ruby, Python	HTTP, AMQP, MQTT	MQ-Specific
HornetQ	Java	AMQP, STOMP, HTTP, OpenWire	MQ-Specific
OpenMQ	Java, C	STOMP, HTTP	MQ-Specific
MSMQ	.NET	-	MQ-Specific
Amazon SQS	Almost all	-	MQ-Specific
Gearman	Almost all	-	Gearman
IronMQ	Almost all	HTTP	HTTP
Mosquitto	C, Python	MQTT	MQTT
HiveMQ	Almost all	MQTT, WebSockets	MQTT
ZeroMQ	Almost all	ZMTP	ZMTP

Table 4.1: Message queue comparison and overview

In this section, widely and commonly used communication protocols that work with the middleware are explained. The selected protocols include AMQP, MQTT, STOMP, and XMPP. There are also other popular protocols such as WAMP[1], ZeroMQ (ZMTP)[2], and OpenWire[3]. OpenWire is the native protocol for ActiveMQ; however, it is similar to AMQP and is rarely used except for ActiveMQ. That is why it is not considered in chapter 5. ZeroMQ and WAMP are not widely used as the other protocols, so they are not considered either.

Before detailing the messaging protocols, an overview and comparison of the message queue implementations can be seen in table 4.1. The supported client libraries, supported protocols, and their native protocols can be seen. In the entries, there are two terms.

[1]The Web Application Messaging Protocol (WAMP) - http://wamp-proto.org/
[2]ZeroMQ Transport Protocol (ZMTP) - http://zmtp.org/page:read-the-docs
[3]OpenWire - http://activemq.apache.org/openwire.html

'Almost all' means client libraries that are from many different and widely used languages and platforms are officially supported. The other term 'MQ-Specific' means that the native protocol of the message queue is not public and specifically used only in that message queue (MQ).

In the table 4.1, the most well-known and common message queue implementations are used. They are RabbitMQ[4], ActiveMQ[5], Apache Qpid[6], IBM MQ[7], HornetQ[8], OpenMQ[9], MSMQ[10], Amazon SQS[11], Gearman[12], IronMQ[13], Mosquitto[14], HiveMQ[15], and ZeroMQ[16].

AMQP

The Advanced Message Queuing Protocol is an application layer communication protocol for messaging systems from OASIS[17]. The essential features of the AMQP are message orientation, queuing, routing (including point-to-point and publish-subscribe), reliability, and security [OHa07]. The Advanced Message Queuing Protocol is a general-purpose message transfer protocol applicable and convenient for many different types of messaging-middleware infrastructures and also for point-to-point message sending. It is a bidirectional protocol that allows each party on a current connection to launch transfers and has rich extensibility and annotation features at practically all layers [Vas17]. This protocol is mainly used for server-to-server or gateway-to-server interactions rather than device-to-device or device-to-gateway interactions since the servers or the gateway devices are computational enough and processing the AMQP requires more CPU power [Sol17]. AMQP is mostly used in business messaging. From the IoT point of view, the AMQP is most appropriate for the control plane or server-based analysis functions [Sch13a].

Vinoski [Vin06] stated that the primary task of the AMQP is to make the implementations from different vendors interoperable in the same way as SMTP, HTTP, FTP, et cetera but in a messaging context. The earlier standards of middleware were only at the API level (e.g. JMS) and were focused on standardising interfaces for the programmers with different middleware implementations rather than offering interoperability between different middleware implementations. Vinoski [Vin06] also explained that unlike JMS,

[4]RabbitMQ - https://www.rabbitmq.com/
[5]ActiveMQ - http://activemq.apache.org/
[6]Apache Qpid - https://qpid.apache.org/
[7]IBM MQ - https://www-03.ibm.com/software/products/en/ibm-mq/
[8]HornetQ - http://hornetq.jboss.org/
[9]OpenMQ - https://javaee.github.io/openmq/
[10]MSMQ - https://msdn.microsoft.com/en-us/library/ms711472(v=vs.85).aspx/
[11]Amazon SQS - https://aws.amazon.com/sqs/
[12]Gearman - http://gearman.org/
[13]IronMQ - https://www.iron.io/mq
[14]Mosquitto - https://mosquitto.org/
[15]HiveMQ - https://www.hivemq.com/
[16]ZeroMQ - http://zeromq.org/
[17]Advancing open standards for the information society (OASIS) - https://www.oasis-open.org/

which represents an API and a set of behaviours that a messaging implementation must provide. The Advanced Message Queuing Protocol is a wire-level protocol. A wire-level protocol is a definition of the format of the message or the data that is sent over the network (over the wire) as a stream of bytes. For this reason, any software that can generate and interpret messages that conform to this message format can interoperate with any other compliant software, regardless of implementation language.

According to Al-Fuqaha, Guizani, and Mohammadi [AGM15], AMQP requires a reliable transport protocol (TCP/IP) to exchange messages. It supports reliable communication via a message delivery guarantee that is three-level QoS (at-most-once, at-least-once and exactly once delivery). Because of its advanced frames and headers, the AMQP offers many different features, and its QoS policies are not restricted to only three-level QoS. It can provide message filtering, the priority of messages, data, or queue timeliness. Furthermore, it offers many different updates regarding the delivery state of the message, such as that it is 'modified,' 'accepted,' or 'rejected.' It can also provide detailed information about the errors [OAS12].

The Advanced Message Queuing Protocol has two main layers: transport and messaging. The transport layer handles all the negotiations such as version agreement, connections, security, and message transfers. The message layer is the actual message with long headers. The transport layer has various frames, including 'begin,' 'end,' and 'attach,' and each of them has many different headers. All the frame types and headers are available in the official AMQP document [OAS12].

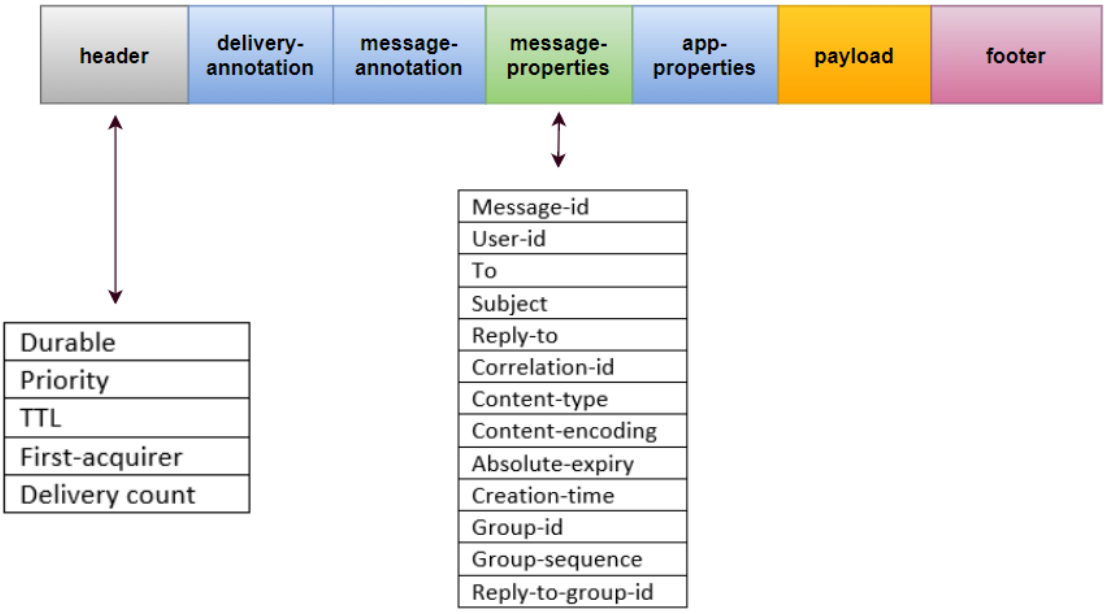

Figure 4.6: Standard AMQP message

In the figure 4.6, a standard AMQP message format is seen. In the header part, fields such as time to live, durability, and delivery attempts can be set. Delivery annotations

convey information from the sending peer to the receiving peer. Message annotations carry information about the message. The section for message annotations is used for properties of the message that are aimed at the infrastructure and should be propagated across every delivery step. In the properties section, there are several important fields such as 'Topic, 'MessageID,' 'Correlation,' 'Content MIME type,' et cetera. More information about all the fields within a frame can be found in the official document of the AMQP [OAS12].

The payload is in binary format, but it supports many MIME types. The message can be arbitrarily long, and if it is too long, it will be sent as smaller frames. In addition, AMQP provides built-in TLS for the security. It encrypts the TCP packets and also has SASL for the authentication.

The native protocol for RabbitMQ is the AMQP. As shown in table 4.1, RabbitMQ can be used in many different programming languages and platforms. To generate AMQP messages, RabbitMQ client libraries can be used. As the interoperability is the fundamental aim of AMQP, the language and the platform do not matter as much as the broker for the generation of message. For example, a .NET developer can use any client library of AMQP for .NET, e.g. RabbitMQ client library for C# to form a message. The developer can then send a message to any AMQP server, e.g. StormMQ, ActiveMQ or RabbitMQ. On the other side of the communication, a Java developer can fetch the message using an AMQP client library, such as libraries from RabbitMQ or ActiveMQ. The critical point is to have an AMQP broker and a message in AMQP format.

To explain the AMQP's primary goal in solving the interoperability problem between various platforms and brokers, some example situations can be imagined. The following example cases are taken from Richards [Ric11].

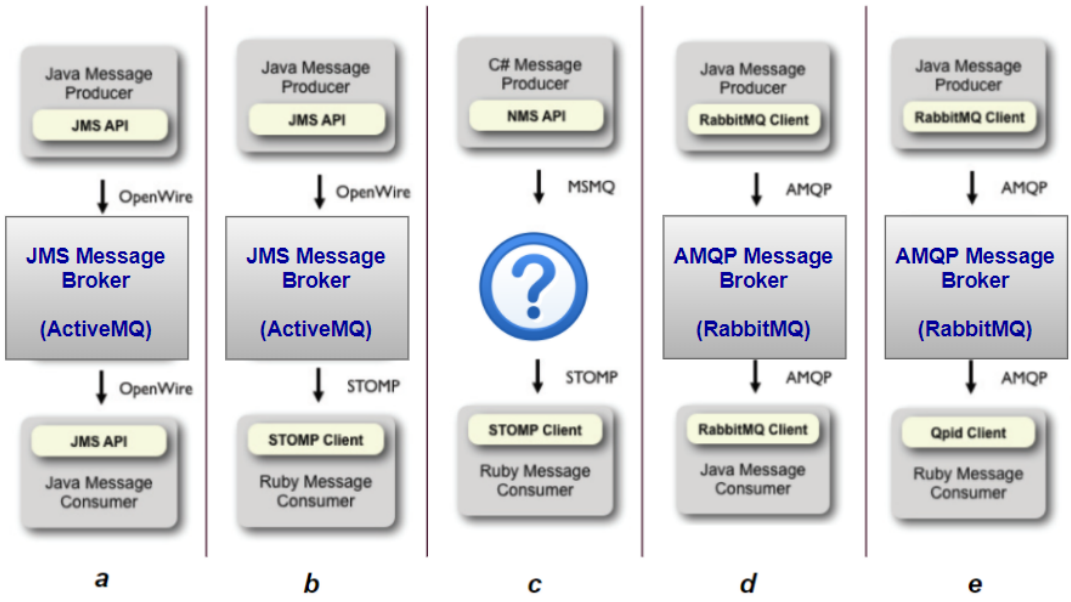

Figure 4.7: AMQP interoperability (adapted from [Ric11])

For example, JMS is designed to be interoperable within the JAVA platform. Messaging integration between Java to non-Java platforms can be difficult to implement [Ric11]. In a simple case from figure 4.7-a, two Java applications are using messaging systems. They use JMS API and ActiveMQ is employed as a broker. The communication is established via OpenWire, which is the native protocol for ActiveMQ. In this case, ActiveMQ can be easily altered with another broker that is JMS compliant, such as HornetQ. Then, instead of OpenWire, the HornetMQ protocol will be used. This change requires just a small code or configuration change. In figure 4.7-b, the integration of Java application and Ruby application can be seen. Since Ruby cannot simply use JMS, it instead utilises STOMP, which is the most common protocol for Ruby. In this case, a broker that can translate OpenWire to STOMP (protocol bridge) is needed. ActiveMQ has luckily built-in OpenWire to STOMP protocol bridge and it can be used in this scenario. The Java application forms the message using JMS and sends it via OpenWire. Next, in the middleware, this message will be transformed to STOMP, and the Ruby application can decode this STOMP message using the STOMP-client library. However, this solution is not satisfying since it is not easy to find a broker that will always support both protocols, and even if there is, then the solution would be restricted to one particular vendor such as ActiveMQ.

In figure 4.7-c, one side is the Microsoft application written in C# and sends messages using NMS API with MSMQ protocol. The other side is the Ruby application with the STOMP communication type. In this case, there should be a broker that has a built-in protocol bridge between MSMQ and STOMP. There could be many similar scenarios. This cross-platform interoperability between message-based systems is the reason why AMQP was created. When AMQP is the protocol, then any AMQP-compliant client library and any AMQP-compliant broker can be used. In figure 4.7-d, RabbitMQ is used as the broker, and the RabbitMQ client library is used to form messages in both Java applications. Java Message Service could also be used with the correct configuration that sets the application layer protocol as AMQP. Messages would be constructed using JMS and could be transported as in the form of AMQP. For integration between C# and Ruby applications, an AMQP-compliant library is needed. These applications cannot use JMS directly since they are not Java applications. In figure 4.7-e, the C# side uses RabbitMQ client libraries, the Ruby side uses Apache Qpid client libraries to form or decode messages, and they use the AMQP format or receive messages via the RabbitMQ broker. However, they could both use RabbitMQ client libraries or Apache Qpid client libraries; this would not change anything. In a similar manner, ActiveMQ could be used instead of RabbitMQ, because it also supports AMQP.

MQTT

Message Queuing Telemetry Transport is a publish-subscribe-based messaging protocol. It works on top of the TCP/IP protocol. Although it is meant to be sent over TCP, some brokers also support MQTT over Websocket. The protocol is lightweight, open, simple, and designed to be easy to implement. These properties make it fit for the use in many circumstances, including constrained environments such as for communication in M2M

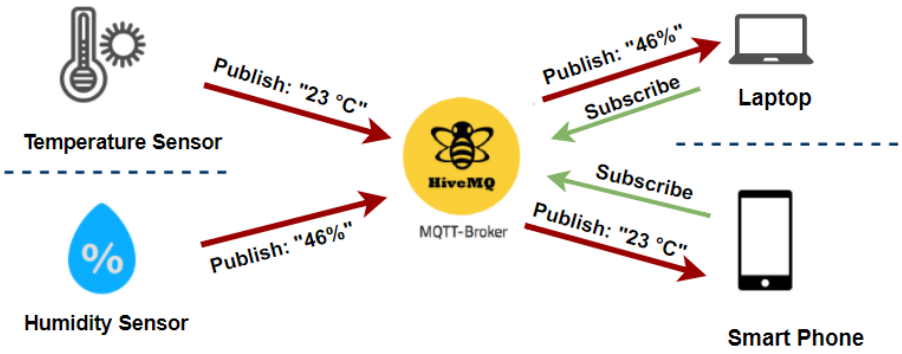

Figure 4.8: MQTT overview (adapted from from [Hiv16])

and IoT contexts where the network bandwidth is vital, and the power consumption is very limited [OAS15]. In the figure 4.8, an overview of MQTT can be seen (with HiveMQ).

The original aim of MQTT is to match device-to-gateway messaging specifications, and it does not satisfy most requirements of the gateway to a data centre, service-to-service connections, or even device-to-device communication [Sol17]. As its name states, MQTT's primary purpose is telemetry or remote monitoring. Its goal is to accumulate data from many devices and transport that data to the IT infrastructure. It targets large networks of small devices that need to be monitored or controlled from the cloud [Sch13a].

For example, MQTT has been used in devices that communicate to a broker via satellite link, in healthcare providers, and in a range of smart home and small device scenarios. It is also a convenient choice to use for mobile applications because of its frame size, low power usage, and its publish-subscribe characteristics, as it can forward data to one or many recipients efficiently.

Message Queuing Telemetry Transport is lightweight, and the header is minimal. In the figure 4.9, the MQTT packet format can be seen. In the frame, there are fixed fields and variable fields. The variable fields can be different for each message type. The header contains the following fields:

- ***Message Type:*** This can be 'Publish,' 'Puback,' 'Subscribe,' 'Unsubscribe,' 'Suback,' 'Connect,' 'Connack,' et cetera.
- ***DUP:*** Holds information about whether the packet is duplicate or not.
- ***QoS:*** For the three-level QoS, '0' is for exactly once delivery, '1' is for at-least-once delivery, and '2' is for at-most-once delivery.
- ***Retain:*** If this field is set as true, the message will be saved in the broker, and when there is a new subscriber, the new subscriber will get the message as soon as it subscribes.

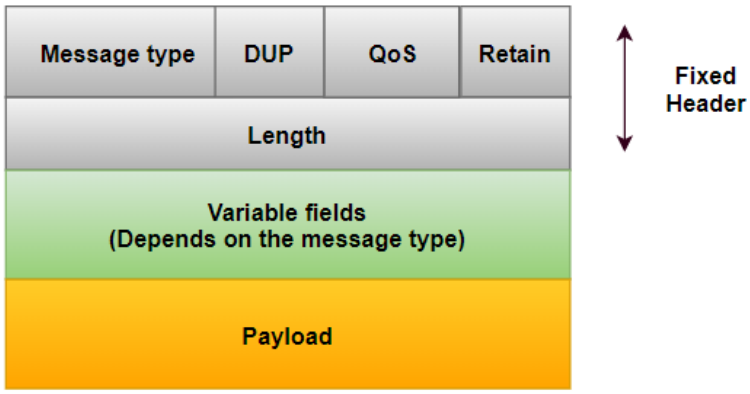

Figure 4.9: MQTT frame

In the variable header, there can be fields such as 'Username' and 'Password' for the 'Connect' type of message. For the 'Subscribe' type, there can be 'Topic Name' and 'Message ID.'

A topic is an UTF-8 string, which is managed by the broker to filter and route messages for each connected consumer. A topic can be made of one or more topic levels. Each topic level is separated by a forward slash (topic level separator) [Hil17]. For example a topic looks like: 'uni-stuttgart/iaas/room1/temperature.' The subscriber who subscribes to this topic will get the temperature data of room1 of IAAS department of University of Stuttgart. MQTT also has wildcard feature in the topics. '+' represents single level wildcard and '#' represents the multilevel wildcard. For instance, 'uni-stuttgart/iaas/+/temperature' gives the temperature information from every room in IAAS. However, uni-stuttgart/iaas/# returns all information from all rooms.

In payload, MQTT can support many different data types such as 'img,' 'text,' and 'zip.' Although MQTT is lightweight protocol, it provides sufficient reliability with three-level QoS [Hil17]. The protocol does not support any built-in security layer; however, most MQTT brokers support TLS. For the authentication, it uses basic 'login' and 'passcode' fields in the header of the 'Connect' message type.

STOMP

Simple (or Streaming) Text Oriented Message Protocol, is a simple text-based protocol, designed for working with MOM. It is intended to send over TCP initially; however, STOMP over WebSocket is more popular than STOMP over TCP. It provides an interoperable wire format that allows STOMP clients to communicate with any message broker supporting the protocol. It is therefore language agnostic, i.e. a broker developed for one programming language or platform can accept communication from any application written in any language or developed in any platform [STO12].

According to Richards [Ric11], STOMP targets for having simplicity, having interoperability, and being a lightweight protocol, yet it does not need to constrain itself as the embedded

device would require. He also stated that the architecture outlook is similar to HTTP (text-based), but it is not RESTful. There are a few commands: 'Connect,' 'Send, 'Subscribe,' 'Unsubscribe,' 'Begin,' 'Commit,' 'About,' 'Ack,' 'Nack,' and 'Disconnect.' For each command, there are just a few headers, such as 'ID,' 'to,' 'destination,' and 'version,' depending on the command. Beside command-specific headers, STOMP also has default headers, including 'content length,' 'content type,' and 'receipt' [STO12]. Like AMQP, STOMP supports many MIME types in addition to binary data. The main difference between AMQP and STOMP is that STOMP is text-based, and that makes it easy to develop with scripting languages such as Perl, Lua, Ruby, or Python. However, AMQP is in binary format and works well with languages such as Java, C#, and high-level other OOP programming languages [Ric11].

Simple (or Streaming) Text Oriented Message Protocol does not have built-in security or QoS policies. These depend on the broker and the transport method. The most used transport method by the brokers is WebSocket. When WebSocket is used, then the protocol can have TLS as security. However, if the broker does not use any security and transports the messages over a simple TCP, then there will be no security. In a similar manner, there is no three-level QoS in STOMP if the broker does not have it. If the broker supports guaranteed delivery or similar features, then STOMP can have these features.

There are, in total, 15 commands, such as 'Begin,' 'End,' 'Error,' 'Send,' 'Message,' and 'Subscribe.' Each of the commands can have up to four or five different headers. They mainly have 'ID' and 'destination' fields in common. The remaining one or two fields depend on the message command. In addition to command-specific types, there are standard header fields. These are 'content length,' 'content type,' and 'receipt.' As STOMP messages can support different MIME types in the body, if the 'content type' is not set, it will be considered as a binary array [STO12].

XMPP

Saint-Andre, Smith, and Troncon [SST09] book explains the XMPP as an open technology for real-time communication, using the XML as the base format for exchanging messages. It offers the ability to deliver messages (small pieces of XML) from one place to another in real time. The XMPP is, in principle, a messaging technology for sending fragments of XML. An XMPP connection is established and continued over TCP sockets that will be long lived, unlike HTTP. Next, an XML stream of bytes will be negotiated and sent to the XMPP servers.

The long-lived TCP connection will be entirely asynchronous. Although it is a real time and instant messaging (IM) application, it is developed to use asynchronous communication in cases where the XMPP servers cannot handle the messages immediately. In this way, clients will not be blocked, and the server will reply as soon as possible. The exchanging of an unlimited number of XML fragments can be possible in one single TCP as opposed to the traditional approach used in web and email technologies, in which a TCP connection is opened to handle only one complete transaction and is then closed again [SST09].

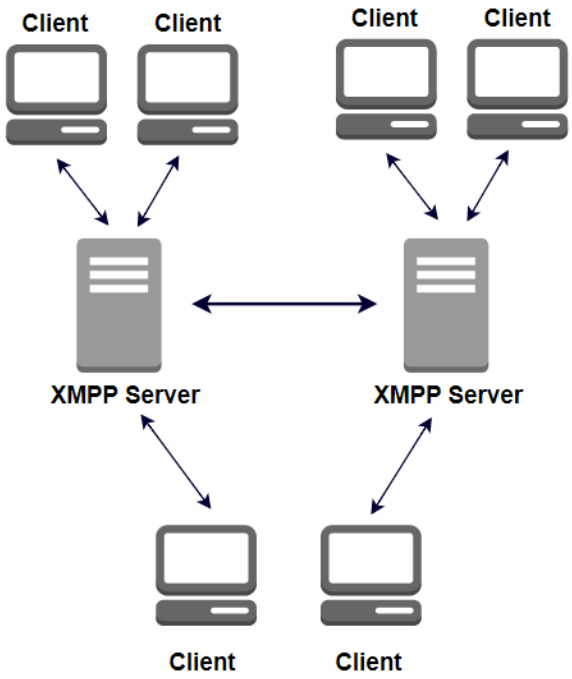

Figure 4.10: XMPP architecture (adapted from [SST09])

The original Jabber developers were focused on building an IM system. However, because of the extensible nature of XML, XMPP has become popular among the application developers who need a reliable, secure infrastructure for the fast exchanging of data, not just IM features. As a result, XMPP has been used to develop a wide range of applications, including alerts and notifications, lightweight middleware and web services, whiteboarding, multimedia session negotiation, intelligent workflows, geolocation, online gaming, social networking, and more [SST09]. XMPP is an extensible communication type, as is implied from the name. The extensions have been made over the years, and now it supports almost all kinds of communication characteristics. The protocol supports point-to-point (request-response and fire and forget), multicast, group messaging, and publish-subscribe.

In the figure 4.10, an XMPP architecture is seen. Although it does not have a central message broker, it has XMPP servers. These XMPP technologies use a decentralised client-server architecture similar to the architectures of the World Wide Web (WWW) and the email [SST09]. The most important feature of using a decentralised client-server architecture is the separation of the development of client and servers. User experience, good UI design, and usability are the focus for XMPP client developers, while server developers can focus on reliability and scalability.

Due to decentralised design of XMPP, it is more robust and reliable, as it will not have a single point of failure, unlike purely peer-to-peer technology, anyone can run his or her XMPP server and can easily attach it to the network. Furthermore, the servers can require important security policies such as user authentication, channel encryption, and prevention of address spoofing.

Like email, but unlike the Web, XMPP systems include plenty of inter-domain connections. However, an XMPP message is delivered to a recipient at a different domain, and the sender connects to its 'home' server, which then connects directly to the receiver's server without intermediate stops. This direct federation model has significant differences from the indirect federation model used in an email (in particular, it helps to prevent address spoofing and certain forms of spam) [SST09].

Some features of XMPP are from the book written by Saint-Andre, Smith, and Troncon [SST09]:

- It implements encryption of the connection between a client and a server or between two servers.
- The authentication service guarantees that parties are trying to communicate over the network are first authenticated by a server, which acts as a kind of guard for the network.
- It provides presence information. It gives the information about the availability of the entity such as offline, busy, or available. Presence information can also include more complicated information (such as whether a person is in a meeting).
- It also provides storing a contact list or roster on an XMPP server. The most frequent use for this service is an IM 'friend list,' but any entity that has an account on a server can use the service to maintain a list of known or trusted entities (e.g. it can be used by bots).
- It supports the classic use of one-to-one messaging, which is personal IM, but messages can be arbitrary XML, and any two devices on a network can communicate and exchange messages—they could be bots, servers, components, devices, XMPP-enabled web services, or any other XMPP entity.
- It is also able to generate a notification, and have it delivered to multiple subscribers using publish-subscribe.

XMPP can be a good choice when: [SST09]

- Need information about network availability, i.e. presence
- Need to deliver real-time and just-in-time alerts and notifications instead of continually polling for updates.
- Need encryption of the communication, strong authentication, and trusted identities.
- Need communication among a distributed network of entities or servers.

Since it is a native 'instance messaging' protocol, it can be useful when developing social networking applications. Moreover, for cloud computing and M2M communication, it makes sense to use real-time messaging to organise activities between a distributed network of entities, especially because presence information and service discovery can determine which entities are available in the system and capable of executing particular jobs [SST09].

```
<message from="you@yourdomain.tld/work"
         to="daughter@yourdomain.tld"
         type="chat">
  <body>Hi honey!</body>
  <active xmlns="http://jabber.org/protocol/chatstates"/>
</message>
```

Figure 4.11: XMPP simple message [SST09]

Since it is a more human-readable protocol and mainly used by humans, in the IoT context, it may be better to use the device-to-server type communication since people are connected to the servers. However, when the efficiency and power is a concern, then it is better to use MQTT for device to server or gateway since parsing XML requires a computational power, and MQTT is a lot more lightweight than XMPP [Ear16].

Extensible Messaging and Presence Protocol does not have many header fields, but it has different types of messages. For example, XMPP has three kinds of messages, and each of these message types has an additional three to five different types. Main three types are 'Message,' 'Info Query (IQ),' and 'Presence' [SST09]. For example, 'Message' type has five different types that include 'Normal,' 'Chat,' 'Group,' 'Error,' and 'Headline.' 'IQ' has four different types, which are 'Get,' 'Set,' 'Result,' and 'Error.'

Furthermore, 'Chat' type (which is sub-type of 'Message') can have header fields such as 'Active,' 'Composing,' 'Inactive,' and 'Gone.' An example of an XMPP message can be seen in figure 4.11. The number of header fields in one message type are normally fewer than AMQP; however, for each message type, there are different header fields. Because of extensible nature of XMPP, header fields have been extended over the years and the XMPP can have many headers fields that are almost as complex as AMQP has.

Extensible Messaging and Presence Protocol uses messages in XML format, and that is why sending binary files requires a different extension. Small binaries can be hashed and added to the XML elements, but this is not the case for transmitting large data. There are two ways to send large files. The sender sends a message that says it needs to send a file, and the receiver and the sender agree on an IP and port pair. The sender then starts sending the binaries directly to the (IP, port) pair through Sockets as ByteSteams. The sender can send the file to a proxy server if the client is not reachable at a specific address (because of firewalls or because the receiver is offline). The sender then activates the file in the proxy server so that the client can claim the data when he or she needs to get it [SST09].

4.2.2 Communication Protocols Without Middleware

The communication protocols in this chapter have similar characteristics as the protocols in section 4.1. However, the main difference is that these types are asynchronous by default and the connection is bidirectional. In this way, they are more like messaging models, but they do not require the broker to work.

DDS

The Data Distribution Service is a data communications standard managed by the OMG[18] targeting low-latency data transferring for distributed programs. The DDS standard defines support for a type-safe application with types (a dynamic discovery of data writers [publishers], data readers [subscribers]) and topics (a rich set of QoS policies) and interoperability between different DDS vendors. The DDS offers high-performance data communications, convenient for real-time and near real-time systems. Some open source DDS implementations in addition to commercial implementations are available and used. They are also used in products built for and targeting embedded communications [Obj13].

According to [Twi17], DDS standard provides developers with a well-defined API that aims for ease of use. This API enables the developer to write portable code, which can also work with different implementations of DDS from different vendors. In the same document, it is also stated that the DDS standard uses the Real-Time Publish-Subscribe (RTPS) standard as a wire level protocol. The primary aim of this protocol is to allow interoperability. By using RTPS, applications that are built with different DDS implementations will be interoperable, i.e. they can communicate and send data to each other without any problem. Data-Distribution Service users will not be locked into specific vendors. They will instead use the same standard and can change the DDS implementation anytime if they need to change in the development cycle. In general, DDS is a peer-to-peer communication model that requires any other device, such as gateways, servers, or daemons that must be run or configured [Twi17].

The Data Distribution Service also offers a flexible publish-subscribe architecture, and these publishers and subscribers can be dynamically discovered. Dynamic discovery enables developers to write extensible applications. Because of dynamic discovery, the developers do not have to configure or know the endpoints, as they will be automatically discovered by DDS [Twi17]. This discovery is not only for determining endpoints. The Data Distribution Service can understand if the endpoint is writing data, just reading data, or both. It will discover the type of data and information being published or subscribed to. It will also see the publisher's offered communication characteristics and the subscriber's requested communication characteristics [Twi17].

In figure 4.12, an overview of DDS architecture is demonstrated. It has an imaginary broker that is called 'Global Data Space.' It is called imaginary since it does not exist as a real broker. 'Global Data Space' is kept updated in clients' memory so that clients will recognise this as a central broker. When there is a new node, topic, or data, the clients will be notified, and 'Global Data Space' will be updated automatically. The 'Global Data Space' also has advanced capabilities. Unlike standard message brokers, it not only manages queues and carries messages; it also connects the data writers and the data readers based on the type of data in which they are interested and, according to the QoS, that they offer or expect. The main thing that is transferred is not the message but the data.

[18] Object Management Group (OMG) - http://www.omg.org

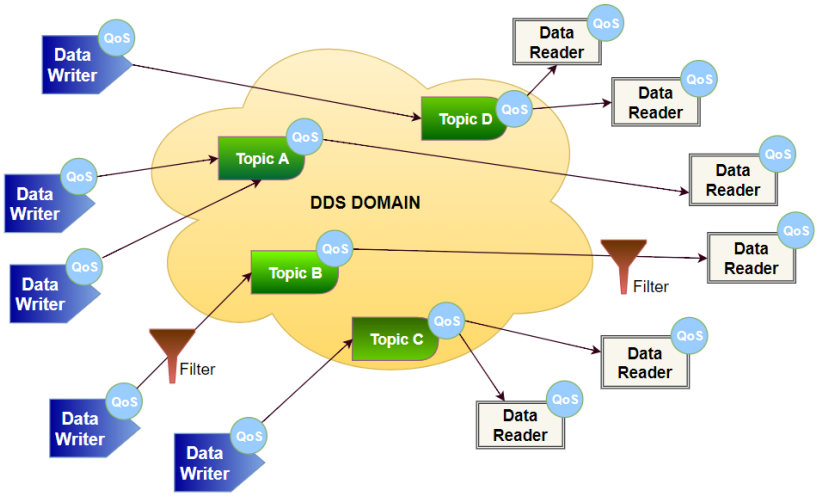

Figure 4.12: (DSS architecture figure is adapted from OMG website[a])

[a]Figure is adapted from - http://portals.omg.org/dds/what-is-dds-3/

Since DDS has a dynamic discovery, multicast and publish-subscribe, it is beneficial for device-to-device, device-to-gateway, and gateway-to-server communication and can even be used between servers. Due to its 'Global Data Space' and highly extensible features, it is a perfect solution for a complete IoT containing sensors, gateways, and servers.

If the IoT or cloud application runs under the same LAN, then DDS works perfectly for whole Cloud or IoT solutions. However, if all the components are not on the same LAN, then the extended version of DDS should be used. There needs to be an additional router service between the LANs over WAN to connect the different parts. In figure 4.13, an example of a DDS router can be seen.

Since DDS is a complete service, it also has a significant and complicated security layer. In this segment, all the negotiations regarding TLS and authentication are performed.

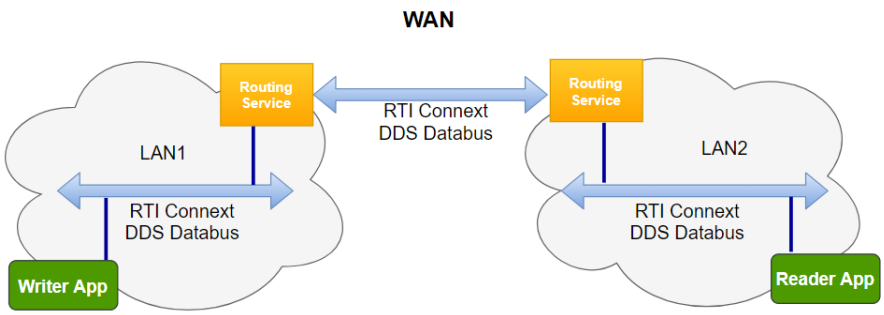

Figure 4.13: DSS over WAN (figure is adapted from RTI website[a])

[a]Figure is adapted from - https://www.rti.com/resources/usecases/real-time-lan-over-wan

CoAP

According to Shelby, Hartke, and Bormann [SHB14], the CoAP is a specialised web transfer protocol developed for the use of constrained devices and constrained (e.g. low-power, lossy) networks. These devices have 8-bit microcontrollers with less ROM and RAM. Constrained networks, such as IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs), have significant error rates and much less throughput. The authors also stated that the protocol is intended for M2M applications such as smart energy and building automation [SHB14].

The Constrained Application Protocol provides a request-response interaction design between application endpoints, supports the built-in discovery of services and resources, and includes fundamental concepts of the Web, such as URIs and Internet media types. The CoAP is designed to quickly interface with HTTP for integration with the Web while also offering specialised demands such as multicast support, low overhead, and simplicity for constrained environments [CLR17].

The Constrained Application Protocol is the HTTP for the low-power or constrained devices. Both HTTP and CoAP work similarly. Similar to HTTP, CoAP uses HTTP verbs ('GET,' 'POST,' 'PUT,' and 'DELETE') and URIs. That is why CoAP can also be called as RESTful service. However, URIs of CoAP is similar to topic names of MQTT rather than a Web URL. For instance, a valid CoAP request is 'GET device1/values/temperature.'

The main difference between CoAP and HTTP, header fields of CoAP are much lighter, and CoAP uses UDP. Since it uses UDP, it allows multicast, and this enables the device to send packets to other devices instead of sending only to the server. Another difference is that HTTP is synchronous, but CoAP does not necessarily need to be synchronous. This is very useful for device-to-device communication because of the multicast feature and can be used in device-to-gateway communication. However, this will not be preferred when the device has enough computational power, as the features of CoAP are limited.

Since CoAP uses UDP as the transport protocol, the security will be DTLS instead of TLS, which is the security layer for TCP. There will be no security loss, though, since the only difference between TLS and DLTS is that one can encrypt TCP packets and the other can encrypt UDP. There is no authentication mechanism in CoAP, but it can be done using the security layer, which is DTLS. The identity of the communicating parties can be authenticated using public-key cryptography [SHB14].

Although HTTP and CoAP are similar, CoAP is included in this category since it has Dynamic discovery (using UDP multicast), and it has server push functionality using the 'Observe' feature. Using the 'Observe' feature, the server will inform the client automatically when there is new information. 'Observe' pattern avoids requesting or polling for the resource continuously. 'Observe' feature can be thought of as publish-subscribe. The CoAP client can simply add an 'Observe' entry in the request header to use the feature. A period (in seconds) can be given with 'Observe' entry. When the period is over, the server will deregister the client and will stop sending automatic updates.

WebSocket

Wang, Salim, and Moskovits [WSM13] explained that WebSocket is a new protocol that enables the server to send information to the browser without the browser having to demand it. The browser can also send data to the server. This is extremely useful for sending notifications and updates because the server can send information as soon as it has (server push) instead of waiting for requests from Web browsers. The authors also defined WebSocket as a computer communication protocol, providing full-duplex communication channels over a single TCP connection. WebSocket is a naturally full-duplex, bidirectional, and single-socket connection.

WebSocket is analogous to socket programming in C# or Java but with the difference that it is for Web [WSM13]. Websocket does not add anything to the TCP layer except for some small features for browsers such as extra security and allowing multiple services to one port [FM11]. Unlike HTTP in which every request is initiated with a new connection, the socket that is connected to the server stays 'open' for communication. That means information can be pushed to the browser in real time. Since the client does not perform polling to obtain the update, Websocket can save bandwidth. However, caching is not enabled in WebSocket; HTTP1.X could consequently be a bit faster in some cases.

Every WebSocket connection begins with an HTTP request. This request is much like any other, except that it includes a special header: Upgrade. The Upgrade header indicates that the client would like to advance the connection to another upper-level protocol. In this case, the protocol is WebSocket. If the server supports WebSocket, the communication is automatically upgraded to WebSocket.

The WebSocket header is lightweight compared to HTTP headers. Contents of each WebSocket packets are generally HTTP1.X requests and responses which are just encoded and wrapped in binary format. That is why, WebSocket does not have many header fields. In figure 4.14, the structure of WebSocket frame (or message) is seen. The message type can be either text or binary, which can be set via 'Op Code' field. 'Length' is for the size of payload. If the length is more than 1 byte, then 'Extended-length' is also used. 'Masking' is used for extra security to avoid attacks to the servers. Like HTPP, big data can be sent via chunks.

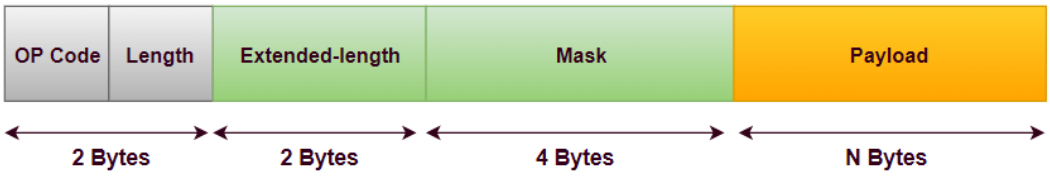

Figure 4.14: WebSocket frame

HTTP2

Hypertext Transfer Protocol/2 is similar to WebSocket, and it was developed to fix several problems of HTTP [BP15]. Both HTTP2 and WebSocket are binary transport protocols that multiplex numerous streams going over a single (optionally TLS-encrypted) TCP connection. The HTTP1.X packets or frames are converted to binary and send over only one TCP instead of sending each request-reply pairs in different TCP. For this reason, HTTP2 is faster and more efficient than HTTP1.X. The other problem with HTTP, the polling problem, was also fixed in HTTP2. Now the connection is open all the time (until one party closes explicitly), and the server can then push if it has new information. In other words, there is no need to poll the server for new information [BP15].

Figure 4.15 shows the comparison between HTTP1.X and HTTP2.

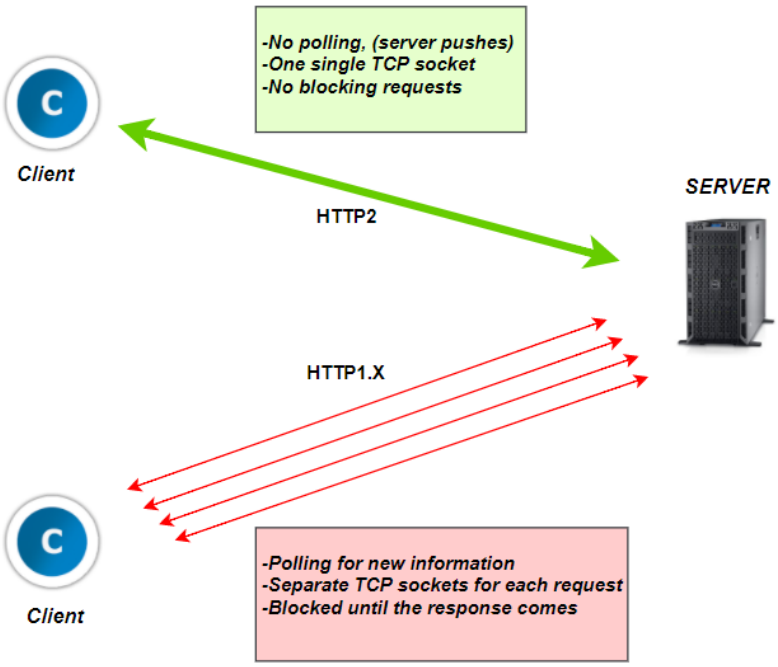

Figure 4.15: HTTP vs HTTP2

HTTP2 and WebSocket are similar except for HTTP2 allowing header compression to reduce the frame length. In addition, HTTP2 can carry only HTTP frames, while WebSocket can carry different message formats such as STOMP, MQTT, HTTP, or a plain TCP packet. The rest is the same for both protocols. As did WebSocket, HTTP2 started with an HTTP1.X frame. The first HTPP1.X frame has 'Upgrade: HTTP2' entry in the headers, and if the server supports HTTP2, then the communication will be continued over HTTP2 [BP15].

5 Transformation of Communication Types

In this chapter, transformation capabilities between communication types will be explained. At the beginning of each section, there will be tables which summarize the characteristics of communication types and transformation capabilities. Only the critical transformations will be detailed with tables and figures. The rest of the transformations will be shortly covered in section 5.7. In each section, there will be explanations regarding the features and the characteristics that will no longer be available after the transformation. What header fields and information can be lost or preserved will also be given. After that, the challenges and the difficulties when the developers want to implement the features of the old communication type using the features of the new communication type. All the challenges and difficulties will be explained; however, one can discard some challenges if the new communication type does not need or does not use such functionalities. For example, a component uses AMQP and it is required to change to HTTP. In such scenario, implementing publish-subscribe feature of AMQP using HTTP is a big challenge. However, the component may use AMQP for just sending information to another component. In this case, the challenge of implementing publish-subscribe using HTPP is not essential.

Before reading transformation capabilities, please read the related chapter in 4. The features of communication protocols are explained, thus no further explanations will be given. For example, in table 5.1 and table 5.2, the entries are only for summary purposes and what they mean can be found in the related chapters above. Please also note that this is not a complete guide of transforming one communication type to another type. This master thesis is an analysis of transformation capabilities and meant to give an idea regarding transformations. That is why there will not be step by step instructions to convert from one protocol to another. This would be beyond the scope of this thesis.

5.1 Transformation Between AMQP and STOMP

In table 5.1, the features of the two types are given. In table 5.2, the transformation capabilities are summarized.

The Advanced Message Queuing Protocol and STOMP are both messaging solutions, and they require supported brokers to work correctly. Although interoperability is the main aim for both, there are differences between AMQP and STOMP. While, STOMP is text-based, and it is famous for scripting languages such as Ruby or Python, AMQP is more popular among non-scripting based languages such as C# or Java since it is a binary based protocol. As they both support various types of interactions such as publish-subscribe,

Features	AMQP	STOMP
Type	Broker based messaging	Broker based messaging
Focus	Interoperability	Interoperability
Protocol type	Binary based	Text based
Sent over	TCP	TCP, WebSocket
Popular in	C++, C#, Java, Python	Ruby, Python, PHP
Interaction Patterns	All	Publish-Subscribe, Fire-and-Forget
QoS Policies	Sophisticated	Depends on the broker
Security	SASL, TLS (SSL)	Depends on the broker
Header Size	Very Large	Small
Supported Payload	Various MIME Types	Various MIME Types

Table 5.1: AMQP-STOMP summary

Transformation	Transformation to AMQP	Transformation to STOMP
Is it possible?	Yes	Yes, with some limitations
Difficulty	Medium	Hard
Limitations	None	-Loss of many header fields -Possible loss of interaction patterns -Possible loss of security

Table 5.2: AMQP-STOMP transformation capabilities

the transformation will not be arduous. STOMP to AMQP transformation could be done without any loss as AMQP is a lot more complicated than STOMP is. In addition, STOMP has no built-in security and QoS, but the broker may provide such features. AMQP has QoS policies due to its complex transport layer and its rich header frame of the message. Although STOMP does not have many header fields, it has different commands, and each command makes a different type of message. For each command in STOMP, there are usually two or three header fields such as 'id' and 'destination.' All these message types can be reproduced by AMQP as it is a powerful protocol. STOMP has 'ACK,' 'NACK,' or 'ERROR' kind of messages and equivalently AMQP has 'accepted,' 'rejected,' or 'error' headers.

Even though there is no information loss, there are challenges. One challenge could be the difference between the implementation of interaction patterns. Although they both support publish-subscribe, the implementations are quite different. While STOMP has 'SUBSCRIBE' or 'UNSUBSCRIBE' type messages, AMQP does not have any header fields for that as it supports publish-subscribe natively. In AMQP, publishers create queues, and subscribers connect to those queues and both parties are unaware of each other. Luckily, this complex queue management is the task of the broker. Every-time a 'SUBSCRIBE' (or 'PUBLISH') type of STOMP messages come, a protocol bridge should generate AMQP requests for creating or connecting to queues, and sends these requests to the broker. Another way could be that

protocol bridge will just map the messages, and a different computing device will manage requests such as 'connect' or 'create.' By this way, the complexity of protocol bridge may be reduced. Bridge then could assign the topic name to 'subject' field of AMQP message and forward the message to another application which handles the AMQP requests. Then the application requests the queue creations or connections as it will know the queue name from the 'subject' field.

However, there will be limitations if the message is transformed from AMQP to STOMP. Since AMQP has very complex header fields, it offers many features that STOMP can't provide such as 'durability,' 'priority,' 'ttl,' 'modified,' or 'creation time.' Another example could be that AMQP has 'rejected' message state or various error types such as 'server not found,' 'unauthorized,' or 'operation failed.' However, STOMP has one message type as 'ERROR.' Thus, when there is a rejection of a message or a server error, in STOMP, it would be returned just as an error and one should check the payload to understand the main cause of the error. On the contrary, in AMQP, one can understand that the actual cause of the error (such as 'server not found' or 'message is rejected') by just checking the headers and payload can be observed if further information about the error is needed.

Another significant limitation is that the STOMP broker may not provide a few interaction patterns such as request-response and multicasting. The Advanced Message Queuing Protocol is nothing but queues, and it offers an excellent interface to its brokers. The brokers, therefore, implement all kind of interaction patterns, as they do not need to implement further extensions to the protocol. However, STOMP does not offer such interfaces and brokers should develop an extension for STOMP to support other interaction patterns. For instance, ActiveMQ has a request-response extension of STOMP; however, most of the other STOMP brokers do not have such functionality. Another limitation is that STOMP can have security if the carrier has. For example, it can be transported over WebSocket in which the packets can be encrypted using SSL. The STOMP broker can configure those security settings. The other limitation of the security is that AMQP has an advanced layer (SASL) for authentication, whereas STOMP has only simple 'Login' and 'Password' fields in the message header. Finally, the payload will not make problems as they both support various data formats such as 'zip,' 'video,' or 'img' in their payload. Thus, payloads can be directly copied, and 'content-type' field can be set in both protocols.

5.2 Transformation Between AMQP and MQTT

Table 5.3 summarizes the features of the two types are given, and table 5.4 gives the summary of the transformation capabilities between these two types.

Message Queuing Telemetry Transport is designed for devices which have limited power consumption, consequently, the transformation can be done only at gateway level or server level. Thus, end device will send MQTT message and the transformation will be done at the gateway or after this level. It also makes sense to use more advanced protocols instead

Features	AMQP	MQTT
Type	Broker based messaging	Broker based messaging
Focus	Interoperability	Limited devices
Protocol type	Binary based	Binary based
Sent over	TCP	TCP, WebSocket
Popular in	C++, C#, Java, Python	C, Python
Interaction Patterns	All	Publish-Subscribe
QoS Policies	Sophisticated	3-level QoS
Security	SASL, TLS (SSL)	Depends on the broker
Header Size	Very large	Very Small
Supported Payload	Various MIME Types	Various MIME Types

Table 5.3: AMQP-MQTT summary

Transformation	Transformation to AMQP	Transformation to MQTT
Is it possible?	Yes	Yes, but very limited
Difficulty	Easy	Very hard
Limitations	None	-Loss of many header fields -Loss of interaction patterns -Possible loss of security

Table 5.4: AMQP-MQTT transformation capabilities

of MQTT after gateway level, since from this level devices are much more capable, and they do not need to use such limited MQTT protocol.

In figure 5.1, a simple scenario can be seen. Devices are sending MQTT to a device with a protocol bridge which does the conversion. Then backend gateway device sends the newly generated AMQP messages to the broker.

Since AMQP is more complicated than MQTT, no data loss will occur when MQTT is transformed to AMQP. Most of the header fields, which MQTT has, are also in AMQP. The payload can be copied 'as it is,' as a binary array. Then content type field of AMQP can be set in the translator if it is known through subscriber configuration [Vas17]. MQTT has a header field to set delivery type (three-level QoS). AMQP does not have such header field in the message, but via broker the necessary configurations can be set and the transport layer of AMQP can handle the rest. The other header field which does not exist in AMQP is 'Retain' field. However, AMQP brokers have this feature as well. The topic of the MQTT message can be written to the 'subject' of the AMQP by the protocol bridge, and it can forward to the gateway device. Then the gateway device gets the topic name from 'subject' field and makes necessary queue creation or subscription requests to the broker.

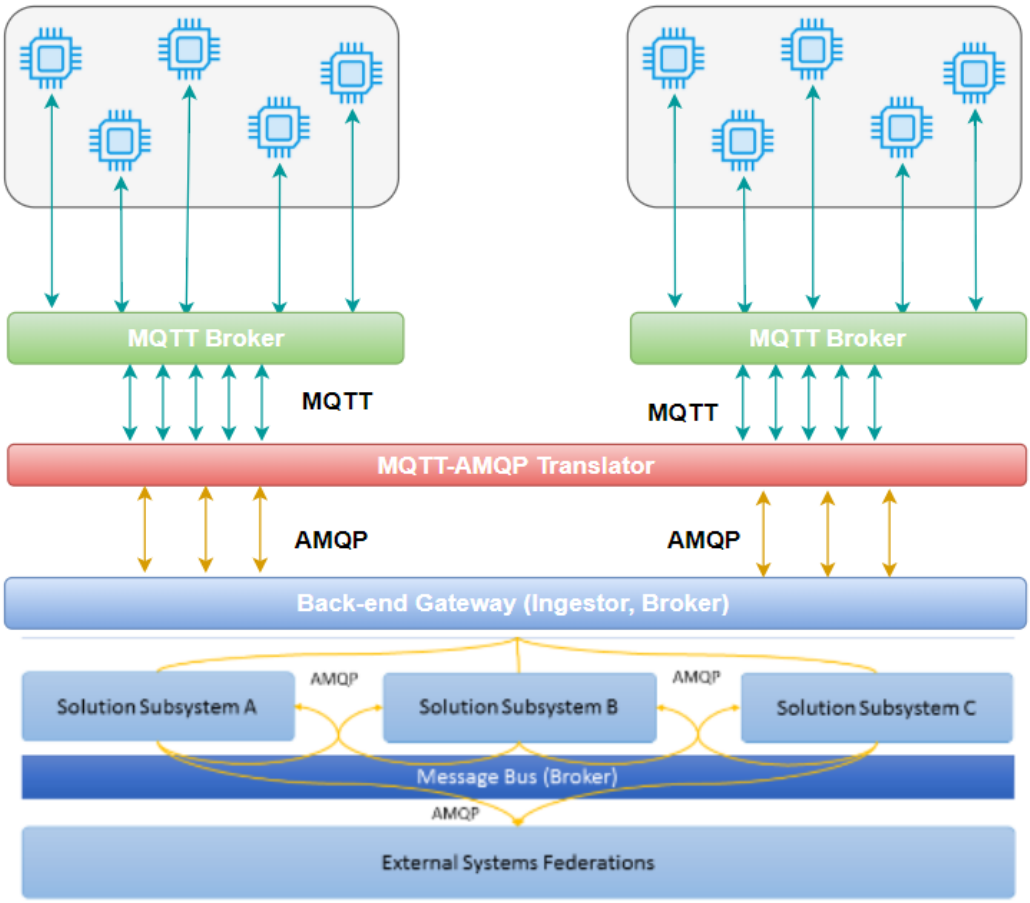

Figure 5.1: AMQP-MQTT scenario (adapted from [Vas17])

In figure 5.2, a small part of AMQP message header and a standard MQTT message header can be seen. 'variable-header' part of the MQTT can have optional 'Retain' and 'QoS' fields.

The transformation from MQTT to AMQP is smooth and without loss, on the contrary, transforming from AMQP to MQTT has limitations and challenges. The main weakness is that MQTT does not have point-to-point messaging capabilities. This limitation has a solution; however, it might not be feasible if there are too many devices in the system. On top of that, the implementation of this solution requires much work. For example, to implement a request-response pattern, a new topic is created for each request, and both requester and replier devices will subscribe to this topic. Requester publishes the message to this topic, and the replies will be read from this topic. Then the topics should be deleted so that there are not many topics at a time as the devices with limited capabilities cannot handle many topics at the same time. However, if there are too many devices communicating with each other, there is no way to decrease the number of topics. In AMQP, this complex task of handling requests and replies with queues is done by the broker. The second significant limitation is that a considerable amount of information will be lost due

to the substantial difference in headers. For example, 'durability' (of queues or messages), 'ttl,' or 'priority' headers fields as well as time, state and error information of the messages will be lost. Among all the features that AMQP offers, only three-level QoS can be achieved. Beside QoS level field, only 'message-id,' 'topic name,' and 'retain' message can be saved.

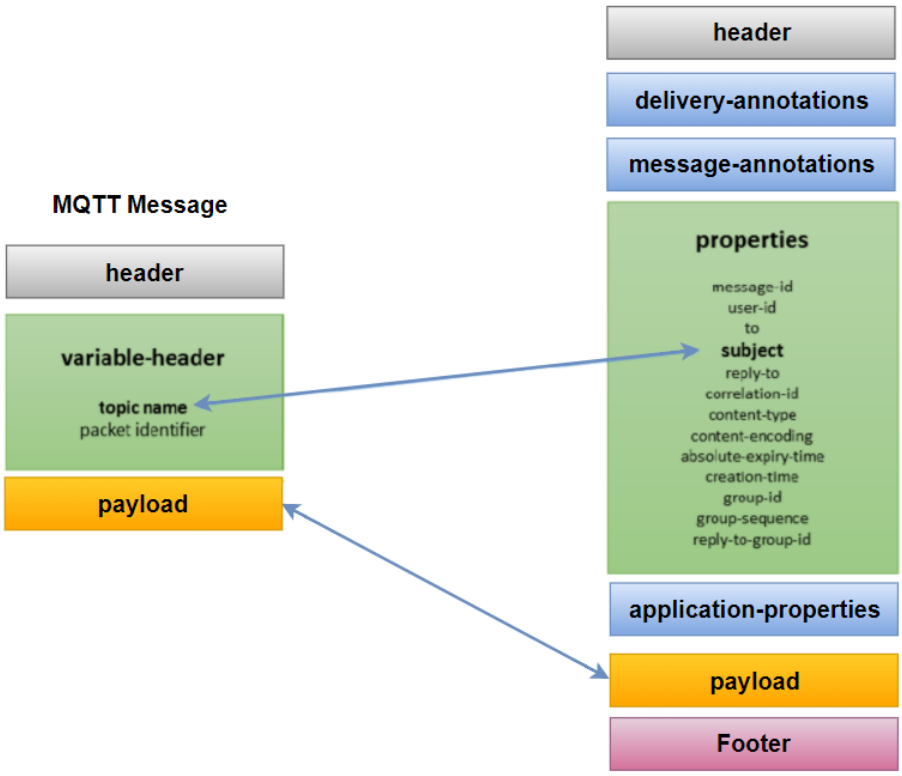

Figure 5.2: AMQP-MQTT headers comparison (adapted from [Vas17])

The security of MQTT is very similar to STOMP. Mostly the brokers use TCP with TLS, (rarely plain TCP), and MQTT has simple 'Login' and 'Password' fields in the connect message for authentication. In contrast, AMQP has built-in TLS and SASL for security and authentication.

5.3 Transformation Between AMQP and XMMP

Table 5.5 gives an outline for the two types, and table 5.6 presents the summary of the transformation capabilities between these two types.

The concepts, implementations, and aims of XMPP and AMQP are different, and that makes XMPP-AMQP transformation hard. The Extensible Messaging and Presence Protocol is mainly used among people; however, recently it is getting popular in the IoT as XMPP offers useful features such as 'Presence.' Although payload can be taken easily since AMQP support many types as well as XML payloads, some header fields in the transformation from XMPP to AMQP will be lost and cannot be preserved. Normally, XMPP does not have

Features	AMQP	XMPP
Type	Broker based messaging	Server based messaging
Focus	Interoperability	Instant Messages and Chats
Protocol type	Binary based	XML based
Sent over	TCP	TCP, WebSocket, HTTP
Popular in	C++, C#, Java, Python	Python, C#, Java,
Interaction Pattern	All	All (with extensions)
QoS Policies	Sophisticated	Sophisticated with extensions
Security	SASL, TLS (SSL)	SASL, TLS (SSL)
Header Size	Very large	Small
Supported Payload	Various MIME Types	Text and Binary

Table 5.5: AMQP-XMPP summary

Transformation	Transformation to AMQP	Transformation to XMPP
Is it possible?	Yes, with some limitations	Yes, but limited
Difficulty	Very hard	Very hard
Limitations	-Loss of few header fields	-Loss of features -Loss of many header fields

Table 5.6: AMQP-XMPP transformation capabilities

many header fields in a message, but it has different header fields for each message types, and mostly these headers do not exist in AMQP. By using extensions of XMPP, it can have too many headers fields can be as complex as AMQP. For example, a 'chat' type of message has 'active' (states that the other side of communication is online and actively listening to this communication), 'composing' (states that the other side is composing messages), or 'gone' header fields. These kinds of headers and features do not exist in AMQP. Another example is that 'Presence' type of message has 'Status' header field which gives information regarding the status of the device or person. Although AMQP has many header fields regarding the status of the message, it does not have header fields for the state of the device. Even though individual header fields for different message types will be lost, the header fields that come with extensions are mainly exist in AMQP. Thus, there will not be a loss of header fields which come with the extensions. The most significant challenge is to implement all these message types of XMPP in AMQP.

Other than the header fields, there are a few XMPP features that AMQP does not have. Contact addition and deletion is an important feature of XMPP which AMQP does not have. To achieve this, the protocol bridge should use publish-subscribe and should create a separate queue between the adder and added device. By this way, they can communicate over a specific queue. When one contact, delete the other one, then the common queue can be deleted. Another limitation is that group messaging and group chat. This can be

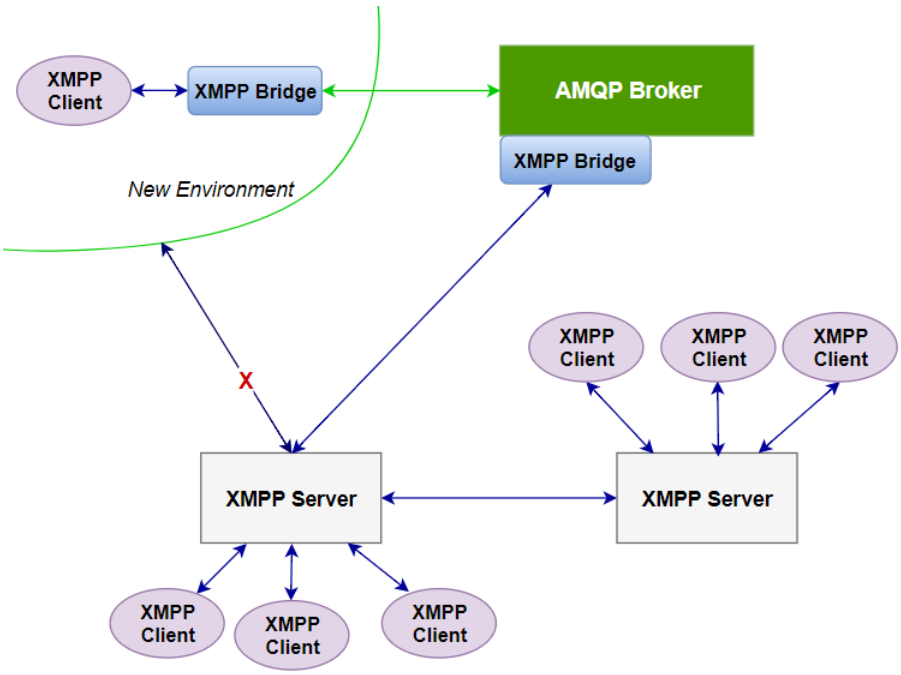

Figure 5.3: AMQP-XMPP scenario

achieved by AMQP using publish-subscribe with complex queue management; however, this also needs much work. The other challenge is publish-subscribe mechanism itself. As explained in section 5.1, AMQP supports publish-subscribe natively. Transportation layer of AMQP provides a complex interface and broker handles all the queue management. However, in XMPP, this is done by extension, and it behaves like another message type of XMPP which is 'PubSub' type. Like the other message types, it also has special header fields such as 'subscribe,' 'subscription,' and 'publish.' This difference leaves a complex task for the protocol bridge to understand the message type and to make necessary requests for queue management. In figure 5.3, an example situation can be seen. An XMPP client may be moved to an environment where XMPP is not supported.

The transformation of the other way, AMQP to XMPP, also has limitations and challenges. The first problem is the transforming publish-subscribe pattern. As already mentioned, AMQP and XMPP handle this model quite differently. Another challenge is that in the background, XMPP client sends a lot of information regarding the state of the messages and the clients. In AMQP, transport layer does all these complex tasks and sets the necessary header fields with the help of the broker. Now the protocol bridge should collect information from clients and the broker. Then it should continuously send notifications to XMPP server about what's happening in the transport layer. Although it is hard to do the conversion, luckily most of the header fields can be saved, because of the extensions of XMPP. It can have many headers such as 'ttl,' 'durability,' 'priority,' or 'creation time' that it usually does not have by extension. Thus, most of the header fields can be transferred from AMQP. However, there is a challenge and a limitation when the payload is not text. The

Advanced Message Queuing Protocol handles all the payload types easily, but XMPP needs an extra negotiation step for sending binary payloads. There still might be limitations. One way of transmitting big binary data is through sockets. The firewall may block socket communication. Another way is to send via a commonly agreed proxy server. In this case, protocol bridge can get the data and forward to AMQP client. However, when protocol bridge is dealing with more than one AMQP client, then sending and receiving big data could be so slow. In addition, AMQP can carry different types of binary payloads (img, jar, rar) files. In XMPP, all these types will be sent as a binary array, and then the receiver should be notified about the type unless the bridge can deduct the type. From the security point of view, there will be no loss as they both support TLS and SASL.

5.4 Transformation Between AMQP and HTTP

Table 5.7 shows the summary of features of both types, and table 5.8 gives an overview of the transformation capabilities between these two types.

Features	**AMQP**	**HTTP**
Type	Broker based messaging	Sync. Client-Server
Focus	Interoperability	Web server and browsers
Protocol type	Binary based	Text based
Sent over	TCP	TCP
Popular in	C++, C#, Java, Python	Python, Java, PHP
Interaction Pattern	All	Request-Response
QoS Policies	Sophisticated	None
Security	SASL, TLS (SSL)	TLS (SSL)
Header Size	Very large	Large
Supported Payload	Various MIME Types	Various MIME Types

Table 5.7: AMQP-HTPP summary

Transformation	**Transformation to AMQP**	**Transformation to HTTP**
Is it possible?	Yes, with some limitations	Yes, but very limited
Difficulty	Hard	Very hard
Limitations	-Loss of some header fields -Loss of features	-Loss of features -Loss of some header fields -Loss of interaction patterns

Table 5.8: AMQP-HTTP transformation capabilities

The main purpose of HTTP is to carry requests and responses between web applications and web servers. However, AMQP is the messaging system in which publishers and subscribers

are totally unaware of each other. They just publish messages or receive messages from queues. That is why HTTP has features that AMQP does not have and, AMQP has many different characteristics. The most important difference is that HTTP is synchronous, and AMQP is asynchronous.

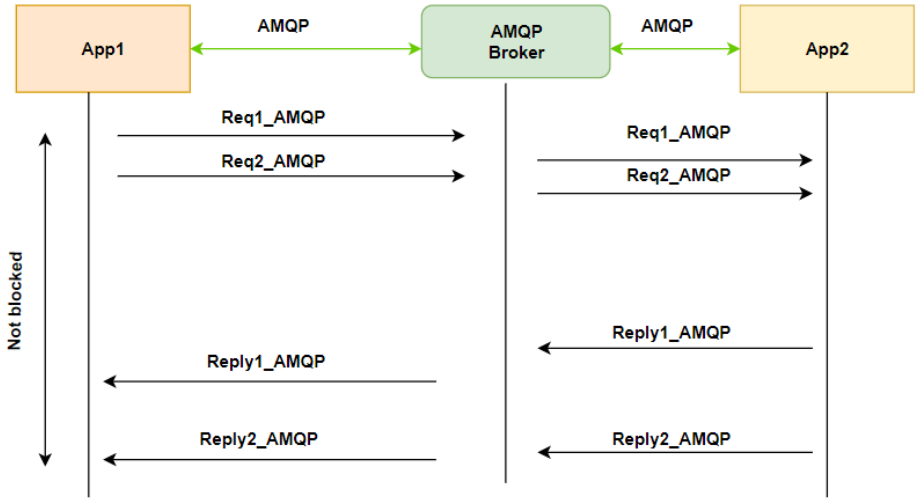

Figure 5.4: Asyncrounous client

The transformation from AMQP to HTTP have too many limitations and challenges. The most important limitation is one side of the communication will no longer be asynchronous. In figure 5.4, 'App1' and 'App2' communicates over AMQP. Since both are asynchronous, they do not block each other, and they respond to each other whenever they want. When the App1 is changed from AMQP to HTTP, it will lose it is asynchronous nature and will be blocked until the response comes. This is shown figure 5.5.

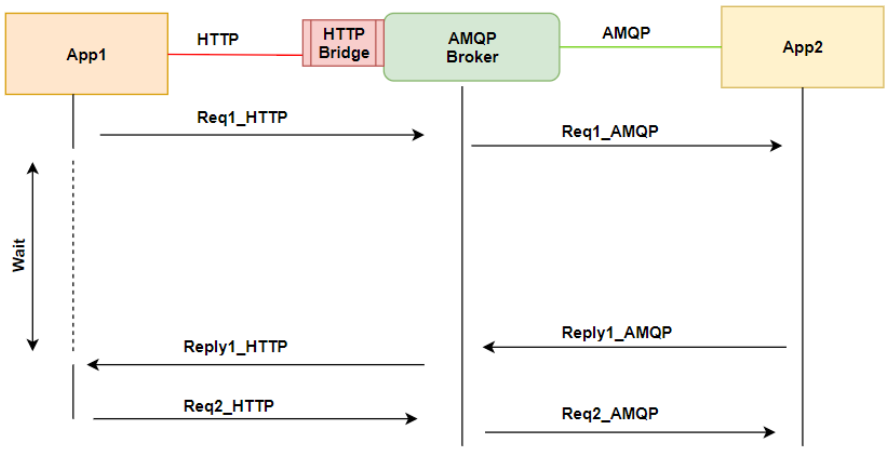

Figure 5.5: Syncrounous client

Another limitation is, HTTP does not have publish-subscribe pattern. This is not an impossible task to do however it needs a lot of work and design to implement. There should be a good interface between the bridge and the clients so that HTTP bridge can interpret if the HTTP side wants to publish something or subscribe to a channel easily. Furthermore, the bridge should handle all the request and responses which are necessary for publishing, subscribing, and receiving messages from the queues. Another limitation is that AMQP has fire-and-forget feature. However, in HTTP, there is a response to every message. Then protocol bridge should detect the messages which do not expect an answer and should not send the response to the AMQP side of the communication. Since both types have rich header fields, the loss of information will be less. Both models have standard header fields regarding message timeliness. For example, AMQP has 'creation time,' 'ttl,' and 'durability,' equivalently HTTP has 'date,' 'age,' and 'max-age.' However, information of the message state such as 'delivered,' 'modified,' or 'accepted' will be gone since HTTP does not have such information. It is also not straightforward to map all the common header fields. Some header fields in HTTP exist in the transport layer of AMQP, and some of them exist in directly message header. For example, while HTTP has header fields such as 'If-Match' or 'Accept,' AMQP handles such information in its transport layer, and therefore, AMQP message does not have these fields in its message frame. The other challenge is to map all the response codes. HTTP has response codes such as '401 – Unauthorized,' '500 – Internal Server Error' and correspondingly AMQP has error header fields such as 'unauthorized-access,' 'internal-error.'

The transformation from HTTP to AMQP have fewer limitations and loss. Since HTPP is just a request-response type, AMQP and its message broker can achieve this. This requires a little bit effort such as creating reply queues and setting 'Reply-to' features. One challenge would be that AMQP is not request-response by default, and it will not send response unless it is told to do. The message broker should set 'Reply-to' field and create a queue for replies so that AMQP side of communication will send replies. If the AMQP side has nothing to reply, then a response should be generated by the protocol bridge so that HTTP side will not be blocked forever. The main limitation is that HTPP has 'Cookies' and 'Caching' mechanism which AMQP does not support. Other than these, most of the header fields can be saved. For example, HTTP has 'MD5-sum' header field. Equivalently, AMQP has footer part which contains information regarding cryptographic checksums. They both have standard fields such as 'Content-Type,' 'Date,' and 'User-Agent' fields. As explained above, AMQP has many different frames and headers in transport layer too. All the information regarding negotiation and transformation is done there. Some of the HTTP headers, such as 'If Match,' 'If Range,' and 'Origin' exists in transformation layer of AMQP.

Security information and payload can be transformed without loss. Both types can deliver many different data types, and they both have 'Content-type' header field. AMQP and HTTP have TSL for encryption of messages. Although authentication of AMQP is more advanced than HTPP, HTTP can have sufficient authorization as well (more than just 'login' and 'password'). AMQP has SASL for authentication, and correspondingly this is done in the header fields of the HTTP. It has 'Authorization' and 'Proxy-Authorization.'

There is one more challenge that has not been discussed yet. Handling of HTTP verbs and URI, i.e. REST style of communication. For this, two cases should be examined. Since HTTP request-response style protocol, there is a clear separation of client and server (or requester and replier).

Case1: Client supports HTTP, Server supports AMQP In this case, HTTP client will send GET and POST requests with a URI. Whenever a POST request comes, HTPP-AMQP protocol bridge should publish a message which contains the data (from POST request) to the queue with the same name as the URI. Afterwards broker will see there is a publish to a queue, and it will send this message to the subscribers of the queue, which will be the server in this case. Whenever a GET request comes, then protocol bridge should find the last published message (by the server) in the queue which has the same name as the URI and respond to the requestor. If the bridge cannot find a message, it should still return something as the client side is synchronous and it will be blocked until a response comes. If the URI name appears for the first time, then necessary subscriptions and queue creations should be done.

Case2: Client supports AMQP, Server supports HTTP In this scenario, whenever client publishes to a queue, AMQP-HTTP protocol bridge should generate a POST request to the server with URI as the same name as the queue name. When an AMQP client subscribes to a queue, protocol bridge should publish the messages to this queue if there is new information from the server. However, when the server has further information, it will not send this information automatically unless it is requested. The protocol bridge should consequently poll the server regularly to know whether there is new information or not. Protocol bridge can do polling by continually sending GET requests with URI as the same name as the queue name. If there is, then the protocol bridge will publish a message to the queue, and then AMQP client will get the message.

5.5 Transformation Between CoAP and MQTT

In table 5.9, the features of the two types are given. In table 5.10, the transformation capabilities are summarized.

The transformation between MQTT and CoAP has challenges and limitations. CoAP has several header fields that MQTT frame does not contain. The 'Options' part of the CoAP message will be lost if it is transformed to MQTT. In contrast, MQTT has 'Retain' and 'Duplicate' fields which CoAP does not have. Another limitation from MQTT to CoAP transformation is that CoAP is transported over UDP whereas MQTT is transported over TCP. That means CoAP cannot use the features that TCP offers, that is also a loss. Moreover, MQTT offers three-level QoS delivery, but CoAP can have only reliable or unreliable delivery. When the 'Transaction type' of the field is set to true, it means, message delivery should be reliable. The bridge should then set the QoS header of the MQTT message as 'QoS Level 0'

Features	CoAP	MQTT
Type	Web transfer protocol	Broker based messaging
Focus	Limited devices	Limited devices
Protocol type	Text based	Binary based
Sent over	UDP	TCP, WebSocket
Popular in	C, Python	C, Python
Interaction Pattern	Request-Response	Publish-Subscribe
QoS Policies	None	3-level QoS
Security	DTLS (SSL for UDP)	Depends on the broker
Header Size	Very Small	Very Small
Supported Payload	Various MIME Types	Various MIME Types

Table 5.9: COAP-MQTT summary

Transformation	Transformation to CoAP	Transformation to MQTT
Is it possible?	Yes, but limited	Yes, but limited
Difficulty	Hard	Hard
Limitations	-Loss of header fields -Possible loss of QoS policies	-Loss of header fields

Table 5.10: COAP-MQTT transformation capabilities

which means 'exactly once delivery.' However, if an MQTT message has QoS 'level 1' or 'level 2,' then this feature will be lost as CoAP message supports only 'reliable' or 'unreliable' delivery. Nevertheless, the payload of both protocols can support many different types of data; there is therefore no limitation here.

In figure 5.6, there is a comparison of standard packet formats of both types. CoAP seems to have more headers fields, but mostly those header fields are optional fields, and they are used for CoAP to HTTP proxy. The essential and critical optional fields are 'Content Type,' 'URI Path,' and 'URI query.' CoAP works like HTTP, and it uses the HTTP Methods 'GET,' 'POST,' and 'DELETE.' The type of the method is given in 'code' field of the message. Similarly, MQTT has message type which can be 'Publish,' 'Subscribe,' or 'Unsubscribe.' CoAP URIs are similar to MQTT topics, and topic names can be used as URIs when transforming. 'Transaction type' holds information about reliability, correspondingly MQTT has QoS level in the header.

Security is not a problem as CoAP uses DTLS which is SSL for UDP and MQTT uses TLS which is SSL for TCP. That means there is no loss as both packets can be successfully encrypted. Besides, both protocols do not have any authentication layer, but they achieve authentication by the header fields. CoAP has tokens for authentication, and MQTT has 'login' and 'password' fields in its connect message. Although it seems simple to transform, there is a challenge to overcome to make the transformation complete. MQTT supports only publish-subscribe, and CoAP supports only request-response.

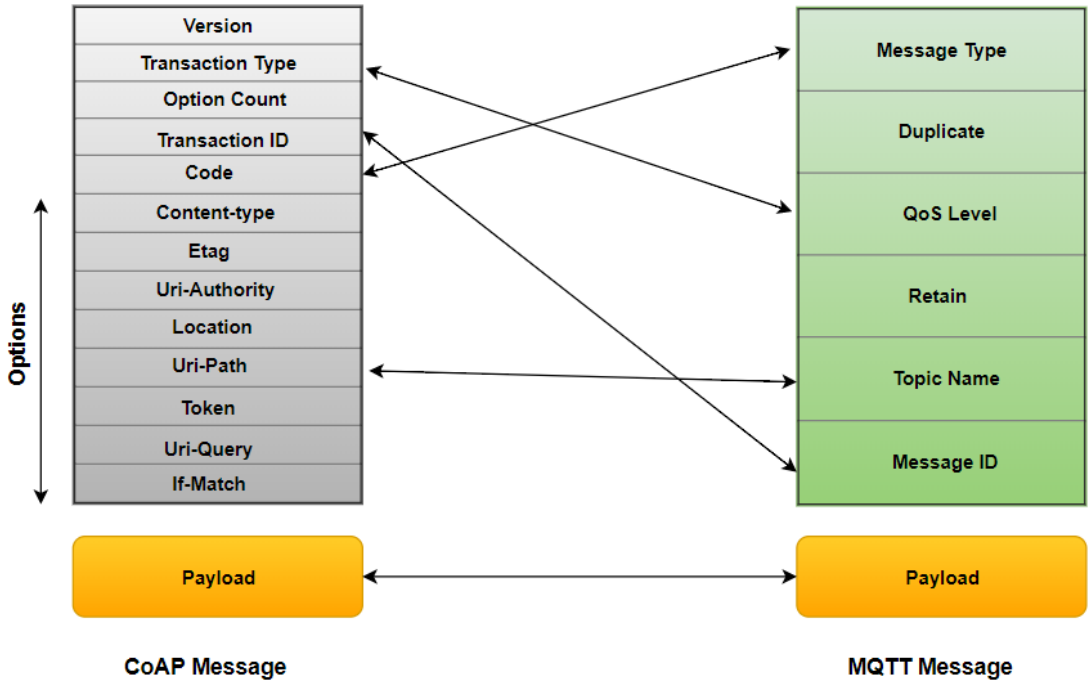

Figure 5.6: CoAP-MQTT headers comparison

The problem of supporting different communication patterns has a solution, but requires much effort. Topic name and URI play the crucial role in the implementation since topic names and URIs are the heart of both protocols. Every-time a CoAP device post something to a URI, an MQTT message will be published to the topic that has the same name as the URI by the protocol bridge. Afterwards, MQTT broker can forward the message to the subscribers of the topic. In a similar manner, when MQTT side publishes to a topic name, the bridge should save the message. As soon as there is a 'GET' request from CoAP device, the bridge should then find the topic name which is sent as a URI, and then it should find the last saved message for that topic. Afterwards, this message will be sent as a response to the 'GET' request of the CoAP device. When all type of MQTT messages ('Subscribe,' 'Unsubscribe,' et cetera) and all CoAP methods ('GET,' 'DELETE,' et cetera) are implemented in this way, communication can be fully established. However, if the CoAP devices are expected to behave like MQTT devices such as subscribing and getting the published messages automatically, then more work and design are needed. Nevertheless, because of 'server push' (i.e. 'observer') feature of CoAP, the server can send notifications or messages when there is new data for the URI that is being observed. Thus, the broker does not need to poll the server; instead, it will register to the URI using 'Observer' feature.

Another significant challenge is that CoAP has a multicast feature (via UDP multicast) and offers dynamic discovery if devices are on the same LAN. CoAP has a particular message type, and when a CoAP device gets this messages, it can convey information regarding the endpoint and the resources. However, this is not possible in MQTT since there is no dynamic discovery feature, and it does not use UDP. This also has a solution, but the

developers should add another smart software that it listens to UDP-multicast packets and responds with the necessary information. This software should be installed on every device in the same LAN so that the protocol bridge can send a particular type of messages to have the information about the devices.

In figure 5.7, there is a scenario describes the transformation need between CoAP and MQTT. The IoT system has embedded devices and these devices are programmed to use CoAP. The devices send information to the first gateway device using CoAP. The gateway device collects information, analyses and prepares for the web server. This information will be shown to the end user on the browser. However, the company may need to use the second gateway device with MQTT for different purposes. In this case, CoAP devices may not be capable of processing MQTT and CoAP requests at the same time and consequently, they need a transformation. Although it is difficult to do the transformation, it is possible with limitations as explained above. Devices will continue to send CoAP messages, and MQTT bridge will transform the messages for the new gateway machine.

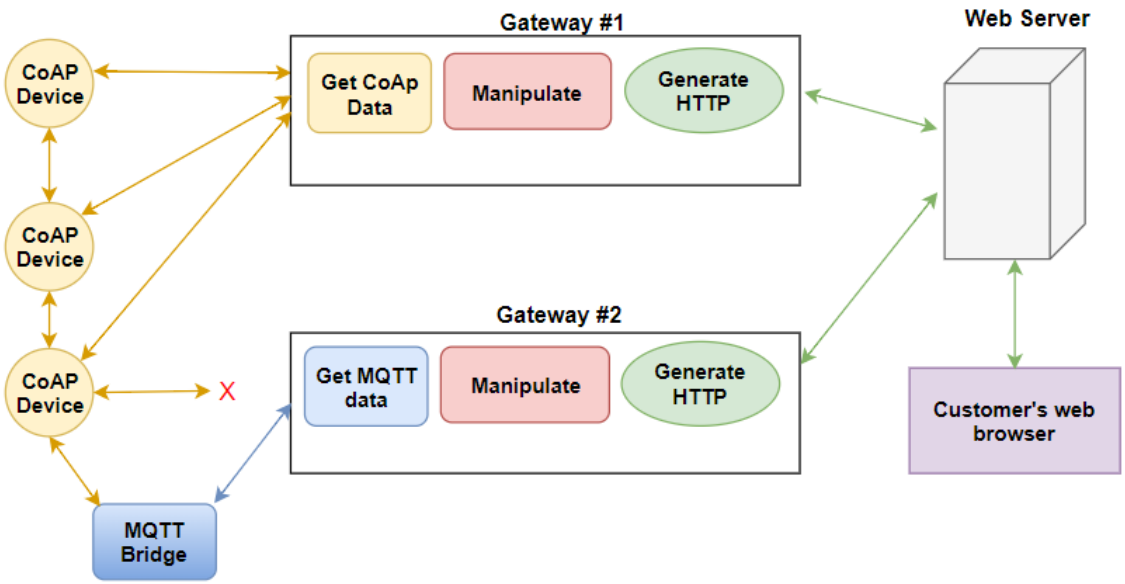

Figure 5.7: CoAP-MQTT scenario

5.6 Transformation Between CoAP and HTTP

As CoAP was designed according to the REST [Fie00] architecture, CoAP offers similar functionality as HTTP protocol offers. In addition, standards of both CoAP and HTTP are defined by the same organizations. For these reasons, it is straightforward to map from CoAP to HTTP and from HTTP to CoAP [SHB14]. The core difference is that CoAP is intended for the use of resource-constrained internet devices, such as wireless sensor network nodes. Since CoAP aims for constrained devices, it uses UDP and less header size.

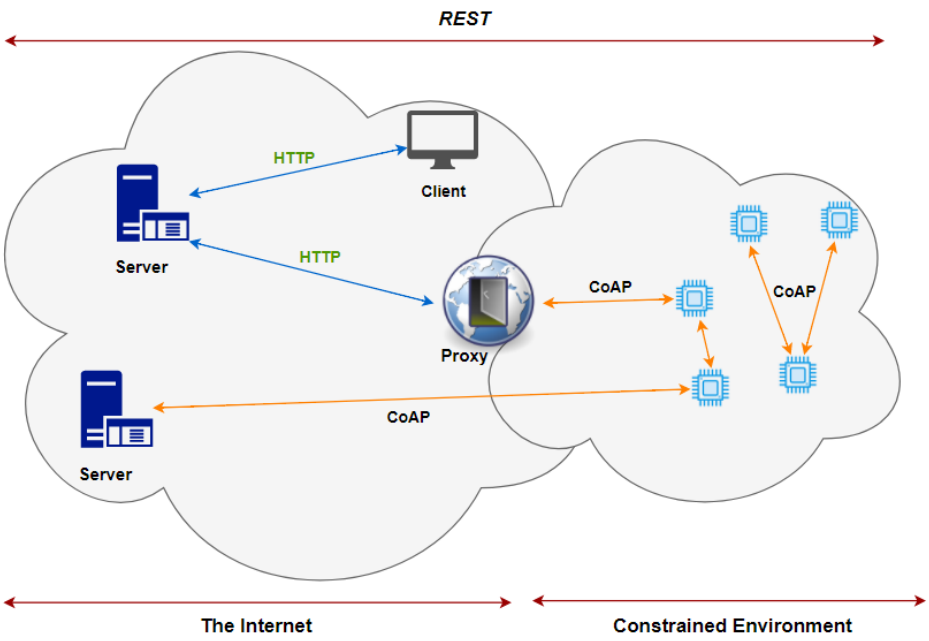

Figure 5.8: CoAP-HTTP proxy (adapted from [Mil16])

This means that transformation from CoAP to HTTP there will not be any loss. In contrast, HTTP to CoAP transformation may a few losses.

The first limitation is that the features offered by TCP will no longer be available and will be replaced by UDP. However, this limitation can be decreased to a certain level, as CoAP can have extra reliability feature. The other limitation is that some header fields of HTTP do not exist in CoAP. Nevertheless, these missing header fields are not the critical ones as CoAP have many optional header fields to limit the information loss when mapping to HTTP.

In figure 5.8, CoAP to HTTP proxy is seen. Since CoAP to HTTP or HTTP to CoAP transformation is common in the IoT, mapping between HTTP and CoaP has already been published by IETF[1] and CoRE[2]. The documentation for the proxy is available by Castellani, Loreto, and Rahman [CLR17]. Castellani, Fossati, and Loreto [CFL12] also published a paper about HTTP-CoAP proxy in which they discuss how to map CoAP to HTTP, and what the currently open issues are (especially in the deployment and the security areas). Several implementations of CoAP-HTTP proxies are given the chapter 3.

[1]Internet Engineering Task Force - https://www.ietf.org/
[2]Constrained RESTful Environments - https://datatracker.ietf.org/wg/core/about/

5.7 Others

The rest of the transformations will be summarized shortly to avoid repetitions. Some transformations are not possible, or some transformations are similar to the transformations which are explained above.

5.7.1 Transformation of DDS

Because of its different and complicated characteristics, transformations involving DDS will be considered as not possible. It is not impossible, but it requires a considerable amount of effort. In DDS architecture, there is an imaginary 'Global Data Space' where everyone publishes or receives information from. 'Global Data Space' is universal and fully distributed data cache and smart software which is running on every member of DDS and makes DDS broker-less. Due to 'Global Date Space,' DDS allows dynamic discovery. When a device joins or leaves the system, everyone is notified. This architecture is very different than the designs of the other protocols.

To emphasize the difference with other communication protocols, DDS can be compared to AMQP which is an advanced messaging protocol. According to CEO of RTI[3], Schneider [Sch13b], DDS provides the real-time, many-to-many, and managed connectivity required by high-performance machine applications. The Advanced Message Queuing Protocol focuses more on enabling fast and reliable business transactions. The middleware of AMQP focuses on tracking all the messages and ensuring that each message is delivered as intended, regardless of failures or reboots, while the middleware of DDS (Global Data Space) matches publishers to subscribers based on their interested data types, topics, and QoS. The result is fast, direct, and controlled communications. While AMQP gives more importance not to lose messages, DDS cares speed, scaling, timing control, and flexible delivery. As opposed to AMQP, it strives to simplify applications by leaving much of the state management to the middleware [Sch13b]. From DDS clients' point of view, it looks like, they are reading or writing from their local memory. Actually what happens is, middleware brings the needed data and updates the local memory of the other client machines if there is new information [Obj13].

Another critical difference is that AMQP is a message-centric protocol while DDS is a data-centric protocol. Although the middlewares of both protocols serve to connect distributed systems, the approaches are entirely different. When a message-centric design is used, the primary role of the middleware is to ensure that messages get to their intended recipients. Data-centric technologies instead analyze and inspect the data itself and can control different types of state behavior. The responsibility of DDS is not only to deliver the data, but to guarantee that all the nodes have a synchronized and common perception of the data's state and value [Sch13b].

[3]Real-Time Innovations (RTI) (A DDS vendor) - https://www.rti.com

Even though both protocols support publish-subscribe natively, the usage is different. For instance, a DDS publisher will only announce that it sends 'temperature' data. The middleware will then find all subscribers who are interested in 'temperature' type of data and make the appropriate data-bus contracts. However, an AMQP publisher may publish a message containing 'temperature' to the channel. Then the broker must forward, through configuration, to the subscribed recipients of that channel [Sch13b].

5.7.2 Transformation of STOMP

Since STOMP and AMQP are similar in many ways and they are categorized under the same category, most of the challenges and limitations will be similar to the transformation of AMQP. The main difference is that AMQP has more complex message format and advanced transport layer. For this reason, it has more header fields, and it gives more information regarding the state of the messages. In section 5.1, the differences and limitations are detailed. Except for those differences, transformation to STOMP will be similar to AMQP. In order to avoid repetitions, the transformation of STOMP will not be further considered. The limitations of transformation between AMQP and STOMP can be read, and then possible limitations and difficulties of the transformation of STOMP between the other types can be inferred.

5.7.3 Transformation of XMPP

In the section 5.3, it was explained how difficult to implement transformation between XMPP and AMQP. Even transformation from/to a strong protocol AMQP, there are too many limitations and difficulties. Further transformations will not be considered as there will be too many limitations and challenges for the other protocols. Transformations involving AMQP can be read, and further limitations of other communication types can be deduced by comparing the communication type with AMQP. There could be too many repetitions if the transformation capabilities between XMPP and other communication types were detailed.

5.7.4 Transformation of WebSocket/HTTP2

Websocket and HTTP2 are just binary transport protocols that multiplex numerous streams going over a single (optionally TLS-encrypted) TCP connection. The contents of each stream are HTTP 1.1 requests and responses which are just encoded, wrapped, and carried differently to gain performance. These types are not applicable to transformation as they just change the way the packets are carried. However, if these types are used for the carrying or transporting the messages or packets of different protocols, limitations or challenges of the transformation of the protocols may be changed. For instance, if the communication type has no built-in security such as STOMP, then WebSocket can give additional security layer which is TLS (SSL). Transporting STOMP messages over WebSocket can eliminate

potential security lost. Furthermore, carrying packets over WebSocket or HTTP2 may have two main advantages unless the communication protocol already supports. The first advantage is that clients will no longer need to poll the server since WebSocket and HTTP2 have 'server push' feature which sends the new information automatically without waiting for requests. Another advantage is that the requests are not blocking requests. When the client sends requests using WebSocket or HTTP2, the client will no longer be blocked. These two important features are missing in HTTP over plain TCP sockets. If the HTTP packets are sent over WebSocket or HTTP2, clients will not need to poll, and they will not be blocked. These problems are explained in section 5.4.

5.7.5 Transformation of RPC

Remote procedure calls are the communication type where one party will remotely execute the function of the other party with given parameters. This communication type is very tightly coupled and does not have any features. Transforming from RPC to a random communication protocol has no limitations since the only thing that the new communication type is to send function name and parameters to the other side. In contrast, when a communication type is transformed to RPC, then it will lose all the features and header fields. Apart from these limitations, there is a severe design problem if one side of communication is RPC and the other side is a messaging or different kind of communication types. If this transformation is somehow needed, the architecture and the design of the system should be reviewed instead of transforming. For this reason, this communication type will be considered as transformation is not possible.

5.7.6 Transformation of Modbus

The main aim of Modbus is to establish the connection between an industrial slave and a master device. A device which measures the pressure of the water tank at a specified period is an example of a slave device. Those kinds of devices have very limited functionalities such as only reading from its registers or writing data to its registers. Furthermore, they are specialized to use one of the automation protocols such as Modbus. The master device can send only a few commands such as read or write, by setting the 'function code' field of the Modbus frame. In a typical IoT scenario, Modbus master polls the slaves, and when there is new data, it delivers this data to a protocol such as MQTT, and the protocol publishes the information. Modbus and the other communication types which are explained above have very different application domains. Thus, they cannot be comparable, and there will be no such scenarios that a device no longer supports Modbus and needs transformation. As a result, the transformation of the Modbus will not be considered in this thesis. Transformation of Modbus can be considered only between other industrial or automation technologies communication types such as Profibus[4]. However, this analysis of transformation between industrial devices is out of the scope of this thesis.

[4]Process Field Bus - https://www.profibus.com/

6 Implementation of Prototype for Detection of Transformation

After researching the analysis of the transformation capabilities, a web application is developed to detect the transformation need when the environment is changed. In chapter 1, there is a motivating scenario which explains what happens when the environment changes. This application aims to recognize whether the transformation is needed or not when the components move to a different environment. In case the transformation is required, the difficulty of the transformation and the limitations of each communication that is supported by the new environment will be given as output.

In this chapter, first the developments tools that are used to make this application will be given. Then the architecture of the application will be explained. There, the critical files, essential functions, and general workflow of the application will be provided. Finally, the core functionality of the application and how the application works will be explained.

6.1 Development Tools

Java[1] (JDK 8) is used for the back-end. Angular 5[2] and Bootstrap 4[3] is used for the front-end. All the Java and Angular codes are programmed in Intellij IDEA[4] which is an Integrated Development Environment (IDE).

The application is a web application because it will be straightforward to integrate to the Eclipse Winery[5]. Eclipse Winery is a web-based environment to graphically model and manage TOSCA topologies and plans [KBBL13]. The motivation scenario, that is given in the introduction chapter 1, can be modelled in TOSCA as a topology. This topology can easily be made using Winery which offers to generate TOSCA models graphically.

Winery also includes the splitting functionality. Before splitting the topologies to the new environment, this prototype can be used as decision support for distributing the components. It can also be helpful to know whether the previous communication between applications can be transformed to the communication protocols of the new environment.

[1] Java - https://www.java.com/
[2] Angular - https://angular.io/
[3] Bootstrap - https://v4-alpha.getbootstrap.com/
[4] Intellij IDEA - https://www.jetbrains.com/idea/
[5] Eclipse Winery - https://projects.eclipse.org/projects/soa.winery

If the transformation is not possible, the splitting process can be aborted or can be retried with another environment. That is why it is important to make the prototype integrable to the Winery which is written in Java and Angular.

The most used third-party libraries are Jackson-Databind [6] for the conversion between JSON files and JSON strings. It is licensed under Apache-2.0. In the Angular part, Multiselect-checkbox-dropdown[7] package is used for drop-down lists on the UI. This package is licensed under MIT license.

6.2 Architecture

In this section, the architecture of the prototype is explained. The information for the users and developers as well as the source codes of the project can be found at this GitHub link[8].

In figure 6.1, the architecture and the main components of the program can be seen. There are two main parts; back-end and front-end. Between these two components an HTTP connection is estabished.

6.2.1 Back-end

The back-end is written in Java using JDK 8. In the back-end, there are three main components. They are *JsonReaderWriter*, *RestServlet*, and *Analyzer*. Other than these essential components, rest of the files contains only the class definitions for the objects that are used in the applications. They are *'Communication.java,' 'Environment.java,' 'DropDownListItem.java,'* and *'AnalysisResult.java.'*

- ***JsonReaderWriter:*** This component is a converter which does the conversion between Java Run-time Objects, JSON-String, and JSON-Files. The content from the front-end comes as a string (JSON-format) in the HTTP body. The content should be saved as a JSON file. The front-end needs the content as well. The JSON files should be read into the memory and converted to string. Then the back-end sends the content as a string in HTTP-Response body. The files should also be converted to objects so that the *Analyzer* can work with them. All these conversions, writing to files, and reading from files are done using this component.
- ***RestServlet:*** This component handles all the HTTP requests and responses. The front-end can send two different types of HTTP requests, namely *'GET'* and *'POST.'* When there is a *'GET'* request, this servlet extracts the file location and information according to the parameters and converts the file content into a string using *JsonReaderWriter*

[6] Jackson-Databind - https://github.com/FasterXML/jackson-databind
[7] Angular Package - https://www.npmjs.com/package/angular2-multiselect-checkbox-dropdown
[8] GitHub Account - https://github.com/mustafaakilli/master-thesis-project

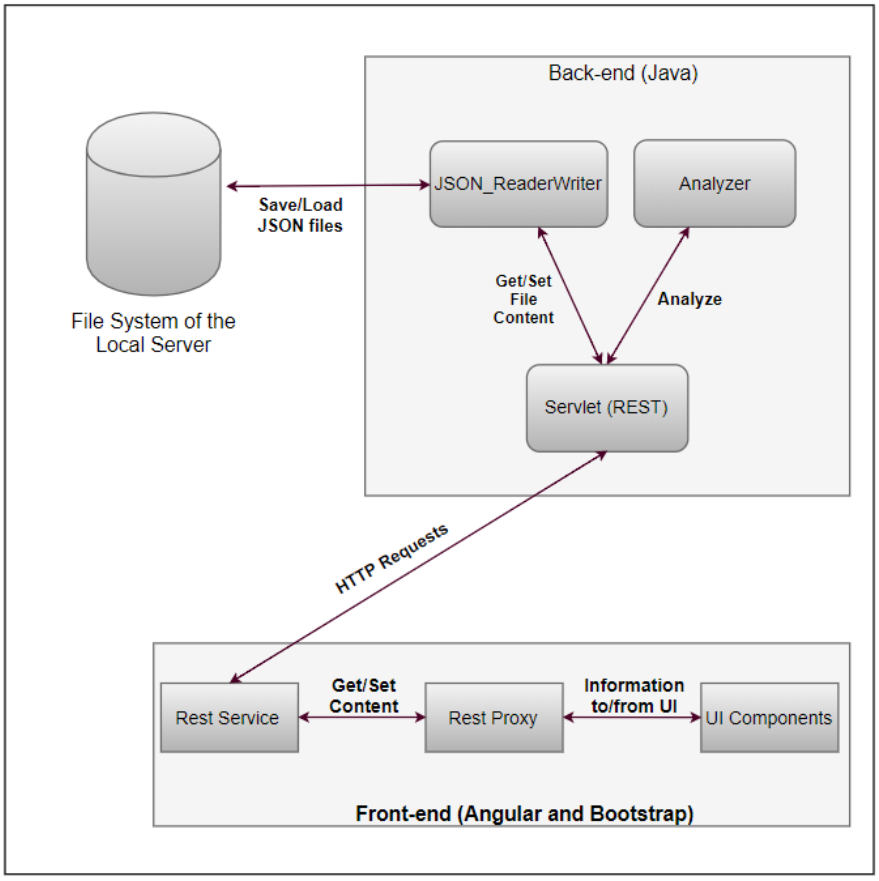

Figure 6.1: Main components of the application

component. Then it adds the resulting string to the HTTP-Response body and sends it back to the front-end. When there is a *'POST'* request, there may be two possibilities. The first one is that new content is needed to be saved. The JSON content is read as a string and using *JsonReaderWriter* component, the content is written to the file and saved in the local file system of the server. The other possibility is that the analysis results are expected. In this case, the servlet converts all the necessary contents (environments and communications) from the files into the Java objects *JsonReaderWriter*. Then it initializes the *Analyzer* component.

- ***Analyzer:*** This component makes the analysis. It has the Java object form of the old communication, the new environment, and its supported communication protocols. Using these three objects, it analyses and determines that whether there is a need for transformation or not. If the transformation is required, then it checks if it is possible or not possible. If the transformation is possible, then it finds the difficulty of the transformation and the possible losses of the transformation. This process is done for each supported communication of the new environment. Then it sends the results in JSON format to the *RestServlet* component which then returns the results in string format to the front-end.

6.2.2 Front-end

The front-end is programmed in TypeScript, HTML, and CSS using frameworks Angular 5 and Bootstrap 4. The application has three different tabs, and for each tab, there is an Angular UI-Component. These components are *Analyzer*, *Environments*, and *Communications*. On the *Analyzer* tab, old environment, old communication, and the new environment can be selected. Then the results of the analysis will come from the back-end. On the *Environments* and *Communications* tabs, new environments or communications can be created. Optionally, the existing environments or communications can be customized.

Other than components, there are also two other important files. These are *RestService* and *RestProxy*. *RestService* makes the HTTP requests. It gets the necessary information for the request, i.e. the parameters and the body from *RestProxy*, and makes the HTTP calls. It also returns the responses to *RestProxy*. *RestProxy* is the heart of the front-end and is the bridge between the UI-Components and *RestService*. It converts all the JSON content, that is coming from the back-end, into the TypeScript objects which are understood by the UI-Components. Furthermore, it generates HTTP requests from the information that is given by UI-Components. Then it delivers to *RestService* which makes the HTTP call.

6.3 Core Functionality

After the application is deployed to the local web-server and the address of *localhost* is entered to the web browser such as Chrome; then the application will be started. Initially, the first tab is selected by default.

In the *Analyzer* tab, there is a form which has the fields 'Old Environment,' 'Old Communication,' and 'New Environment.' All the fields are mandatory and must be filled. The user can select the environments from the already created environments. When 'Old Environment' is selected, 'Old Communication' drop-down list will be filled. The available communications are the supported communication protocols of the 'Old Environment.' Then the user should select the 'Old Communication' and a new environment. Then submit button will be enabled, and as soon as it is clicked, the result will be shown on the right side of the page, in the 'Output box.'

In the figures 6.2 and 6.3, the *Analyzer* tab initially and an example output can be seen.

In the second tab (*Environments* tab), a new environment can be created, or an already created environment can be customized. It has a form which contains the fields, 'Environment Name' and 'Supported Communications.' 'Environment Name' must be unique. Otherwise, it may overwrite the existing file. 'Supported Communications' field is a drop-down list which contains all the available communication protocols. The user can customize an already existing environment. In that case, the supported communications part will be automatically filled from the selected environment.

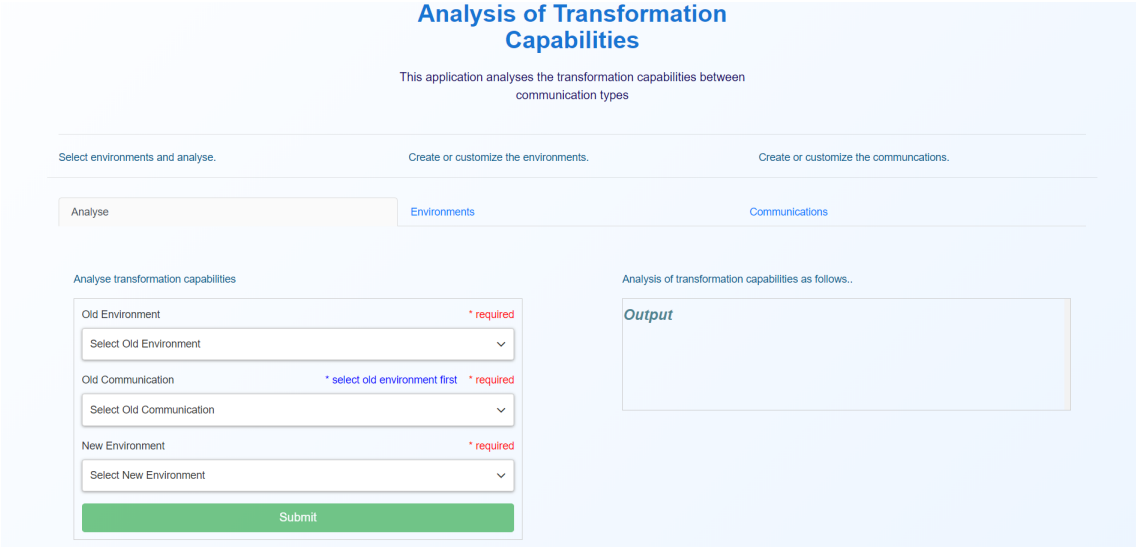

Figure 6.2: Analyzer tab (before filling the fields)

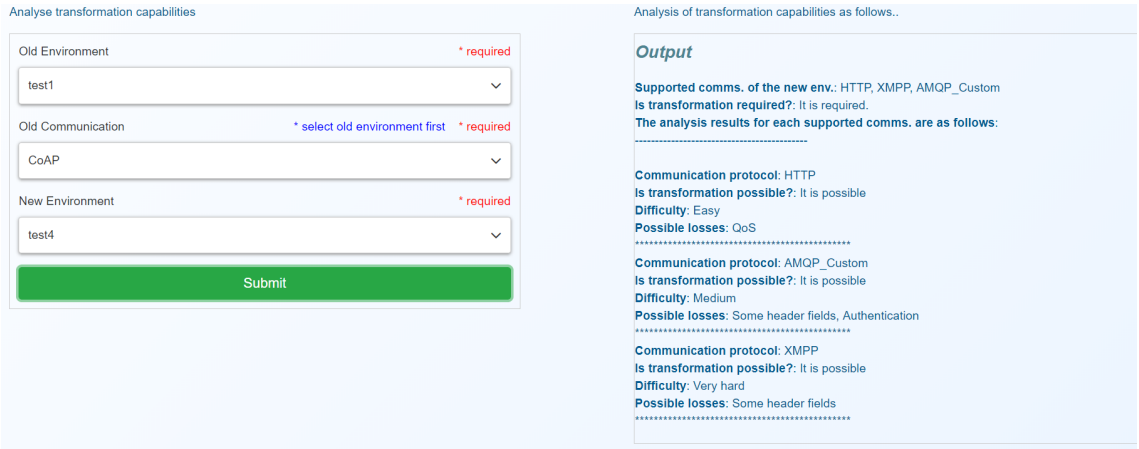

Figure 6.3: Analyzer tab (after submitting)

In figure 6.4, the *Environments* tab initially is shown, and figure 6.5 shows the *Environments* tab when its form is filled.

In the *Communications* tab, similar to the *Environments* tab, a new communication can be created, or an already created communication can be customized. It has a form which contains the fields, 'Communication Name' and ten other drop-down lists. The name of the communication must be unique again.

While some of the drop-down lists are mandatory, a few of them are not mandatory. 'Communication Type,' 'Communication Pattern,' 'Request Type,' 'Header Size,' 'Payload Type,' and 'Payload Size' must be filled.

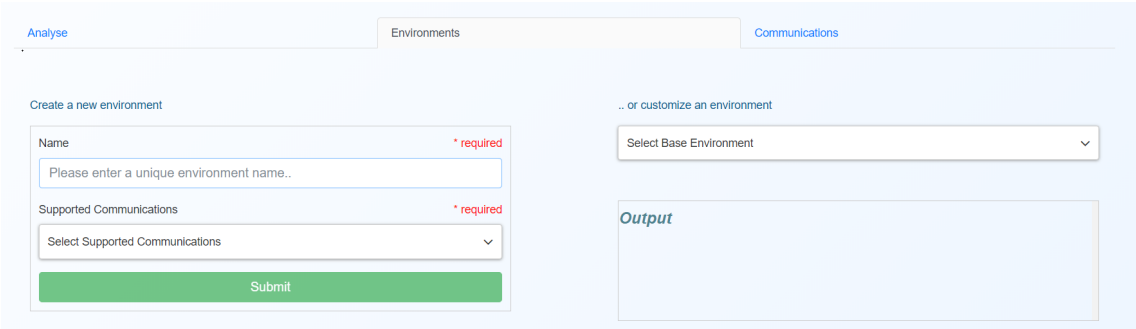

Figure 6.4: Envrionments tab (before filling the fields)

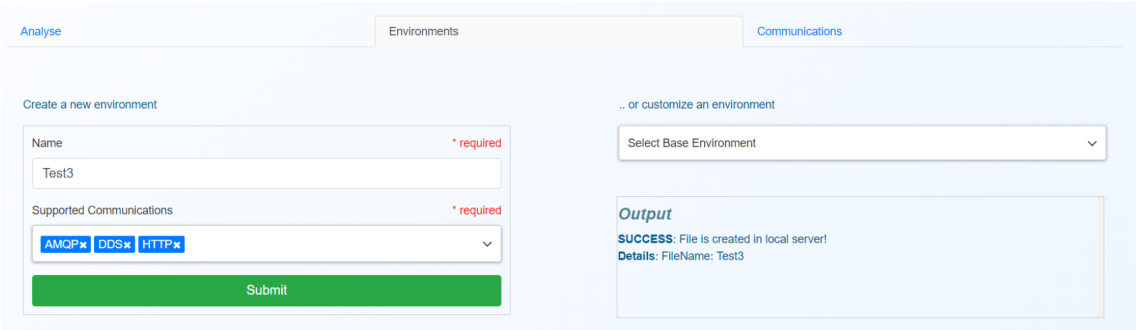

Figure 6.5: Environments tab (after filling the fields)

- ***Communication Type*** can be 'Client/Server (One-directional),' 'Client/Server (Bi-directional),' 'Brokered Messaging,' 'Brokerless Messaging,' 'Data-Centric Brokered Messaging,' or 'Data-Centric Brokered Messaging.' It can have only one selection. Multi-selection is disabled for this drop-down.
- ***Communication Pattern*** can be 'Publish/Subscribe,' 'Request/Response,' 'Fire/Forget,' or 'Multicasting.' Multi-selection is allowed.
- ***Request Type*** can be 'Blocking (Sync.)' or 'Not Blocking (Asnyc.).' Multiselection is allowed.
- ***Header Size*** can be 'Small,' 'Medium,' or 'Large.' Multi-selection is not possible. If the application sometimes uses large header size and sometimes uses very small portion of the header, then the application needs a large header size. That is why there is no need for multi-selection in drop-down list.
- ***Payload Type*** can be 'Text,' 'Binary Array,' or 'Various MIME types.' Multi-selection is allowed since application may sometimes use text and sometimes use image for payload.
- ***Payload Size*** can be 'Small' or 'Large.' Again multi-select is not allowed, as the same reason for 'Header Size.'

'Securtiy Type,' 'Authentication Type,' 'QoS Type,' and 'IoT Level' can be left empty. For instance, if the 'Security Type' is empty, that means there is no security.

- ***Securtiy Type*** can be 'DTLS,' 'TLS,' or 'Has own complex layer.' Only single selection is allowed.
- ***Authentication Type*** can be 'Simple Login/Passcode,' 'SASL,' or 'Has own complex layer.' Only single selection is allowed.
- ***QoS Type*** can be 'TCP,' '3-level QoS,' or 'Complex.' Multi-selection is allowed.
- ***IoT Level*** can be 'Device to Device,' 'Device to Gateway,' 'Gateway to Server,' and 'Server to Server.' Only single selection allowed.

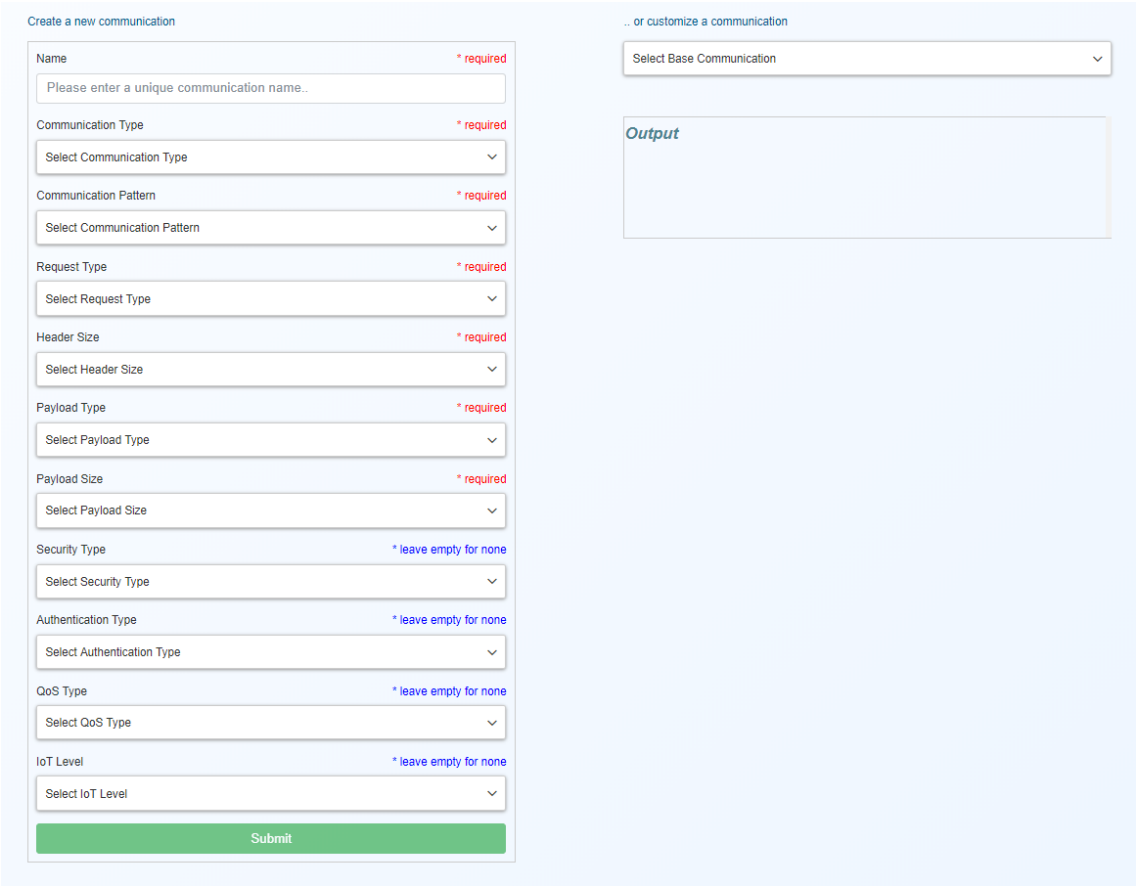

Figure 6.6: Communications tab (before filling the fields)

In the figures 6.6 and 6.7, the communication tab initially and after filling its form can be seen.

Unlike *Environments* tab, in *Communications tab*, there will be default communication protocols: AMQP, STOMP, MQTT, HTTP, CoAP, XMPP, and DDS are pre-added. They can be customized or a brand new communication can be created. If a brand new communication is built, then the results of the analysis will not be perfect, and they will be just guesses since

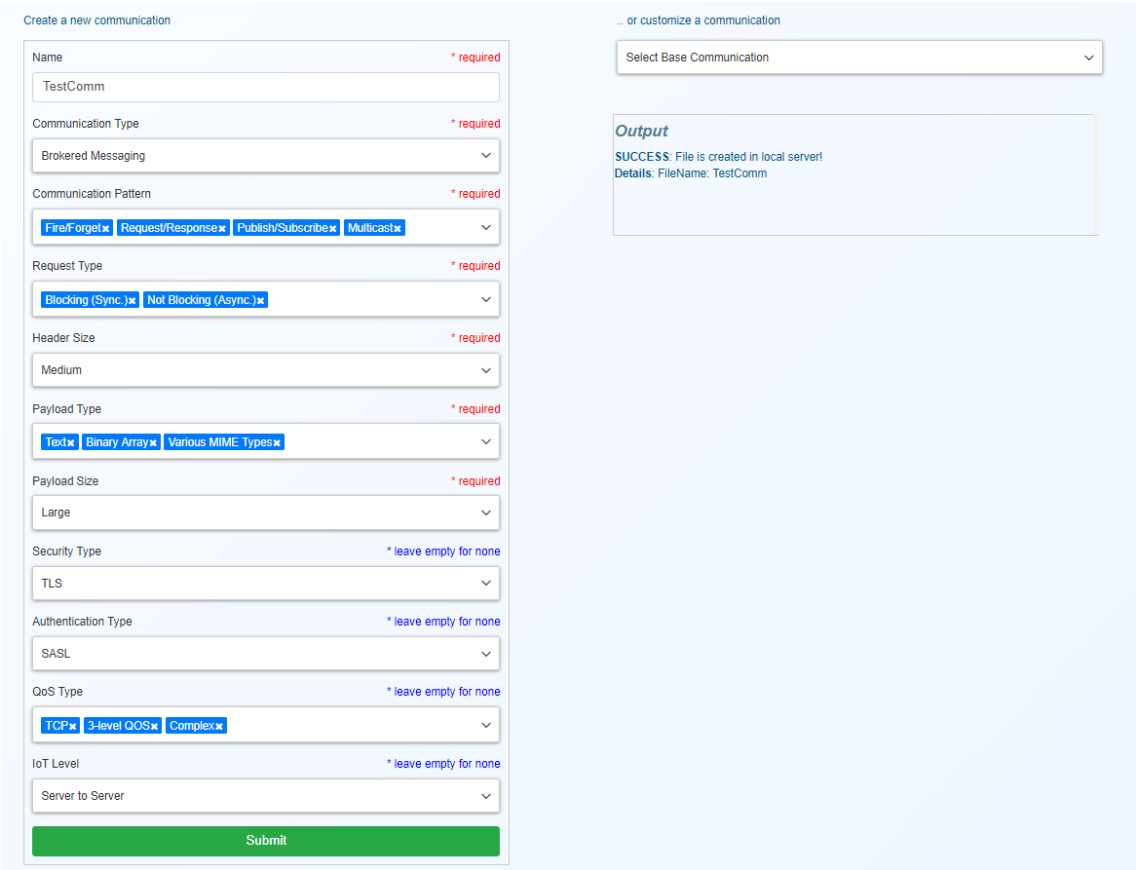

Figure 6.7: Communications tab (after filling the fields)

the new protocol may have a very different architecture and header fields. The analysis results for the primary types mentioned above and the customized communications from these types will have accurate results. The results will be consistent with the findings of this thesis.

Communications can also be customized. For example, an application which uses HTTP may only need a few header fields and may always use text type in the payload. In this case, a new version of HTTP can be created by customizing it. When a communication type is customized, the 'Communication Type' field will be disabled, and further changes to this drop-down list are not allowed. 'Communication Pattern' will be pre-selected and can be changed. The rest of the fields can be selected freely, but the drop-down list selectable items may be limited. In other words, some of the values described above may not be selectable. For example, when AMQP is customized, the 'IoT Level' drop-down list will not contain all the four elements. Since AMQP is not used in 'Device-to-Device' level or 'Device-to-Gateway' level, the selectable items will only be 'Gateway to Server' and 'Server to Server.' The communication name will be assigned automatically to a unique random string, and it can be edited.

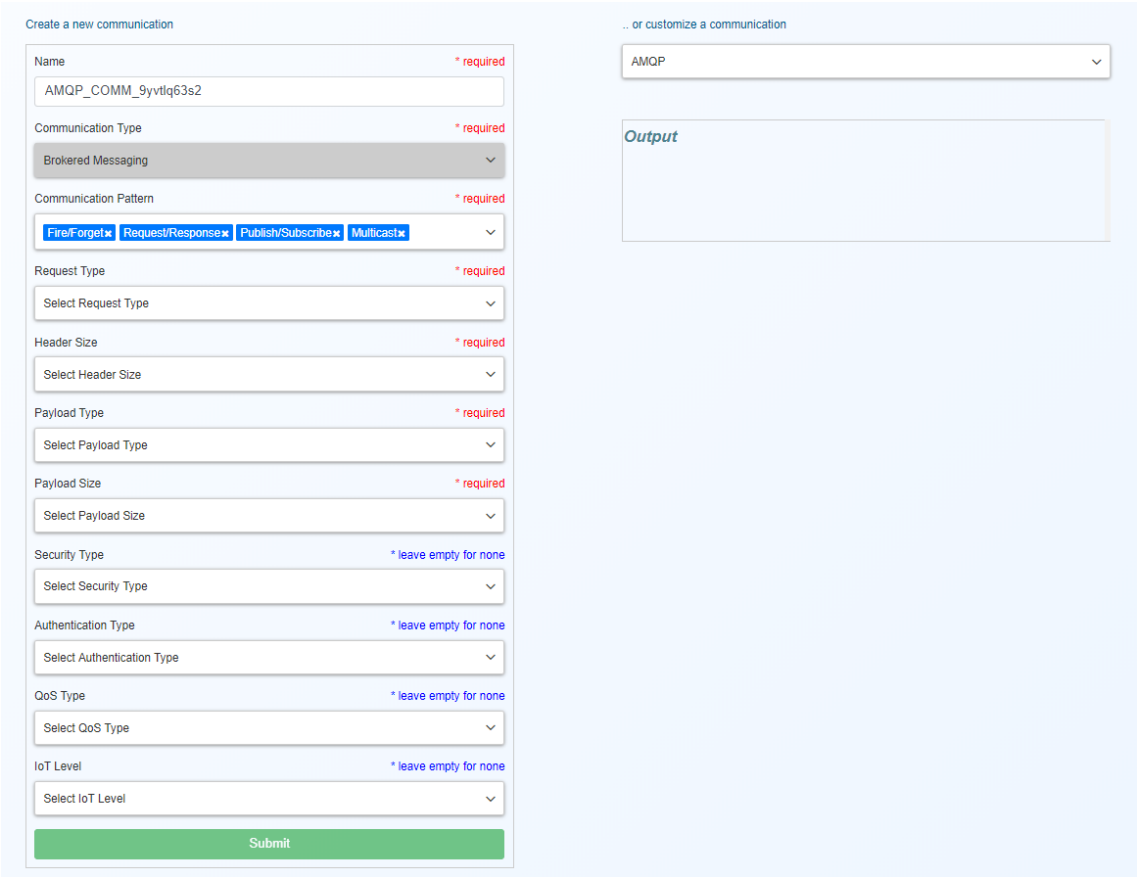

Figure 6.8: Communications tab (customizing a communication)

In figure 6.8, an example case can be seen. The selected protocol for customization is AMQP.

The 'Communication Type' field is disabled because it is related to the architecture of the communication. Customizing a communication should not change the architecture and the implementation of the communication protocol. Some applications may use only particular features of the protocols. For instance, two applications communicating over AMQP may just use request-response pattern, text as payload, and a small part of the headers. In that case, the transformation of this variation of AMQP is simpler than transforming the all features and header fields of AMQP. That is why the analysis results will be different to this variation. The customization feature of this application can be used as depicted in figure 6.8 and a variation of AMQP can be created.

7 Conclusion and Future Work

The goal of this thesis was to analyse the transformation capabilities between communication types of cloud application components. First, the communication protocols were categorised, and their characteristics were explained. Second, transformation capabilities of the communication protocols were analysed. Finally, a prototype that detects whether the transformation is needed when the environment changes and offers decision support for redistributing the components was implemented.

In chapter 4, the communication types were categorised into two categories: one-directional communications and bidirectional communications. The application layer protocols were predominantly considered. Lower level protocols and industrial protocols were not discussed, as they are beyond the scope of this thesis. Only Modbus was explained to demonstrate an example from industrial protocols. In the first category, the features and characteristics of HTTP, REST, RPC, and Modbus were provided. In the second category, AMQP, STOMP, MQTT, XMPP, DDS, CoAP, HTTP2, WebSockets, and messaging APIs were explained.

After the categorisation and the characteristics of the communication types, the transformation capabilities between the most well-known communication types were described in 5. For each analysis of communication types, the determination of whether the transformation was possible, suggestions on how to perform the transformation, and the limitations and challenges of the transformation were explained. Some of the transformations are easy to transform, while others are difficult. Transforming from simple protocols to a sophisticated protocol is less challenging since there are not as many steps to implement. For example, the transformation from HTTP to AMQP or from MQTT to AMQP is not that difficult to achieve, as AMQP supports almost all the features and characteristics that the others have. However, it is not as straightforward to transform from a complicated protocol to a simple protocol. For instance, the transformation from AMQP to HTTP is problematic, as AMQP has many characteristics that HTPP does not have. Such characteristics should be implemented using the features of HTTP, and that is a challenging task. Regardless of the communication type, converting header fields of a protocol to header fields of another protocol is not easy. The difficulty of the transformation is thus not only bound to the complexity of the protocols. Sometimes the transformation is still laborious even if both protocols support many features. For example, the conversion from XMPP to AMQP is arduous, although AMQP and XMPP are both powerful protocols. The reason is that some protocols implement the same features in an entirely different way. While AMQP handles publish-subscribe by making use of the middleware, XMPP does this using a different message type and its headers. Finally, some transformations are not even possible, such as

the transformation of the DDS, which has different architecture and implementation that makes the transformation not feasible.

In the motivating scenario in chapter 1, an example was given. The example explained a scenario in which the communications may need a transformation. A prototype that detects such transformations was implemented. In chapter 6 the details of the prototype were provided. The prototype was a web application developed using Java and Angular. The user can define communication types and customise the already defined types, such as MQTT or CoAP. The user can also create environments and set his or her supported communication types. Finally, the user can analyse the transformation capabilities by selecting environments and the previous communication between them. The results will show whether the transformation is possible or not. For each supported communication protocol of the new environment, there will be analysis results. The results contain the difficulty and potential losses of the transformations.

For future work, the prototype can be added to the Winery tool. In Winery, there is a functionality called *'Split and Match,'* developed by Saatkamp et al. [SBKL17]. The function automatically splits the topology if a new environment is needed. Before splitting, the application developed in this research can be used to determine whether the split is possible or not by checking the analysis. Further work, after the integration with Winery, would be to have the fields in the prototype become filled automatically by obtaining the information from the topology. This application can then be called automatically when the splitting functionality is used. In this way, the whole process would be fully automated. In addition, there is room for further improvements to the application. For instance, industrial communication protocols can be analysed and added to this application. Another point to improve would be adding more detailed error handling. Such improvement points can be found in the 'Readme' file of the source code at this GitHub link[1]. Lastly, the same study analysis can be performed by considering only industrial communication protocols.

[1] GitHub Account - https://github.com/mustafaakilli/master-thesis-project

Bibliography

[AGM15] A. Al-Fuqaha, M. Guizani, M. Mohammadi. "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications." In: *IEEE COMMUNICATION SURVEYS & TUTORIALS* 17.4 (2015) (cit. on pp. 26, 38).

[BBKL14] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. "TOSCA: Portable Automated Deployment and Management of Cloud Applications." In: *Advanced Web Services*. New York: Springer, Jan. 2014. Chap. TOSCA: Portable Automated Deployment and Management of Cloud Applications, pp. 527–549. ISBN: 978-1-4614-7534-7. DOI: 10.1007/978-1-4614-7535-4_22 (cit. on p. 11).

[BBLS12] T. Binz, G. Breiter, F. Leymann, T. Spatzier. "Portable Cloud Services Using TOSCA." In: *IEEE Internet Computing* 16.03 (May 2012), pp. 80–85. ISSN: 1089-7801 (cit. on p. 11).

[BN84] A. D. Birrell, B. J. Nelson. "Implementing Remote Procedure Calls." In: *ACM Transactions on Computer Systems* 2.1 (Feb. 1984) (cit. on p. 33).

[BP15] M. Belshe, R. Peon. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. Ed. by M. Thomson. May 2015. URL: http://httpwg.org/specs/rfc7540.html (cit. on p. 51).

[Bra89] R. Braden. *Requirements for Internet Hosts : Communication Layers*. Internet Engineering Task Force, 1989. URL: https://tools.ietf.org/html/rfc1122 (cit. on p. 15).

[BSW14] A. Brogi, J. Soldani, P. Wang. "TOSCA in a Nutshell: Promises and Perspectives." English. In: *Service-Oriented and Cloud Computing*. Ed. by M. Villari, W. Zimmermann, K.-. K. Lau. Vol. 8745. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pp. 171–186. ISBN: 978-3-662-44878-6. DOI: 10.1007/978-3-662-44879-3_13. URL: http://dx.doi.org/10.1007/978-3-662-44879-3_13 (cit. on p. 11).

[CFL12] A. P. Castellani, T. Fossati, S. Loreto. "HTTP-CoAP cross protocol proxy: an implementation viewpoint." In: *2012 IEEE 9th International Conference on Mobile Ad-Hoc and Sensor Systems (MASS 2012)*. Vol. Supplement. Oct. 2012, pp. 1–6. DOI: 10.1109/MASS.2012.6708523 (cit. on pp. 27, 68).

[CLR17] A. Castellani, S. Loreto, A. Rahman. *Guidelines for Mapping Implementations: HTTP to the Constrained Application Protocol (CoAP)*. Feb. 2017. URL: https://tools.ietf.org/html/rfc8075 (cit. on pp. 27, 49, 68).

[CMM07] G. Cugola, M. Migliavacca, A. Monguzzi. "On Adding Replies to Publish-subscribe." In: *Proceedings of the 2007 Inaugural International Conference on Distributed Event-based Systems*. DEBS '07. Toronto, Ontario, Canada: ACM, 2007, pp. 128–138. ISBN: 978-1-59593-665-3. DOI: 10.1145/1266894.1266918. URL: http://doi.acm.org/10.1145/1266894.1266918 (cit. on p. 29).

[Cur04] E. Curry. *Message-Oriented Middleware*. Ed. by Q. H. Mahmoud. John Wiley and Sons, Ltd, 2004. ISBN: 0-470-86206-8 (cit. on p. 21).

[DSA14] P. Desai, A. Sheth, P. Anantharam. *Semantic Gateway as a Service Architecture for IoT Interoperability*. Tech. rep. Ohio Center of Excellence in Knowledge-enabledComputing (Kno.e.sis), Wright State University, Oct. 2014. DOI: 10.1109/MobServ.2015.51. URL: https://www.researchgate.net/publication/267157426_Semantic_Gateway_as_a_Service_Architecture_for_IoT_Interoperability (cit. on p. 28).

[Ear16] A. R. Earls. *XMPP: IoT protocol winner, or second place to MQTT?* Nov. 2016. URL: http://internetofthingsagenda.techtarget.com/feature/XMPP-IoT-protocol-winner-or-second-place-to-MQTT (cit. on p. 46).

[EFGK03] P. T. Eugster, P. A. Felber, R. Guerraoui, A.-M. Kermarrec. "The Many Faces of Publish/Subscribe." In: *ACM Comput. Surv.* 35.2 (July 2003), pp. 114–131. ISSN: 0360-0300. DOI: 10.1145/857076.857078. URL: http://doi.acm.org/10.1145/857076.857078 (cit. on p. 29).

[FAS+16] K. Fysarakis, I. Askoxylakis, O. Soultatos, I. Papaefstathiou, C. Manifavas, V. Katos. "Which IoT protocol? Comparing standardized approaches over a common M2M application." In: *2016 IEEE Global Communications Conference, GLOBECOM 2016 - Proceedings* (Oct. 2016) (cit. on p. 26).

[Fie00] R. T. Fielding. "Architectural Styles and the Design of Network-based Software Architectures." PhD thesis. University of California, Irvine, 2000. URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf (cit. on pp. 33, 67).

[FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014. DOI: 10.1007/978-3-7091-1568-8 (cit. on pp. 20, 21).

[FM11] I. Fette, A. Melnikov. *The WebSocket Protocol*. Dec. 2011 (cit. on p. 50).

[Foc14] FocusGroup. *M2M service layer: APIs and protocols overview*. Tech. rep. Telecommunication Standardization Sector of the International Telecommunication Union (ITU), Apr. 2014. URL: https://www.itu.int/dms_pub/itu-t/opb/fg/T-FG-M2M-2014-D3.1-PDF-E.pdf (cit. on p. 27).

[FR14] R. T. Fielding, J. F. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. July 2014. URL: https://tools.ietf.org/html/rfc7230 (cit. on p. 32).

[GT02] D. Gourley, B. Totty. *HTTP: The Definitive Guide*. Beijing ; Sebastopol, CA : O'Reilly, Sept. 2002. ISBN: 1565925092 (cit. on p. 33).

[Hei00] D. Heimbigner. *Adapting Publish/Subscribe Middleware to Achieve Gnutella-like Functionality*. Tech. rep. Department of Computer Science, University of Colorado, Boulder, Dec. 2000 (cit. on p. 29).

[Hil17] G. C. Hillar. *MQTT Essentials - A Lightweight IoT Protocol*. Packt, Apr. 2017. ISBN: 9781787287815 (cit. on p. 42).

[Hiv16] HiveMQ-Team. *MQTT Essentials*. 2016. URL: https://www.hivemq.com/blog/mqtt-essentials/ (cit. on p. 41).

[Hop03] G. Hophe. *Hub and Spoke or Zen and the Art of Message Broker Maintenance*. Nov. 12, 2003. URL: http://www.enterpriseintegrationpatterns.com/ramblings/03_hubandspoke.html (cit. on p. 21).

[HW04] G. Hohpe, B. Woolf. *Enterprise Integration Patterns*. Addison-Wesley, 2004. ISBN: 0321200683 (cit. on pp. 17, 18, 21, 31, 34).

[KBBL13] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. "Winery – Modeling Tool for TOSCA-based Cloud Applications." In: *11th International Conference on Service-Oriented Computing*. LNCS. Springer, 2013 (cit. on p. 73).

[KCVA15] V. Karagiannis, P. Chatzimisios, F. Vazquez-Gallego, J. Alonso-Zarate. "A Survey on Application Layer Protocols for the Internet of Things." In: *Transaction on IoT and Cloud Computing* 3 (1 2015), pp. 11–17 (cit. on p. 25).

[KG14] R. Kukreja, N. Garg. "Remote Procedure Call: Limitations and Drawbacks." In: *International Journal of Research (IJR)* 1 (10 Nov. 10, 2014). ISSN: 2348-6848 (cit. on p. 34).

[LS18] A. Luoto, K. Systä. "IoT Application Deployment Using Request-Response Pattern with MQTT." In: *Current Trends in Web Engineering*. Ed. by I. Garrigós, M. Wimmer. Cham: Springer International Publishing, 2018, pp. 48–60 (cit. on pp. 28, 29).

[Mas11] M. Masse. *REST API Design Rulebook*. O'Reilly Media, Inc., 2011. ISBN: 1449310508 (cit. on p. 33).

[Mil16] T. Milbourn. *IP versus CoAP for IoT Communications*. July 15, 2016. URL: https://www.u-blox.com/de/ip-versus-coap-iot-communications (cit. on p. 68).

[Mod04] Modbus. *MODBUS Application Protocol Specification V1.1*. June 4, 2004. URL: http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1a.pdf (cit. on pp. 34, 35).

[Mod06] Modbus. *Modbus Messaging Implementation Guide v1*. Oct. 24, 2006. URL: http://www.modbus.org/docs/Modbus_Messaging_Implementation_Guide_V1_0b.pdf (cit. on pp. 34, 35).

[MSB16] S. Mijovic, E. Shehu, C. Buratti. "Comparing application layer protocols for the Internet of Things via experimentation." In: *2016 IEEE 2nd International Forum on Research and Technologies for Society and Industry Leveraging a Better Tomorrow, RTSI 2016* (2016) (cit. on p. 26).

[OAS12] OASIS. *OASIS Advanced Message Queueing Protocol (AMQP) Version 1.0*. Ed. by R. Godfrey, D. Ingham, R. Schloming. Aug. 7, 2012. URL: http://docs.oasis-open.org/amqp/core/v1.0/cos01/amqp-core-complete-v1.0-cos01.pdf (cit. on pp. 38, 39).

[OAS13] OASIS. *Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0*. Jan. 31, 2013. URL: http://docs.oasis-open.org/tosca/tosca-primer/v1.0/cnd01/tosca-primer-v1.0-cnd01.pdf (cit. on p. 11).

[OAS15] OASIS. *MQTT Version 3.1.1*. Ed. by A. Banks, R. Gupta. Dec. 10, 2015. URL: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.pdf (cit. on p. 41).

[Obj13] ObjectManagementGroup. *The Data Distribution Service Specificiations v1.4*. Mar. 2013. URL: http://www.omg.org/spec/DDS/ (cit. on pp. 47, 69).

[OHa07] J. O'Hara. "Toward a Commodity Enterprise Middleware." In: *Queue* 5.4 (May 2007), pp. 48–55. ISSN: 1542-7730. DOI: 10.1145/1255421.1255424. URL: http://doi.acm.org/10.1145/1255421.1255424 (cit. on p. 37).

[PL16] K. Patil, S. Lahudkar. "A Survey of MAC Layer Issues and Protocols for." In: *International Research Journal of Engineering and Technology* 2 (Feb. 2016), pp. 175–186. ISSN: 2395-0072 (cit. on p. 27).

[Pod13] P. Podila. *HTTP: The Protocol Every Web Developer Must Know - Part 1*. Apr. 8, 2013. URL: https://code.tutsplus.com/tutorials/http-the-protocol-every-web-developer-must-know-part-1--net-31177 (cit. on pp. 32, 33).

[RA13] L. Richardson, M. Amundsen. *RESTful Web APIs*. O'Reilly Media, 2013. ISBN: 1449358063 (cit. on p. 33).

[RBF+16] L. Reinfurt, U. Breitenbücher, M. Falkenthal, F. Leymann, A. Riegg. "Internet of Things Patterns." In: *Proceedings of the 21st European Conference on Pattern Languages of Programs (EuroPLoP)*. ACM, 2016. DOI: 10.1145/3011784.3011789 (cit. on p. 16).

[RBN+12] C. Rodríguez-Domínguez, K. Benghazi, M. Noguera, J. L. Garrido, M. L. Rodríguez, T. Ruiz-López. "A Communication Model to Integrate the Request-Response and the Publish-Subscribe Paradigms into Ubiquitous Systems." In: *Sensors* 12.6 (2012), pp. 7648–7668. ISSN: 1424-8220. URL: http://www.mdpi.com/1424-8220/12/6/7648 (cit. on p. 31).

[Ric11] M. Richards. "Understanding the Differences Between AMQP and JMS." In: *NFJS Magazine* 3 (3 May 2011). URL: http://www.wmrichards.com/amqp.pdf (cit. on pp. 26, 39, 40, 42, 43).

[RS16] A. Rayes, S. Salam. *Internet of things-from hype to reality: The road to digitization*. 2016. Chap. 2, pp. 35–56. ISBN: 9783319448602. DOI: 10.1007/978-3-319-44860-2 (cit. on p. 26).

[SBKL17] K. Saatkamp, U. Breitenbücher, O. Kopp, F. Leymann. "Topology Splitting and Matching for Multi-Cloud Deployments." In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*. SciTePress, Apr. 2017, pp. 247–258. ISBN: 978-989-758-243-1 (cit. on pp. 11, 12, 84).

[Sch13a] S. Schneider. *Understanding The Protocols Behind The Internet Of Things*. Oct. 9, 2013. URL: http://www.electronicdesign.com/iot/understanding-protocols-behind-internet-things (cit. on pp. 26, 37, 41).

[Sch13b] S. Schneider. *What's The Difference Between DDS And AMQP?* Apr. 15, 2013. URL: http://www.electronicdesign.com/embedded/what-s-difference-between-dds-and-amqp (cit. on pp. 69, 70).

[SHB14] Z. Shelby, K. Hartke, C. Bormann. *The Constrained Application Protocol (CoAP)*. July 2014. URL: https://tools.ietf.org/html/rfc7252 (cit. on pp. 49, 67).

[SJ16] T. Salman, R. Jain. "Networking Protocols and Standards for Internet of Things." In: *Internet of Things and Data Analytics Handbook* (2016), pp. 215–238. DOI: 10.1002/9781119173601.ch13. URL: http://doi.wiley.com/10.1002/9781119173601.ch13 (cit. on p. 27).

[Sol17] Solace. *Understanding IoT Protocols – Matching your Requirements to the Right Option*. Jan. 25, 2017. URL: https://solace.com/blog/use-cases/understanding-iot-protocols-matching-requirements-right-option (cit. on pp. 16, 37, 41).

[SS17] P. Sethi, S. R. Sarangi. "Internet of Things: Architectures, Protocols, and Applications." In: *Journal of Electrical and Computer Engineering* 2017 (2017). ISSN: 2090-0147 (cit. on p. 26).

[SSCK17] S. B. Sasirekha, S. Swamynathan, S. Chandini, K. Keerthana. "Analysis of Application Layer Protocols in Internet of Things." In: *Advances in Computing and Data Sciences* 721 (2017), pp. 550–561. URL: http://link.springer.com/10.1007/978-981-10-5427-3 (cit. on p. 26).

[SST09] P. Saint-Andre, K. Smith, R. Troncon. *XMPP: The Definitive Guide*. Ed. by M. E. Treseler. O'Reilly Media, Inc, Apr. 2009. Chap. 1 and 2. ISBN: 978-0-596-52126-4 (cit. on pp. 43–46).

[STO12] STOMP. *STOMP Protocol Specification, Version 1.2*. Oct. 22, 2012. URL: https://stomp.github.io/stomp-specification-1.2.html (cit. on pp. 42, 43).

[Tho01] S. Thomas. *HTTP Essentials: Protocols for Secure, Scalable Web Sites*. New York : Wiley, 2001. ISBN: 978-0471398233 (cit. on pp. 32, 33).

[TK17] P. Thota, Y. Kim. "Implementation and Comparison of M2M Protocols for Internet of Things." In: *Proceedings - 4th International Conference on Applied Computing and Information Technology, 3rd International Conference on Computational Science/Intelligence and Applied Informatics, 1st International Conference on Big Data, Cloud Computing, Data Science and Engineering, ACIT-CSII-BCD 2016* (2017), pp. 43–48 (cit. on pp. 25, 26).

[Twi17] TwinOaksComputing. *What can DDS do for You?* July 24, 2017. URL: http://www.omg.org/hot-topics/documents/dds/CoreDX_DDS_Why_Use_DDS.pdf (cit. on p. 47).

[Vas17] C. Vasters. *From MQTT to AMQP and back*. May 13, 2017. URL: http://vasters.com/blog/From-MQTT-to-AMQP-and-back/ (cit. on pp. 28, 37, 56–58).

[Vin06] S. Vinoski. "Advanced Message Queuing Protocol." In: *IEEE Internet Computing* 10.6 (Nov. 2006), pp. 87–89. ISSN: 1089-7801 (cit. on p. 37).

[Wah16] P. Waher. *Choose the Right Communication Pattern for Your IoT Project*. June 1, 2016. URL: https://software.intel.com/en-us/articles/communication-patterns-for-the-internet-of-things (cit. on pp. 18–20, 31).

[WSE14] M. Weyrich, J. P. Schmidt, C. Ebert. "Machine-to-machine communication." In: *IEEE Software* 31.4 (2014), pp. 19–23. ISSN: 07407459. DOI: 10.1109/MS.2014.87 (cit. on p. 27).

[WSM13] V. Wang, F. Salim, P. Moskovits. *The Definitive Guide to HTML5 WebSocket*. APress, 2013. ISBN: 978-1-4302-4741-8 (cit. on p. 50).

[YSA16] M. B. Yassein, M. Q. Shatnawi, D. Al-Zoubi. "Application layer protocols for the Internet of Things: A survey." In: *Proceedings - 2016 International Conference on Engineering and MIS, ICEMIS 2016* (2016) (cit. on p. 25).

All links were last followed on March 24, 2018.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature