Schweiger Manuel, BSc

Connecting IoT Devices to the Cloud using Message Queue Protocols

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisors

Dipl.-Ing. Dr.techn. BSc Georg Macher
Dipl.-Ing. BSc Michael Krisper

Institute of Technical Informatics
Head: Univ.-Prof. Dipl.-Infrom. Dr.sc.ETH Kay Uwe Römer

Graz, August 2020

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date	Signature

Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während dieser Masterarbeit und meiner Studienjahre unterstützt und motiviert haben.

Zuerst gebührt mein Dank den Herren Dipl.-Ing. Michael Krisper und Dr. Georg Macher vom Institut für Technische Informatik an der TU Graz, die meine Masterarbeit betreut und begutachtet haben. Trotz der speziellen Umstände in dieser ungewöhnlichen Zeit wurden immer Wege gefunden für eine gute Zusammenarbeit. Für die hilfreichen Anregungen und die konstruktive Kritik bei der Erstellung dieser Arbeit möchte ich mich herzlich bedanken.

Des Weiteren gilt mein Dank meinen beiden Arbeitgebern, Volker Knapp und Michael Mally, die mich täglich fördern sowie motivieren und mir diese Masterarbeit in ihrem Unternehmen primtec ermöglichten.

Ebenfalls bedanke ich mich bei meinen Eltern, Heidi und Fredi, die mich während meines Studiums stets unterstützt und immer an mich geglaubt haben.

Nicht zuletzt gebührt mein Dank Alexandra, die immer für mich da ist und auf deren Unterstützung ein großer Teil meines erfolgreichen Studiums beruht.

Abstract

In the course of this work, the foundation for the development and implementation of an *IoT/Cloud* system is to be laid, with the aim of equipping each product of the Styrian IT company *primtec GmbH* with *IoT* functionalities to be able to carry out maintenance and support of these remotely. At present, it is not possible to carry out remote maintenance, which is reflected in enormous time and cost expenditures for the company. The industrial environment in which primtec's customers are located depicts a further challenge due to its restrictions imposed by companies' internal network policies and firewalls.

The first part of this thesis focuses on the concepts *Cloud* and *IoT* by giving a deep insight into their history, technology, and influence on science. During the second part, challenges and problems concerning primtec and its customers are listed and evaluated, from which the implementation requirements were then derived. In the final part, a solution gets developed and implemented by introducing a *CloudIoT* extension for the existing products. This extension includes an intern *IoT* machine-to-machine communication framework by using a RabbitMQ message broker and a so-called *orchestrator* which handles the data exchange with the *Cloud*. In addition, this thesis also provides a general comparison between the three most comprehensive cloud providers and an analysis of the essential messaging protocols used in connection with *IoT*.

Since the developed framework is a basic concept, additional improvements and extensions are proposed to bring the final product to a sufficient production level in terms of security and usability.

Kurzfassung

Im Zuge dieser Arbeit sollen Grundsteine für die Entwicklung und Umsetzung eines *IoT*/*Cloud*-Systems gelegt werden, mit dem Ziel jedes Produkt des steirischen IT-Unternehmens *primtec GmbH* mit *IoT*-Funktionalitäten auszustatten um Wartung und Support dieser aus der Ferne durchführen zu können. Momentan ist eine Fernwartung nicht möglich, woduch hohe Zeit- und Kostenaufwände für das Unternehmen entstehen. Das industrielle Umfeld der Kunden *primtecs* stellt durch firmeninterne Netzwerkrichtlinien und Firewalls eine weitere Herausforderung dar.

Der erste Teil dieser Arbeit fokussiert sich auf die Konzepte *Cloud* und *IoT*, indem ein detaillierter Einblick in deren Geschichte, Technologie und Einfluss auf die Wissenschaft gegeben wird. Im zweiten Teil werden die Herausforderungen und Probleme, die *primtec* und dessen Kunden betreffen, analysiert und bewertet, woraus dann wiederum die Anforderungen an die Implementierung abgeleitet werden. Im finalen Teil wird eine Lösung entwickelt und implementiert, indem eine *Cloud*-Erweiterung für die bestehenden Produkte eingeführt wird. Diese Erweiterung umfasst ein internes *IoT* Machine-to-Machine-Kommunikations-Framework unter Verwendung eines RabbitMQ Message Brokers und eines sogenannten *Orchestrators*, der den Datenaustausch mit der *Cloud* abwickelt. Darüber hinaus wird in dieser Arbeit ein allgemeiner Vergleich zwischen den drei größten *Cloud*-Anbietern sowie eine Analyse der wichtigsten Messaging-Protokolle, die im Zusammenhang mit *IoT* verwendet werden, erstellt.

Da es sich bei dem entwickelten Framework um ein Basiskonzept handelt, werden zusätzliche Verbesserungen und Erweiterungen vorgeschlagen, um das Endprodukt in Bezug auf Sicherheit und Benutzerfreundlichkeit auf ein ausreichendes Produktionsniveau zu bringen.

Contents

List of Figures

List of Tables

1 Introduction

Internet of Things (*IoT*) and *Cloud* are two highly innovative concepts which are used in many areas today. Global players such as Google, Microsoft, and Amazon derive the most of their profits from their cloud platforms, which are being used by more and more companies due to their high scalability, low costs, and simplicity. The Styrian IT company *primtec GmbH* also wants to take a step towards advanced cloud technology. The basis for this is a communication and control platform which will be researched, analyzed, and implemented in this master thesis.

primtec is a company that focuses on the automatic identification of items using the RFID technology as well as providing custom warehouse management solutions. RFID (*radio-frequency identification*) describes the recognition of so-called RFID-tags, which are usually small stickers bond to the objects to be identified. By emitting an electromagnetic field using a dedicated RFID-reader device, these tags get triggered and send digital data, usually a unique code used for identification.

These RFID solutions by primtec are spread worldwide, but remote monitoring them is currently not possible. In case of an error, it can take hours until the failure is recognized by an employee at the customer's premises and several additional hours until according error reports are forwarded to primtec.

This work aims to develop and implement the base of an "Internet of Things" framework combined with cloud functionalities suited for an industrial environment. This framework should make it possible to monitor these devices, detect errors early, and provide additional features such as remote updates. It should be easy to utilize in existing applications while also providing the necessary functionalities to establish a connection with a cloud platform automatically.

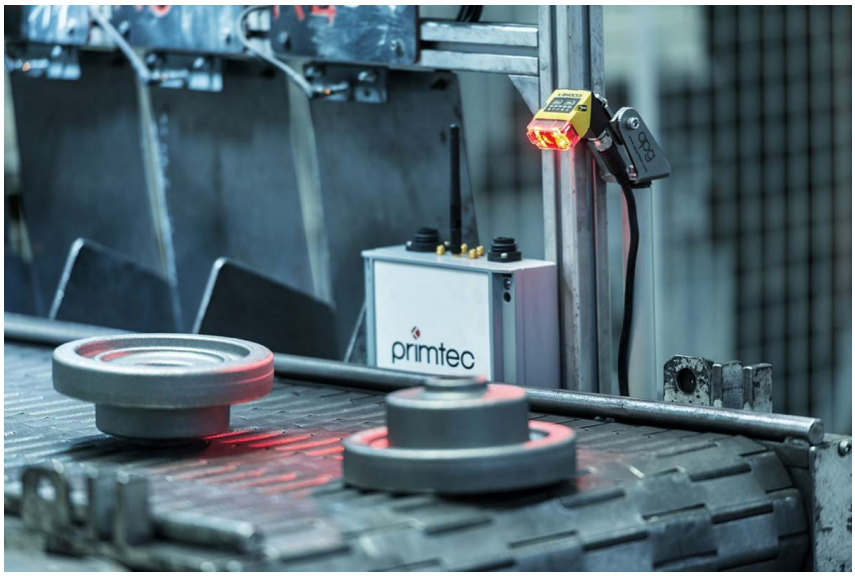

Figure 1.1: primtec GmbH RFID technology on a production line, *taken from primtec-IoT (2020)*

1.1 Structure

This chapter provides a brief introduction to primtec's products, IoT, and its architecture, as well as the definitions of cloud services and platforms.

Chapter 2 introduces and compares the three cloud provider Google Cloud Platform, Amazon Web Service, and Microsoft Azure. So-called *message brokers* are described in the concluding section.

Scientific related work to the topics *Internet of Things*, *Cloud Computing*, and *Message Brokers and Protocols* can be found in chapter 3.

The current situation of primtec and the constraints imposed by the industrial sector are explained in chapter 4. From this, requirements for the implementation of this software are defined and broken down in detail.

Chapter 5 describes the implemented solution by focusing on each module, which are the messaging broker, the client software, the orchestrator, the cloud, and the web interface, separately.

Chapter 6 provides an assessment of how each requirement is met with the current implementation.

Limitations, improvements, and future work are described in chapter 7.

Finally, in chapter 8, the whole thesis is briefly summarized once more, and a conclusion is made.

1.2 Introduction to primtec

primtec GmbH is a company located in Graz (Austria), which creates solutions for digitization in various industries and business areas. primtec's solutions include RFID picking stations, inventory systems, warehouse management, and logistics technologies, all of which are supported and automated by RFID technology. The customers are companies from industry, logistics, medical technology, and public institutions, whereby in all areas, the simple and error-free handling of the software plays an important role.

1.3 Products by primtec

This section shows two basic versions of primtec products currently used by customers worldwide: the *Packing Table* and the *Collector*. They do not have any decentralized data acquisition and are therefore the first products to be equipped with the cloud systems presented in this thesis. Both products have been developed for a Windows 10 environment and show information on a connected display. These devices can be connected to the in-house network either by LAN or WIFI but do not need access to the Internet.

The Packing Table

The *Packing Table* is one of primtec's products for use in warehouses and helps to pack products that have to be sent to customers correctly. It consists of an ergonomic table, a computing unit with a monitor, an RFID reader

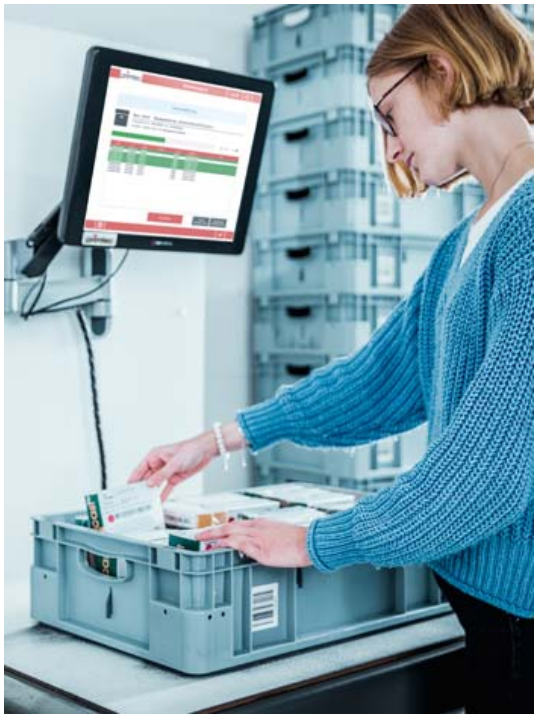

Figure 1.2: The primtec GmbH Packing Table, *taken from primtec-PackingTable (2020)*

mounted under the table surface, and an optional bar code scanner. It is usually connected to a data source (e.g., SAP) and displays all the items belonging to a particular order. While placing these articles in a box standing on the table, the RFID labels are scanned and compared with the nominal value. The display shows which items are missing or incorrectly placed, and if each correct item is present, the user can complete the order by pressing a button on the display.

Since the packing table uses the RFID technology, a prerequisite is that all items are tagged with an RFID tag. However, in case of a broken RFID label, the bar code scanner can be used as a fallback.

Each packing table uses a local database (SQLite) for logging. The RFID tags read by the scanner get decoded and written into a database, as well as connection changes to the reader, network, or other peripheral devices. Another essential piece of information is the exact listing of each item of the completed order. If an error appears, it is crucial to know which items are shipped in which box.

Since the packing table can process hundreds of tags simultaneously, many log entries can appear simultaneously. It is of high importance that no log entries are lost, as this would distort the traceability of the actions performed by the user. Missing tags or inaccurately recognized tags can lead to incorrectly shipped products, which can have fatal consequences, especially in the medical field. The timing of the logs plays a minor role, so real-time data transfer is not necessary.

The packing table does not feature an update functionality. Therefore, if a software update is required, a primtec technician must manually update the program, either locally or remotely, using a screen-sharing software like *Teamviewer*[1].

The Data Collector

The *Data Collector* is a software module by primtec used in several custom made solutions in many different scenarios like production lines or supply

[1] https://www.teamviewer.com/en/

chains. It consists of two RFID readers, a computing unit, as well as a display. It covers two functionalities: adding a tag entry to a list shown on the display when an RFID tag gets read by the scanner and removing the entry from the list if the RFID tag is departed. These entries are written to a local database.

These two basic functionalities are an essential requirement for many products and projects by primtec. An example would be a production line where each item must be tracked as it passes through a particular station or machine. Since it is often the case that several items pass the reader within one second, fast evaluation and reporting are crucial to ensure a correct and error-free process.

Similar to the packing table, there is no decentralized logging or update mechanism, and real-time logging is not relevant, whereas the completeness and integrity of the data are crucial.

1.4 The Internet of Things

The Internet is a dynamic technology still undergoing constant changes and expansions. It started as a platform where people can connect and communicate with each other and was hence mostly a human-to-human (H2H) system. The term *Internet of Things* began to become popular in the late 90s and described an autonomous network of smart devices based on RFID identification. As stated by K. Chopra, K. Gupta, and A. Lambora (2019): "The next upcoming form of communication that uses Internet as the underlying technology is The Internet of Things (IoT). IoT extends the capabilities of Internet to enable machine-to-machine communication (M2M)." This means that everyone and everything can communicate with each other. This change opens a whole new world of possibilities since data can be sent automatically by the smallest devices, in terms of processing power, to large computer clusters, which process and analyze this data and thus invoke specific actions based on this data that could otherwise not be processed on small devices.

As described in Dorsemaine et al. (2015), billions of objects will soon be connected and communicating with each other without any need for human interference. Smart objects will take over control of daily routines and communicate with each other. An illustrative example of this would be a temperature sensor that measures the current temperature inside a greenhouse and sends the data to a cloud platform. There, an unusually low temperature can be detected based on past temperature changes and with the help of machine learning. The cloud calculates the required power of the heaters and sends the ”heat-command” to the corresponding heaters inside the greenhouse. These actions only take a few seconds, on a controller, inside a mini amateur greenhouse for under 100$.

We live in a world where everything is becoming smarter, more intelligent, and more connected. The consumer market is already adapting accordingly, and huge profits are made by selling smart devices such as *Amazon Alexa*, smart light bulbs, or similar devices. Other examples are intelligent houses equipped with smart light bulbs, door locks and loudspeakers, smart cities with intelligent traffic lights, and smart cars with car-to-car communication and intelligent car-to-infrastructure communication.

An Architecture for the Internet of Things

Numerous technologies are associated with IoT, such as wireless sensors, networks, barcodes, RFID, NFC, low energy wireless communications (LoRaWAN), Bluetooth Low Energy, Zigbee, Narrowband IoT, and cloud computing (Li, Xu, and Zhao (2014)). Even though IoT and all related technologies differ significantly, the base architecture is identical. Dorsemaine et al. (2015) described the underlying architecture based on four levels:

1. *Local Environment*: These are the actual IoT devices connect through a wired (including Ethernet and optic fiber) or wireless technology (including Bluetooth and WIFI) within their environment. Local pickup points (optional) can gather the data from weak devices, in terms of battery or computing power. They can act as a gateway to the other layers of architecture.
2. *Transport*: This layer is used for communication between the IoT devices (local environment) and the command servers (e.g., cloud).

3. *Storage and data mining*: The storage and data mining levels act as the "brain" in an IoT network. It usually takes place inside the cloud (described in 1.5), which provides a massive amount of processing power and storage capacity.
4. *Availability (GUI, API)*: The final level is used to access the data either in the form of a graphic user interface (GUI) or an application programming interface (API). An example of a GUI would be a website that shows a dashboard containing charts and graphs of the data. An API, on the other hand, is used to make the data accessible for other programs that can further process it.

In this thesis, we use the following definition for IoT by Dorsemaine et al. (2015): "Group of infrastructures interconnecting connected objects and allowing their management, data mining and the access to the data they generate".

1.5 Cloud Computing

The definition and taxonomy of the term *cloud* has and is steadily changing over the last years. Fatemi Moghaddam et al. (2015) define cloud computing as: "Cloud computing is a technology that uses the concepts of virtualization, processing power, storage, connectivity, and sharing to provide a pool of resources, store and share them between various devices via a broad network (i.e., Internet) to offer on-demand services to end-users in compliance with the concepts of isolation, security, distribution, and elasticity".

Due to the flexibility, the low costs, and the extensibility, cloud computing is one of the most prominent keywords in the IT sector and industry (Panetta (2020)). Today there are three big global players dedicated to cloud computing and providing public cloud functionality: *Amazon Web Services (AWS)* [2]., *Microsoft Azure* [3], and *Google Cloud* [4], presented in 2.1. *Alibaba*

[2] https://aws.amazon.com/
[3] https://azure.microsoft.com/
[4] https://cloud.google.com/

Cloud [5] would be another big global player that has gained much popularity, especially in recent years, but is out of scope for this thesis.

1.5.1 Cloud Service Models

According to the *National Institute of Standards and Technology* (Mell and Grance (2011)), also called NIST, three different kinds of cloud service models are defined:

IaaS - Infrastructure-as-a-Service

"The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls)." (Mell and Grance (2011))

IaaS describes the virtualization of hardware, servers, operating systems, and the physical network. Developers can define and configure these components and, within certain limits, use it afterward as if it were their own. Examples would be the *Google Compute Engine* [6], which is part of the Google Cloud Platform, or most products of Amazon Web Services and Microsoft Azure. IaaS is the most flexible of all cloud service models - the client can completely control the infrastructure, resources can be purchased on-demand, and the computational power is scalable based if needed.

PaaS - Platform-as-a-Service

"The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages,

[5] https://www.alibabacloud.com/
[6] https://cloud.google.com/compute

libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment." (Mell and Grance (2011))

PaaS virtualizes the software back-ends, including run-time environments, virtual machines, and libraries. An example would be *AWS Elastic Beanstalk* [7], which can be used to deploy, manage, and scale web applications implemented in, for example, ASP.NET. The advantage is the flexibility regarding the resources, which can be scaled dynamically, the accessibility for developers, and the simplification of deployment workflows.

SaaS - Software-as-a-Service

"The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g. web-based email), or a program interface. The consumer does not manage or control the underlying cloud infrastructure, including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings." (Mell and Grance (2011))

The virtualization of specific applications and front-ends is called SaaS. The advantage of SaaS is that applications can be hosted on cloud servers and be accessed from all around the world without the need for client-side installation, maintenance, and updating. Usually, a standard web browser, such as Google Chrome or Mozilla Firefox, is sufficient to use these online apps. Dropbox, Cisco WebEx, and WhatsApp Web are commonly used examples (some of these SaaS applications provide downloadable software in addition, which is not considered as SaaS).

[7] https://aws.amazon.com/elasticbeanstalk/

1.5.2 Cloud Deployment Models

In addition to the described cloud service models, the NIST also defined four deployment models for clouds (Mell and Grance (2011)):

- *Private Cloud*: It describes a cloud hosted for exclusive use by a single organization. It is owned and managed by the organization or a third party.
- *Community Cloud*: This service model stands for a cloud hosted for exclusive use by a specific community of consumers/organizations with shared concerns. It may be owned by one or more of the organizations or a third party.
- *Public Cloud*: Public cloud, on the other hand, describes a model where the cloud is hosted by a company, academic or governmental organization and is used by the general public.
- *Hybrid Cloud*: A hybrid cloud is a combination of the two or more models described above.

1.6 Messaging Protocols

IoT devices and cloud servers need to communicate with each other. Therefore languages with different grammar and guidelines, standardized shapes, and certain flexibility are required. These languages must be recognizable and processable by all devices - the World-Wide-Web, E-Mails, or any kind of data exchange, would otherwise not be possible. The Internet today uses a plethora of protocols, each with specific purposes for different layers of abstraction (ISO/OSI stack), such as DNS for resolution of names to IP addresses, TCP for reliable communication of packets, FTP for file transfers, and HTTP as transfer-protocol. Although they are made for different purposes and have different characteristics, they are based on the internet protocol stack, as shown in Figure 1.3.

Assuming we have a cloud server, also known as the data requestor, and multiple IoT devices, the data providers, the only thing missing is the middleware, including the message protocol best suited for this purpose. It should handle the connection between data requestor and data providers,

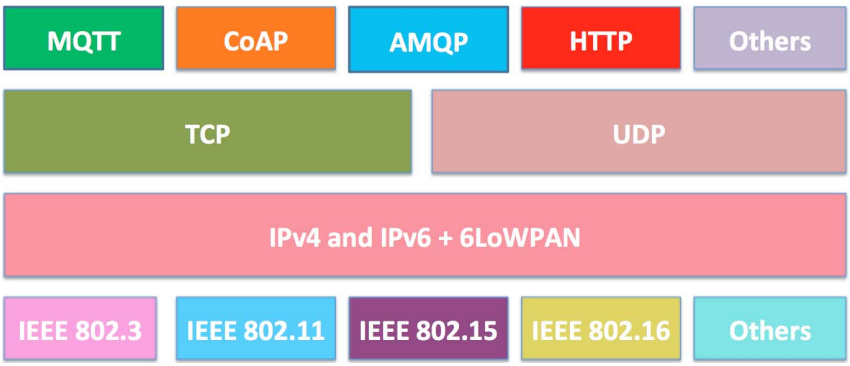

Figure 1.3: IP stack with IoT Technologies, *taken from Naik (2017)*

while also validating each participant. Over the years, different protocols of this kind have emerged, all associated with different advantages and disadvantages and suited for specific requirements.

As claimed by Naik (2017), choosing the right messaging protocol can be a challenging and daunting task for organizations, and therefore four of the most popular and widely accepted protocols in IoT are described and compared in this section: *MQTT*, *AMQP*, *HTTP*, *WebSockets*, and *WebRTC*.

1.6.1 MQTT - Message Queuing Telemetry Transport Protocol

MQTT was introduced in 1999 and is designed in a very simple and light-weight way. It is often used in large networks with weak computational devices. MQTT is a publish/subscribe protocol, which means that a client, for example, a smart temperature sensor, publishes its data to a specific channel, while another client can subscribe to this channel and thus receives all data. A so-called message broker takes care of all channels and ensures that all subscribed clients receive the corresponding data. Additionally, the broker can save the data for future clients, which means that all data from the past is transmitted if a client subscribes to a channel. The basic structure of an MQTT data transmission system is shown in Figure 1.4

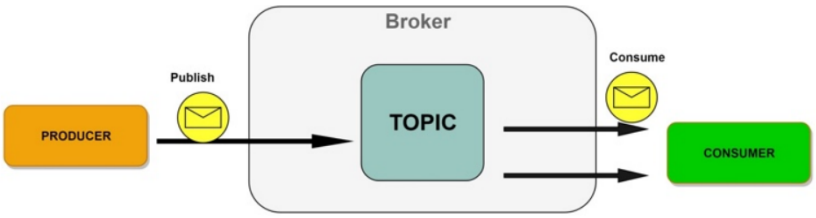

Figure 1.4: Structure of the data transmission system using MQTT, *taken from Uy and Nam (2019)*

The payload of an MQTT message can take up to 256 megabytes, which is usually sufficient to transmit logs and sensor data. The data itself is sent in plain text, which means that MQTT does not feature any kind of encryption, but due to the fact that messages are sent over TCP, TLS/SSL can take care of the security and integrity.

MQTT features three levels of ”Quality of Service” (QoS) which can be chosen by the client:

- *At most once*: Messages sent with this QoS follow the ”fire and forget” principle, which means that each message is sent without a guarantee of delivery. The message broker does not store them, and no data is re-transmitted.
- *At least once*: This level guarantees that messages are received by the subscriber. The publisher stores and resends the message until a so-called ”PUBACK” (publish acknowledgement) package is received.
- *Exactly once*: ”Exactly once” is the last and most time-consuming QoS. A ”handshake” is performed to guarantee that the message is received once by the subscriber.

A workflow of the ”At most once” QoS can be seen in Figure 1.5. Client-1 connects to the MQTT network while Client-2, which is already connected, publishes a message to the ”Log” channel. As soon as Client-1 subscribes to ”Log”, it receives the message from Client-2, which in the meantime, has been stored at the broker.

The whole documentation and specification of MQTT can be found at the ”OASIS Open” homepage (OASIS-Open (2020))

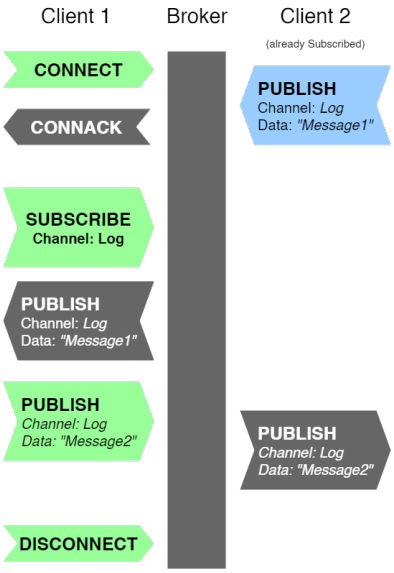

Figure 1.5: Workflow of an MQTT connection with the least Quality of Service, *redrawn from Wikipedia (2020)*

1.6.2 AMQP - Advanced Message Queuing Protocol

AMQP, developed in 2003, is another publish/subscribe based protocol using a message broker as middleware, but in contrast to MQTT, features additionally a request/response system as well as a wide range of features such as reliable queuing, topic-based publish/subscribe messaging, flexible routing and transactions (Naik (2017)).

The main difference to MQTT is that messages are not routed directly to the consumer but instead, are sent to a so-called ”exchange” (Uy and Nam (2019)). These exchanges can be compared to post offices that distribute copies of the messages to queues, according to predefined rules called ”bindings”. Clients can subscribe to these queues and therefore receive the messages automatically or fetch them on demand. The queue/exchange data transmission system can be seen in Figure 1.6. Four different routing algorithms, which can be used by the exchanges, are supported.

Direct Exchange

"Direct Exchange" can be used both for "unicast", which means a single message recipient, and for "multicast", where several message receivers are desired. This algorithm utilizes a "routing key" k used by the queue to bind on a particular exchange. If a message enters the exchange with a routing key r, the exchange routes the message to the queue if a rule $k = r$ was defined (RabbitMQ (2020)).

Fanout exchange

The "Fanout Exchange" can be compared to the MQTT topic subscription. The routing key, if defined, is ignored, and incoming messages are distributed as copies to all bound queues (RabbitMQ (2020)).

Topic Exchange

If messages can have specific types, the "Topic Exchange" should be used. If the message's routing key matches a pattern defined during the binding of the queue to the exchange, the messages get delivered (RabbitMQ (2020)).

Headers Exchange

When the exchange uses the "Headers Exchange" algorithm, the routing key is ignored; instead, specific attributes are taken from the message header. The message gets distributed from the exchange to the queue if the header's value matches a value specified in the binding (RabbitMQ (2020)).

Similar to MQTT, AMQP also builds upon TCP as transport-protocol with TLS/SSL for security, and it also features the same QoS levels as described in 1.6.1. While sending messages, publishers can attach meta-data, the so-called "message attributes", which is, with a few exceptions, not visible to the broker and can be used by the receiving applications.

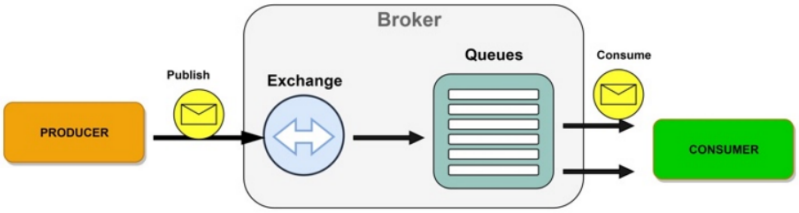

Figure 1.6: Structure of the data transmission system using AMQP, *taken from Uy and Nam (2019)*

1.6.3 HTTP - Hyper Text Transport Protocol

HTTP, developed by Tim Berners-Lee at the Cern in 1989 and released in 1997, is an application protocol to communicate between servers and clients. It functions as a request-response protocol, which means that the client requests data, and the server responds to it. In contrast to AMQP and MQTT, HTTP does not use topics to which messages are related to, but instead, it uses an "Universal Resource Identifier" (URI) which consists out of:

- *Scheme*: The scheme specifies the context behind the URI. The software which is used to access the URI knows in consequence how to interpret the requested data. Common schemes are *HTTP*, *HTTPS*, *FTP*, *file* and *mailto*.
- *Authority*: The authority contains an optional and according to RFC 3986 obsolete *userinfo* and the *host*. The host can either be a DNS registered name or an IP address, followed by an optional *port*.
- *Path*: The path is a sequence of segments separated by a slash and usually relates to the file path on the corresponding server. It ends either at the end of the URI, at a hash symbol, or a question mark.
- *Query*: If the path alone is not sufficient to identify the requested data, an optional query can be used introduced by a question mark. Queries are often used if data is saved to a database.
- *Fragment*: Fragments, identifiable by the prefix "#", can be used to refer to specific segments within the requested resource. An example would be the URI to a particular section on a website.

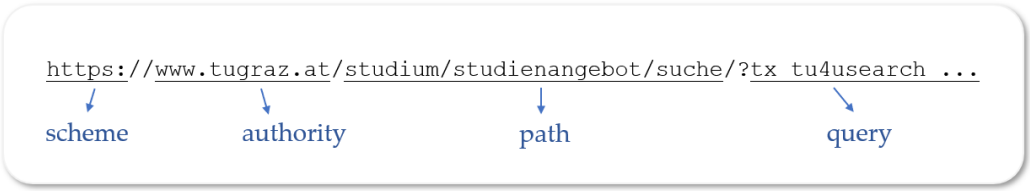

Figure 1.7: A *Uniform Resource Identifier* example

Real-time data transfer is not possible with HTTP since the client always has to perform a data request. This means that, if a website shows, for example, the live state of a soccer game, the client must request the intermediate state permanently in a defined time interval. This behavior is described as *polling* and can be considered as a so-called *half-duplex* mode. HTTP also uses TCP as transport protocol and can be secured via TLS/SSL but does not feature any kind of QoS.

Nowadays, HTTP is the standard protocol of the world-wide-web - the Internet, as we know it, would not be possible without this protocol. However, in IoT, HTTP plays only a minor role because HTTP opens and closes a connection on each data request due to its *half-duplex* design. Additionally, HTTP requires a specific URI, which may not be available if the communication partner is unknown. Nevertheless, it is often used for authentication and authorization mechanisms, such as in Guner, Kurtel, and Celikkan (2017), where an IoT system based on MQTT communication was developed, and the administrative tasks were performed using HTTP.

1.6.4 WebSocket

WebSocket, standardized in 2011, is a protocol providing *full-duplex* communication. *Full-duplex* describes simultaneous communication in both directions, which is not possible using standard HTTP. With HTTP version 1.1, *pipelining* was released, giving the client the opportunity to send multiple requests without waiting for a response. Controversially, this is also sometimes referred to as *full-duplex*).

In contrast to HTTP, real-time data transfer is possible as soon as the client and the server have an open WebSocket connection. This means that the server can send data to the client without a previous data request.

Like HTTP, WebSocket uses TCP as transport-protocol, and the connection can also be encrypted using TLS/SSL.

1.6.5 WebRTC

WebRTC[8] is an open-source project developed for real-time communication in web browsers and mobile applications. Intending to eliminate the co-occurrence of communication browser plugins and applications, the *World Wide Web Consortium (W3C)* and the *Internet Engineering Task Force (IETF)* released the first version in 2011 (Edan, Al-Sherbaz, and Turner (2017)). Today it is supported by all common web browsers, including Google Chrome, Mozilla Firefox, and Microsoft Edge.

The peer-to-peer communication relies on a UDP connection encrypted with *DTLS*, which is a derivative of *SSL*.

[8] https://webrtc.org/

2 Background

This chapter provides a quick overview of cloud-providers and message brokers. Considering the fast-growing field of cloud computing and providers, the results of this master thesis represent a snapshot of 2020.

2.1 The three big Cloud Provider

2.1.1 AWS - Amazon Web Service

Amazon Web Services (AWS) was founded in 2006 and has grown to become one of the largest cloud providers. According to MarketWatch (2020), AWS had a dominant market share of 51% in 2017. In recent years, this number has been shrinking as other large companies entered the cloud market. Currently, according to ParkMyCloud (2020), Amazon Web Services holds a market share of 32.4%, which is about one-third of the total cloud provider market and is therefore still in the lead.

Today, an enormous amount of companies use AWS, including Netflix, Facebook, BBC, and LinkedIn. AWS offers a wide range of services and provides daily updates and expansions. Theses services include databases, storage, Internet of Things frameworks, machine learning, block-chain frameworks, developer tools, and many more. Payment is usually on a pay-as-you-go basis, which means that there is no fixed subscription cost, but a fee based on the usage (e.g., messages per second), the hardware architecture chosen by the user, and various additional services such as security and redundancy features. Primarly due to the large number of different options, services, and frameworks, pricing can become intransparent and challenging to calculate in complex applications.

2.1.2 Microsoft Azure

With a market share of about 17.6% (ParkMyCloud (2020)), Microsoft Azure is the second major player in cloud computing. It was publicly released in 2010 and has become a suitable counterparty for AWS. At first glance, the number of features is similar to AWS, but it is not as advanced in detail. However, a significant advantage of Microsoft Azure is the integration of .Net developers. Several development packages are available for .Net applications, making it very easy to connect the application to the cloud. The pricing for Microsoft Azure is similar in structure compared to AWS.

2.1.3 GCP - Google Cloud Platform

Google Cloud Platform, publicly released in 2012, is the newcomer among the cloud providers. A market share at about 6% (ParkMyCloud (2020)) in 2020 is relatively low compared to AWS and Azure, but it is on the rise. Enormous investments in data centers worldwide have been made to provide excellent connectivity, availability, and latency in all regions. The pricing is different from AWS since the user can choose a suitable hardware architecture, and costs are calculated by specifying the server uptime.

2.2 Message Brokers

A message broker is a software module that enables applications, machines, and services, located in a network, to communicate. The communication is based on messages sent from a source to a destination, which can also be called a message-oriented middleware. The message broker handles the delivery of the messages to the destinations while also providing additional features, for example, the validation of the sent messages or storing them (IBM-Cloud-Education (2020)).

Another technology, often used by the message broker itself, is called *message queues*. These message queues act as a temporary message store waiting for a message destination to receive those messages actively.

Message brokers are distinguishable by the two modes in which they can work:

1. *Point-to-point messaging*: Point-to-point messaging describes a communication pattern where messages are sent from one sender to exactly one message receiver. This can be compared to postal traffic, where the sender of a letter has to provide the desired recipient's address to send this exact message to exactly this recipient. Systems with this behavior are often called *one-to-one* systems.
2. *Publish/Subscribe messaging*: Using the Publish/Subscribe pattern, in contrast to point-to-point messaging, multiple message receivers can be addressed. This is done by publishing the message to be sent to a particular topic. This message is then forwarded to all those who have subscribed to this topic. Unlike point-to-point messaging, it can not be compared to mail traffic, but rather to a magazine subscription where each subscriber gets a copy of the original message. This broadcast-style message distribution can, therefore, also be called a *one-to-many* system (Sachs et al. (2010)).

2.2.1 RabbitMQ

RabbitMQ[1] is one of the most popular open-source message brokers. Originally developed for the telecommunication industry, it is nowadays used by many influential companies such as *Runtastic*, *Trivago*, and *Reddit*.

The RabbitMQ server program, implemented in Erlang, was designed to work with AMQP but has extended to a plugin infrastructure, which makes it possible to use RabbitMQ also with MQTT and STOMP.

In addition to the server software, operable on Windows, Linux, and macOS, RabbitMQ published numerous software libraries that can be used to implement the client software in the most common programming/scripting languages, including C#, Python, Java, and JavaScript.

RabbitMQ provides two security functionalities: SSL/TLS support and client user authentication.

[1] https://www.rabbitmq.com/

2.2.2 Eclipse Mosquitto

Eclipse Mosquitto[2] is an open-source MQTT broker implemented in the language C with the great advantage of being designed in a light-weight way, making it possible to run on nearly every device even though the computational power is low. Additionally, Mosquitto provides C libraries for the implementation of MQTT clients.

Mosquitto can also be used in combination with SSL/TLS; however, according to Babovic, Protic, and Milutinovic (2016), it does have some performance issues in specific scenarios.

2.2.3 HiveMQ

HiveMQ[3] is another MQTT broker designed for business-critical IoT systems and, according to their website (https://www.hivemq.com/), scales up to 10 million connected devices. It can be deployed to a local private cloud, a hybrid cloud, or a public cloud, including AWS and Microsoft Azure.

HiveMQ is not open-source, but they released a light open-source community edition. As stated on their website, the main differences between the *HiveMQ Profession/Enterprise* and *HiveMQ Community Edition* are the features for business-critical applications, enterprise security integration, and monitoring/observability functionalities.

2.2.4 Other Message Brokers

There are countless message brokers supporting different messaging protocols, using various technologies, and being either open-source or closed-source. Some of them gained popularity because of good marketing and search engine optimization, and some disappeared over the years. In this thesis, only three are presented due to the sheer number of brokers, which would go beyond this work scope. However, it must be pointed out that

[2] https://mosquitto.org/
[3] https://www.hivemq.com/

there are many more that could be considered to implement the solution. A rough overview of the most popular message brokers, and a small comparison regarding their features, can be found on https://ultimate-comparisons.github.io/ultimate-message-broker-comparison/. A more scientific approach was made by John and Liu (2017) who surveyed the message brokers that are "in vogue" today.

3 Related Work

This chapter focuses on related work about the Internet of Things, cloud, and message protocols. Several concepts and technologies are presented, and work based on the terminology is explained.

3.1 Scientific Work related to Internet of Things

In 1999 Kevin Ashton used to term ”Internet-Of-Things” in one of his presentations. According to Ashton et al. (2009), he was the creator of the term and the philosophy behind it. The concatenation of the two words ”Internet” and ”Things” basically describes the shift from the ”classic” internet, where every information is created by humans, to a new idea, where information is created and reprocessed by machines - the so-called ”things”. Computers combined with RFID technology could thus interact with the real world for the first time while also being connected amongst themselves.

Since 1999 several definitions and changes to IoT appeared, and it is still changing. In 2014 Li, Xu, and Zhao (2014) created an overview of IoT in the past, present, and future.

A few significantly milestones are presented in Madakam, Ramaswamy, and Tripathi (2015) such as:

- *1999*: The term Internet of Things is coined by Kevin Ashton.
- *2000*: LG announced its first Internet of refrigerator plans.
- *2003-2004*: RFID is deployed on a massive scale by the US Department of Defense in their Savi program and Wal-Mart in the commercial world.

- *2005*: The UN's International Telecommunications Union (ITU) published its first report on the Internet of Things topic.
- *2008*: Recognition by the EU and the First European IoT conference is held.
- *2008*: US National Intelligence Council listed the IoT as one of the 6 "Disruptive Civil Technologies" with potential impacts on US interests out to 2025.
- *2010*: Chinese Premier Wen Jiabao calls the IoT a key industry for China and has plans to make major investments in Internet of Things.

The Internet of Things is nowadays a fast-growing field with many different technologies and application fields, such as medical, industrial, and governance. Madakam, Ramaswamy, and Tripathi (2015) showed that there are also some flaws within IoT. Their main observation is the lack of standardization in definitions, in architecture as well as protocols. A step in the direction of standardization is made by Dorsemaine et al. (2015), who invented a taxonomy to create the greatest common divisor in terms of naming conventions between all technologies and fields in IoT.

Besides, Gigli and Koo (2011) tried to provide application developers a starting point in the confusing and overwhelming world of IoT and simplified IoT by introducing a categorization of its services.

To make IoT even clearer and standardize the technology, a three-layer architecture, as proposed by Zhonggui Ma et al. (2013), can be used, consisting of a physical layer, a network layer, and an application layer. However, this can still be extended to 6 levels, as suggested by M. Zhang, Sun, and Cheng (2012), who also discussed the key technologies used, including RFID, SOA, and WSN.

Another publication that focuses on the generalization of IoT was published by Kriti Chopra, Kunal Gupta, and Annu Lambora (2019), who reviewed the future of IoT while also explaining the overall basic workflow, some possible applications, and generic architecture.

One of the leading research areas is the authentication and security of IoT and clouds. Saadeh et al. (2016) discussed the main security issues in IoT and also compared several different authentication techniques as well as analyzing them regarding their security. Another overview of IoT

security strategies was made by Gou et al. (2013) The authentication and authorization are especially focused by Albalawi et al. (2019), who, after comparing and surveying different methods, concluded that there is no *best solution*. However, the selection of security methods should instead be based on the requirements of IoT systems and their security needs.

Nevertheless, there are always security aspects that must be taken into account when developing an IoT solution. Twenty of these considerations were presented by Singh et al. (2016), including the protection against attackers during data transmission, identity management, certification, and protection against the cloud provider itself.

3.2 Scientific Work related to Cloud

A brief introduction and the explanation of the basic terms to Cloud systems can be found in 1.5 as well as in Rimal, Choi, and Lumb (2009). Prajapati, Sharma, and Badgujar (2018) explained the history of the cloud, described the basic concepts, and gave an analysis using the example of a *smart city*.

Over the last years, many different benefits have been gained, and an enormous amount of applications, e.g., image processing (Ghaffar and Vu (2015)), voice and emotion recognition (Z. Zhang and Lim (2015)), have arisen by offloading heavy computations from one device to the cloud.

K. Kumar and Lu (2010) asked whether outsourcing computationally intensive calculations on mobile devices to the cloud can save energy and concluded that this is certainly possible, but that the problems of security, privacy, and reliability are again being faced.

Another topic of cloud research is the comparison and analysis, regarding a specific topic or in general, between different cloud providers, such as AWS, GCP, and Azure, presented in 2.1. Dordevic, Jovanovic, and Timcenko (2014) compared Microsoft Azure and Amazon Web Services regarding their performance, usability, and features, and came to the conclusion that each provider has its pros and cons but is generally similar. Since there is no direct winner of theses three, an evaluation has to be made for each situation. Ghaffar and Vu (2015) were examining AWS, CloudSigma, and Azure in

order to establish a satellite image processing service and identified AWS as their favorite because it met their needs the most.

A different approach to clouds was made by Buyya, Yeo, and Venugopal (2008), who looked at the cloud providers from a market-oriented point of view. They discussed the idea of computation-power-resource trading, where brokers mediate between cloud providers and consumers by buying capacities of resource power and lease these directly to the customer. However, one of the notable concerns was the interchangeability between cloud providers, requiring better support for interaction protocols to accomplish compatibility between them.

3.3 Scientific Work related to Message Protocols and Message Brokers

As described in 3.1, the *Internet of Things* is all about communication between *things* and the Internet. The language and its grammar, which is used by these *things*, is called *Message Protocol*. There exist numerous of these protocols, and some of them, which are particularly crucial for *IoT* and therefore for this thesis, have been described in 1.6.

Yassein, Shatnawi, and Al-zoubi (2016) did a more comprehensive survey about application layer protocols. In their detailed analysis and comparison, they concluded that each protocol has its strengths and weaknesses in a specific scenario and should be chosen according to the application requirements and environmental constraints. For example, MQTT is considered the right choice for low power devices, whereas a RESTFUL web service may be more useful for bigger machines due to its easy and simple implementation and well-known protocol base.

A web performance test of IoT messaging protocols, including AMQP and MQTT combined with different message broker implementations, was performed by Babovic, Protic, and Milutinovic (2016). As a result, they recommend MQTT in most IoT scenarios but also warn against using message brokers without further investigation. The message broker Mosquitto (2.2.2),

in particular, was used as a negative example due to its poor performance in certain scenarios.

Sreeraj, N. S. Kumar, and G. S. Kumar (2017) worked the other way around by implementing a framework that predicts the performance of MQTT and AMQP in different scenarios based on several variables, including the latency, packet loss, payload size, and *Quality of Service*, which is described in 1.6.1.

Guner, Kurtel, and Celikkan (2017) showed an excellent example of how to use a message broker and several message protocols to implement an IoT application. They set up an architecture consisting of several IoT sensors, a message broker, a so-called orchestrator, and mobile devices as data requestors. This system is especially relevant for this work, since the data providers, i.e., the sensors, are not connected to the Internet, but are in an internal network. Since the data-requestor, i.e., the mobile device, wants to query the data via the Internet, the orchestrator must form the bridge between the Inter- and Intranet. The intern communication was done by using MQTT and HTTP for authentication and authorization. However, this architecture was described as too complex to handle by Kang et al. (2017) because of the maintenance of the additional orchestrator and message broker. They instead advise using the AWS IoT Core, which acts as a message broker in the cloud. This system's disadvantage is the required internet connection of the IoT devices, which would not be feasible for primtec.

A comparative evaluation of AMQP and MQTT was done by Luzuriaga et al. (2015), who tested both protocols in scenarios similar to vehicular networks, where the connection may suffer, and the latency may be high. Both protocols performed well, and in conclusion, they recommend using AMQP to build reliable and scalable infrastructures.

The lack of additional security features using MQTT on devices with low computational powers was the topic of the paper written by Peniak and Franeková (2018). They invented an application node called the *Validator*, which validates the published data values based on a functional transformation and by comparison with an expected pattern.

Hoffmann et al. (2018) also analyzed the security and safety of MQTT, CoAP, and MQTT by comparing them to the safety standards IEC-61508 and IEC-61784. They describe the situation as a trade-off between more safety and more energy consumption and are currently working on adapting one of the existing protocols.

4 Problem and Requirements

The subject of this thesis is the investigation and implementation of an IoT/Cloud concept that is to be used by all products of primtec. The goal is that every primtec device should be equipped with the IoT functionalities, but in this thesis, the focus is on the "Packing Table" and "DataCollector", described in the introduction. This chapter provides a quick overview of the current situation at primtec and its customers while also describing the industrial field's limitation. In addition, section 4.3 lists all requirements that are needed for the end product.

4.1 Current Situation

The life-cycle of a primtec product begins with the installation and commissioning. Usually, a primtec employee must be on-site, which takes one to two full days, followed by several days of manual monitoring and observation.

The products are designed to operate continuously - reliability is mandatory but can not be guaranteed. As these products consist of many different components, for example, the operating system, the software, an RFID reader, and a PLC interface, problems will eventually occur - they can not be prevented. However, early detection could significantly reduce their effect.

If a problem occurs, it can currently take up to several days for the customer to recognize it, followed by additional days for internal problem solving and redistribution of the update, which again requires a primtec employee at the customer's premises. Since logging is done offline, the log files must be gathered by a worker and sent to the primtec office for further investigation.

These processes are time-consuming, cost-intensive, and prone to error. Therefore, primtec needs a centralized server (cloud) that collects all logs, metadata, and status updates from all customers and their products. This makes it possible to detect failures at an early stage, update the products remotely, and monitor each device separately without the need of a primtec employee to be on-site at the customer's premises.

4.2 Constraints

Since most customers are from an industrial sector, client-side internet connection to a server may not be possible due to security reasons. The local network may also fail since the WIFI connection may not be available at all locations.

primtec uses the Microsoft .Net framework for all projects, which must be easy to combine with the IoT/Cloud solution of this thesis. Additionally, primtec has a Microsoft subscription, which includes Microsoft Azure. Therefore other non-open-source IoT or cloud frameworks may be ineligible.

Windows is not the only platform that should be supported. In the future, the cloud system should also be usable with Linux and Android devices.

As described in 1.4 as well in 3.1, IoT, and the technologies associated with it are rapidly growing and ever-changing. Therefore it is essential that the IoT solution of the thesis can be extended and changed without much effort. Besides, a so-called vendor lock-in should be avoided. Vendor lock-in describes a scenario where a customer cannot easily replace a used service or product with an equivalent solution from another provider.

4.3 Requirements

To be able to compare different solutions for the given problems, requirements are defined. Some of these requirements are essential, while others are less important in the first step. Therefore each requirement can have

either high, medium, or low importance. An overview of all requirements can be seen in Table 4.1.

Requirements	
Requirement	Importance
R01: Reliability and Quality	High Importance
R02: Confidentiality, Integrity and Authenticity	High Importance
R03: Restricted access to the World Wide Web	High Importance
R04: Web Interface	High Importance
R05: Easy Integration to existing Products	High Importance
R06: Logging to the cloud	High Importance
R07: Monitoring Data	Medium Importance
R08: Remote Commands	Medium Importance
R09: Statistics	Low Importance
R10: Notification System	Low Importance
R11: Availability	Low Importance

Table 4.1: The requirements and their importance

4.3.1 R01: Reliability and Quality

A highly reliable system describes a system which continuously performs according to its requirements, whereas the quality defines how it performs these actions.

Regarding this thesis, a solution is needed that guarantees that the data sent by the client to the cloud is not lost. As described in 4.2, clients can be offline for an undefined time, hence data buffering is required. Once the client reconnects, all buffered data must be transferred to the cloud. To ensure that the data is received by the cloud, an acknowledgment needs to be sent from the cloud to the data sender, thus directly to the client or the message broker.

4.3.2 R02: Confidentiality, Integrity, and Authenticity

In IT security, confidentiality refers to protecting the data against attackers, which in the case of this thesis, means that the information in the data is not readable to anyone but the trusted clients and the cloud system itself.

Integrity defines that data sent by one party, in this case, the client, to another party, the cloud, has not been altered during its transmission and also has no errors.

On the other hand, authenticity ensures that the data being sent is really from this client and not from another party.

All three keywords are of high importance and must be ensured since customers of primtec are generally from the industrial sector, and industrial espionage and manipulation pose a real threat.

4.3.3 R03: Restricted access to the World Wide Web

Due to security reasons, most customers of primtec have their own intranet, which is used for data transmission, sharing information, machine control, or simple chat tools. In most cases, this intranet is accessible throughout the entire company premises via WiFi hot spots.

A connection from outside the intranet is usually prohibited and only feasible in exceptional cases under strict control. To create an IoT solution, where multiple IoT clients exchange data with a decentralized cloud, it is necessary to keep the access points from inside to outside and vice versa to an absolute minimum.

To accomplish this, a so-called *orchestrator* is required, which acts in the intranet as a central point of contact for all data traffic as well as managing the clients. This service is the only one equipped with internet access, whereas the IoT clients communicate only within the intranet.

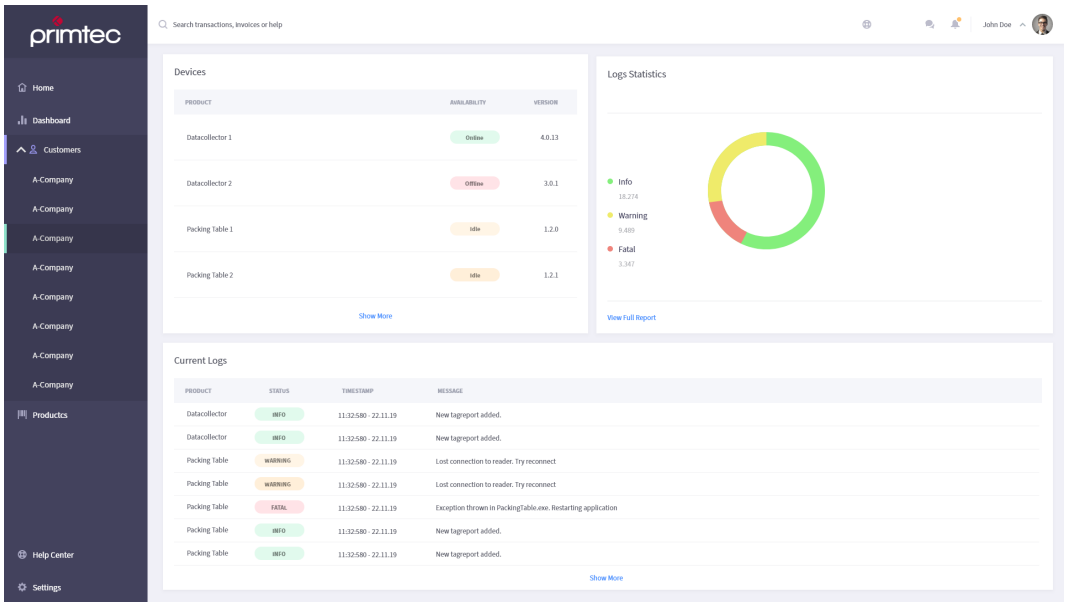

Figure 4.1: Website concept mockup showing all devices of a customer, their states and the log collection

4.3.4 R04: Web Interface

The features described above should be easy to access by a primtec employee - therefore, a web interface is required. After a login procedure, the primtec employee should see all primtec software clients currently available and the associated operating data (4.3.7). The remote commands (4.3.8) should be executed with one button click, and the statistics should be clearly presented.

A design concept of such a web interface can be seen in Figure 4.1. The customers are listed in the navigation area on the right side. By clicking on a customer, a general information page should appear, showing all currently used devices, their log entries, and statistics. The overall status at the customer's site can be easily recognized by the color-coding of the log levels, monitoring data, and device statuses. This web interface should initially only be visible to primtec employees, but a dedicated web interface for customers would also be conceivable in the future.

4.3.5 R05: Easy Integration to existing Products

In recent years, many different solutions have been produced by primtec for many different customers. A subsequent integration should not require much effort, and future primtec developers should be able to use this system without much additional work.

This requires a system that is as transparent as possible and that operates from the background without having an impact on other processes. Device configuration and user intervention are to be brought down to a minimum to provide a *plug and play* user experience.

4.3.6 R06: Logging to the Cloud

Logging is an essential part of software development as it is used for troubleshooting, analysis, and general information. The log entries can be seen as a diary of the software consisting of one-time messages describing the current action of the devices, problems that occurred, or simple information with the addition of a timestamp and a log level that can be interpreted as the importance of the log entry. These logs enable a developer to understand and reconstruct the procedures of the software.

primtec software uses the inbuilt .NET log framework to perform logging with six different log levels:

- Trace
- Debug
- Information
- Warning
- Error
- Critical

This framework also offers the possibility to define the target of these logs, such as a file or a simple console window. These log entries are currently saved in a local file that can be opened and examined with a simple text reader software. Since this requires direct access to the device and a worker on-site, it is not feasible with several customers distributed worldwide.

One goal of this thesis is to extend the log targets by a cloud destination. From the programmer's point of view, the whole cloud logging should be completely transparent, which means that there should be no additional function calls, but only a small setting in the software configuration.

4.3.7 R07: Monitoring Data

In addition to logging, primtec needs a system to store specific operating data generated by the software and the machines. This data could include software versions, temperature values, hardware load, or data describing the status of the RFID readers. All this information should be stored in another database and easy to access via the web interface (4.3.4).

Critical device data must be easily recognized by color-coding the individual states, e.g., red for errors and yellow for warnings.

To show the online/offline state of each device, a heartbeat system is needed. After a certain amount of time, a device can be assumed as offline.

Based on the monitoring, a primtec employee could then attempt to optimize the processes, correct errors, or even analyze the yield for the customer. This feature is not necessary for the first step, but it should be possible to implement it in the future.

4.3.8 R08: Remote Commands

As access to the machines is usually difficult due to local distances, remote commands are required. These commands can include prompts to initiate an update process, reconfigure the RFID reader's parameters or increase and decrease the log level as described in 4.3.6.

Especially the update procedure is one of the most time-consuming processes at primtec. The software update is currently sent to the customer by Email, where one of the workers has to update each device manually. This can be avoided by triggering an update process remotely. Therefore, the

online web interface has to display each device's current software version while also providing the functionality to start the update procedure.

Since primtec products can be offline for longer periods, these commands should be executed as soon as the device gets an online connection to the cloud system.

For the implementation of the remote commands, two different approaches were be considered:

- *Dynamic Commands*: Dynamic commands can be seen as a regular program code which could be sent from the web-service to the client. This code gets then processed and executed by the client.
- *Static Commands*: In contrast to dynamic commands, static commands only use a fixed set of commands. Each command from the web service triggers a particular action on the client.

Dynamic commands would have the benefit of being completely dynamic. The website user could write individual code which then gets executed as soon as it arrives at the client. This approach, however, would be highly vulnerable to security attacks. Since the just written code gets sent through the Internet, manipulation of the messages can not be ruled out (*man-in-the-middle attack*), so that the introduction of malicious code would be possible. To keep the attack vector as low as possible, static commands should be used.

4.3.9 R09: Statistics

Another add-on is statistics, which should be visible in the web interface (4.3.4). These statistics should show how the values described above (4.3.7) change in a given time window. Since these statistics are not only of interest to primtec but also to the customer itself, an export feature is required, making it possible to pass on this data to the customer.

The importance of statistics is low in the first step, but would later be an enriching feature for the customers.

4.3.10 R10: Notification System

Based on the data described in 4.3.7, a notification system would be another conceivable feature. If, for example, the CPU load of a computer at the customer exceeds a critical value, an Email or SMS should be sent to a primtec employee.

Similar to 4.3.7, this is not a quintessential feature, but it should be possible to implement it as an add-on.

4.3.11 R11: Availability

The availability of a system defines the time frame in a certain time interval in which the functionality of this system is available. Cloud provider usually outline their availability with a ratio of server up-time per year. Azure (2020), for instance, guarantees an availability of 99,99%, also called a "four nine" availability, for most of their cloud database services. This means that their cloud service can have a maximum server down-time of 4 minutes 23 seconds per month. Another method to categorize the availability of machines and services in classes is the *Availability Environment Classification* (AES) developed by the *Harvard Research Group* (Kaskade (2020)). The classes are:

- *Conventional* (AEC-0): Function can be interrupted, data integrity is not essential.
- *Highly Reliable* (AEC-1): Function can be interrupted, data integrity must be however ensured.
- *High Availability* (AEC-2): Function may be minimum interrupted only within fixed times during the main operating hours.
- *Fault Resilient* (AEC-3): Function must be maintained, within fixed times during the main operating hours, continuously.
- *Fault Tolerant* (AEC-4): Function must be maintained continuously, 24-7 enterprise (24 hours, 7 days the week) must be ensured.
- *Disaster tolerant* (AEC-5): Function must be available under all circumstances.

The cloud availability is not of high importance to primtec since the cloud solution is an add-on to their products, reducing the error rate in the longer term. Immediate error correction is not possible due to the possibility of offline clients, and therefore a cloud availability of 99%, which means less than 8 hours per month, would already be sufficient. In terms of the *Availability Environment Classification*, the class AEC-1 would be adequate.

5 Contribution

The contribution of this thesis consists of two parts: a review of the cloud providers and the actual implementation.

First, in part one, a comparison between the cloud providers, presented in 2.1, based on the pricing and features, and between the message brokers, presented in 4.2, was made. Furthermore, the relevance and need for the primtec IoT solution were determined. The results can be seen in Table 5.2, Table 5.4, and Table 5.3.

Then, in part two, based on the constraints given in section 4.2 and on the requirements stated in section 4.3, a solution was implemented. The model of the implementation was the idea presented by Guner, Kurtel, and Celikkan (2017), which was explained in chapter 3.3, but due to the requirements, several modifications and adaptions have had to be made. An overview of the actual architecture is visualized and explained in 5.2.

The solution itself consists of a total of five parts, which are broken down and explained individually. A detailed description of the individual technologies used can be found in chapter 2. Visual Studio 2019 was used as development platform, and some software libraries by primtec were utilized, which get explained briefly but not in detail due to the corporate secret.

5.1 Review and Comparison of the Cloud Providers

5.1.1 Free Trial

Each of the three cloud providers offers different types of free trials for new cloud developers.

Google Cloud Provider - Free Trials

Google offers a whole year of free usage for multiple different products. These products include "Compute Engine", "Cloud Storage", "App Engine", and "Firestore", which is used for databases. The full list can be found on Google's website: https://cloud.google.com/free. Each of these free products is limited to a certain amount of storage or requests per month, which is usually more than enough to start developing a cloud system. In addition, Google Cloud Provider allocates 300$ for new users, which can be spent on non-free products.

After 12 months or after spending the 300$, the free trial ends, which means that all resources are stopped, and all data in the *Compute Engine* is lost. However, by upgrading to a paid account, the data can be restored within the next 30 days.

Microsoft Azure - Free Trial

Microsoft also offers, similar to Google, a free trial account for one year. During the year, a wide range of free products can be used, each limited by either a time window or a certain amount of storage (Free Azure Products). To register for the free trial, the user has to start with a free 30-day account. As a bonus, Microsoft provides 170$, which can be spent within this time frame. If the user spends more than the 170$ or after 30 days, an account upgrade has to be performed to unlock the full 1-year trial.

From this point on, the user may be charged for exceeding the free products' limits or for using the non-free products.

Amazon Web Services - Free Trial

AWS also offers a 1-year free trial, which includes over 60 different products (Free AWS Products). These can be used within the 12 months or even forever, but are also limited by a certain amount of storage or requests per month. Additionally, a rotating selection of new products can be used for a short time free of charge in order to test them.

5.1.2 Pricing Examples

Comparing the three cloud providers based on pricing is a difficult task as it depends heavily on metrics such as server availability, message count, and hardware. Online services like `www.cloudability.com` and `www.reoptimize.io` were only created to compare the prices in specific scenarios, which in turn are not free of charge. For this thesis, the largest common divisor between these services regarding primtec was chosen - a cloud database for saving the device logs.

Thus, three different scenarios were created that could be relevant for primtec, a low-budget scenario, a mid-tier scenario, and a high-end solution.

Low-Budget Scenario

The low budget scenario focuses on keeping the price as low as possible. Therefore, the device logs are only transmitted in a two-hour time frame. This results in a cloud database that only needs to be online for a small amount of time, saving much money. However, the disadvantage is that the device logs are not available in real-time, and the user may have to wait a whole day to receive the corresponding logs. Since real-time can not be achieved, the latency of the connection is not essential. Each log entry

expires after one week, which keeps the required amount of storage low. This scenario is regarded as the starting point for a cloud environment, which means that only a few customers with few IoT devices are supported. The type of database management system does not matter, so the cheapest one is selected.

Mid-Tier Scenario

The mid-tier scenario is an upgraded version of the low-budget scenario. The logs are transmitted within a few seconds and saved for one week. The system should be able to handle up to 10 customers, each with up to 10 devices. If available, Microsoft SQL Server or PostgreSQL is used as a database management system, and a backup mechanism is provided.

High-End Scenario

In this scenario, the cloud should handle hundreds of customers with up to one hundred devices. Database backups and duplicates are needed to ensure a 100% reliability. The data should be stored for one month, and all additional service features by the cloud providers are used.

Instance Type Details			
Name	Provider	# vCPUs	RAM per Core
db-pg-f1-micro	GCP	1	0,6 Gb
db.t3.micro	AWS	2	0,5 Gb
db.t3.medium	AWS	2	2 Gb
db.x1.16xlarge	AWS	64	15,25 Gb
db.z1d.2xlarge	AWS	48	8 Gb

Table 5.1: Details to hardware acronyms used in the tables below

Google Cloud Provider			
System	Low Budget	Mid-Tier	High End
Database System	Postgre-SQL	Postgre-SQL	SQL Server 2017 Standard
Number of Instances	2	4	4
Instance Type	db-pg-f1-micro	2 vCores - 4Gb Ram	32 vCores - 16 Gb Ram
Location	Frankfurt	Frankfurt	Frankfurt
Storage Type	HDD	SSD	SSD
Storage Amount	10 Gb	1 Tb	30 Tb (maximum)
Hours / Day	2	24	24
Days / Week	5	7	7
Backup Size	-	50 Gb	5 Tb
High Availability	no	no	yes
Price per Month	1,69 €	1.131,15 €	68.676,44 €

Table 5.2: The pricing of the three scenarios with Google Cloud Provider

Amazon Web Services			
System	Low Budget	Mid-Tier	High End
Database System	Postgre-SQL	Postgre-SQL	SQL Server Standard
Number of Instances	1	4	4
Instance Type	db.t3.micro	db.t3.medium	db.z1d.2xlarge
Location	Frankfurt	Frankfurt	Frankfurt
Storage Type	SDD	SSD	SSD
Storage Amount	20 Gb	1 Tb	16 Tb (maximum)
Hours / Day	2	24	24
Days / Week	5	7	7
Backup Size	-	500 Gb	16 Tb
Notes	-	RDS Aurora Backup	RDS Aurora Backup
Price per Month	3,63 €	1.227,79 €	47.395,56 €

Table 5.3: The pricing of the three scenarios with Amazon Web Services

Microsoft Azure			
System	Low Budget	Mid-Tier	High End
Database System	Postgre-SQL	Postgre-SQL	Azure SQL
Instance Type	1 vCore Gen 5	4 vCore - 32 Gb worker & 4 vCore - 16 Gb coordinator	80 vCore Gen5
Location	Germany West Central	Germany West Central	Germany West Central
Storage Amount	10 Gb	1 Tib (worker) & 1 Tib (coordinator)	30 Tb (maximum)
Hours / Day	2	24	24
Days / Week	5	7	7
Backup Retention	-	-	Weekly backup retention for 5 weeks
Backup Size	-	100 % provisioned storage	4 Tb
Tier	Basic	Memory Optimized	Business Critical
Price per Month	2,48 €	1.614,44 €	43.628,53 €

Table 5.4: The pricing of the three scenarios with Microsoft Azure

Hardware Details

The cloud hardware, also called the instance type, is specified using an abbreviation or acronym. Table 5.1 lists all of them, used in the other tables to compare prices, and describes the hardware architecture in detail.

5.1.3 Choosing a Cloud Provider

Google Cloud Provider was chosen as the host for the database used in the implementation since the free trial program was most appealing and easy to set up. *High Availability* can be enabled, ensuring that the database is spread across multiple data centers, preventing long downtimes due to maintenance or network errors. However, any cloud provider can be used, as databases are supported by any cloud provider, and *High Availability* is offered by all three.

5.1.4 Message Broker Comparison

Choosing a message broker was not a particularly challenging task since all of them are sufficient for the solution. Encrypted communication is the most crucial feature, which all of them presented in 1.6, provide. RabbitMQ was chosen, which also has the benefit of being an open-source project. It provides *NuGet packages*, the libraries used while developing a C# project, which also favored this choice. During the implementation, attention was paid to the interchangeability of these brokers in order to be able to exchange the brokers without significant notable in case of later design changes.

5.2 General Architecture

Like Guner, Kurtel, and Celikkan (2017), the solution consists of a cloud/server, a message broker, an orchestrator, and multiple clients. The basic idea is to install the message broker on a server at the customer's premises, which

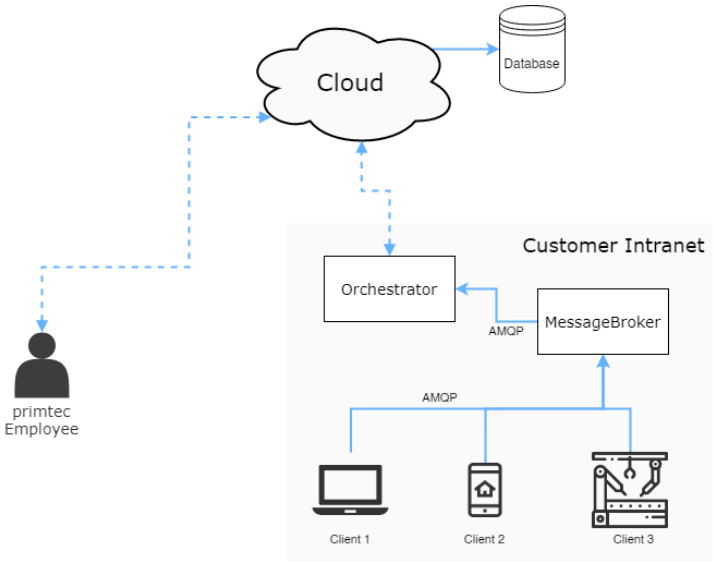

Figure 5.1: Architecture Sketch of the *IoT/Cloud* Framework

administrates the AMQP and MQTT communication. A sketch showing the architecture containing all components can be seen in Figure 5.1.

The actual IoT devices, from now on only called ”clients”, are connected to the customer’s intranet (e.g., via WIFI) and use AMQP and MQTT in order to broadcast their log information into the ”log-channel”, send heartbeat signals into the ”heartbeat-channel” as well as receive commands from the ”command-channel”. The message broker ensures that these messages get sent over the network to all subscribed parties and can also be configured to store these messages for future subscribers. It is important that the whole communication takes place in the local network - therefore, internet access by the clients is not needed, fulfilling the requirement ”Restricted access to the World Wide Web” 4.3.3.

Since all messages sent to the ”log-channel” have to be forwarded to the cloud, an additional instance is needed called the ”orchestrator”. The orchestrator is implemented as a Windows service, which can be hosted on the same local server as the message broker. A Windows service is similar to a regular application, except it is invisible for the user and operates from the background. The advantage of this solution is that after a reboot of the

system, and if configured correctly, Windows restarts the installed services, which means that no extra restart of the orchestrator by a user is needed.

The orchestrator represents the bridge between the cloud, hosted in the world wide web, and the customer's intranet. This means that the orchestrator is the only part that requires access to the Internet. The main task of the orchestrator is to listen to the log- and heartbeat-channel. As soon as it receives a client's log message, it transmits the message to a cloud database. Since remote commands are required (4.3.8), the orchestrator also forwards messages the other way round, which means that a predefined set of commands can be sent from the Internet (in this case, the web user interface) to the clients by using the the "command-channel".

The next component is the cloud, which is hosted on the world wide web. This implementation uses only a cloud database as cloud feature, which is offered by all cloud providers described in 2.1. It can easily be exchanged, and a "vendor lock-in" does not happen.

In summary, the implementation now consists of clients generating logs and sending them to the local network. These logs are then sent via the message broker to the orchestrator, which sends these logs from the local network to a cloud database on the world wide web.

The final component is a web service and web user interface, which, similar to a regular website, can be hosted with dedicated web hosting platforms, on an own server infrastructure, or a cloud provider, a feature provided by all three introduced cloud providers.

This website can be accessed using a regular web browser, e.g., Google Chrome, by using the corresponding IP address or, if needed, by a domain, which of course, has to be registered beforehand. All log messages by all clients are visible on this web site in a clear and structured way. They can be filtered and sorted, and additional information about the current state of the client is shown.

```
[
{rabbit, [
    {tcp_listeners, []},
    {ssl_listeners, [5671]},
    {ssl_options, [{cacertfile, "D:\\working\\Playground\\tls-gen\\basic\\result\\ca_certificate.pem"},
                   {certfile, "D:\\working\\Playground\\tls-gen\\basic\\result\\server_certificate.pem"},
                   {keyfile, "D:\\working\\Playground\\tls-gen\\basic\\result\\server_key.pem"},
                   {verify, verify_peer},
                   {fail_if_no_peer_cert, false}]}
    ]}
].
```

Figure 5.2: The RabbitMQ configuration file using SSL

5.3 RabbitMQ Message Broker

The backbone of the whole implementation is the *RabbitMQ*[1] message broker (presented in section 2.2.1), which is available for Windows, macOS, and Linux. After installation and before starting the broker service, TLS has to be enabled to meet the requirement "Confidentiality, Integrity and Authenticity" (4.3.2). Detailed information on how TLS ensures these features can be found on *How TLS provides identification, authentication, confidentiality, and integrity* by IBM (2020).

Configuring RabbitMQ can be done by using the configuration file, shown in Figure 5.2. To use a TLS encrypted communication, a signed certificate file is needed for the RabbitMQ message broker along with a certificate for each client and one for the orchestrator. In this proof of concept setup, self-signed certificates were generated using the *tls-gen*[2] tool available on GitHub.

It is important to note that these certificates are meant to be used in the development and should be swapped with real signed certificates provided by the customer's network administrator.

In addition to the TLS communication, the second RabbitMQ authentication feature is used: a username/password pair for each client and the orchestrator. These login credentials can either be configured on the RabbitMQ web UI, which is hosted locally as soon as the message broker is started, or by using the RabbitMQ command-line tool *rabbitmqctl*.

[1] https://www.rabbitmq.com/download.html
[2] https://github.com/michaelklishin/tls-gen

The user authentication system provides the possibility to manage permissions for each channel. For this solution, the ”log-channel” can only be read by the orchestrator user, while the client users are the only ones with the permission to write in this channel. The *command channel*, on the other hand, is configured in exactly the opposite way.

Since the user credentials are by default sent in plain text while establishing the connection, it is essential to use the user authentication only in combination with TLS.

Finally, the plugin *rabbitmq-mqtt* was enabled using *rabbitmqctl* to allow MQTT messages next to AMQP. The RabbitMQ message broker should be running on a separate machine and needs access to the customer's local network.

5.4 The Client Software

5.4.1 NLog Configuration

Software by *primtec GmbH* is usually contained within an application container which provides the UI skeleton, basic functionalities as well as the basic *NLog*[3] logging configuration. NLog is a free, open-source logging framework designed for .NET, and can be used by simply including the dedicated NLog NuGet package to the application project. The benefit of NLog is that the developer only has to configure the *NLog.Logger* once in this application by loading an NLog configuration file.

Inside the configuration file, which can be swapped on demand, one can specify log targets such as files, a database, a simple console window, and many more while also defining each target's log level.

For this solution, a local SQLite database target was added to the existing ones inside the primtec NLog configuration file, which can be seen in 5.3. This ensures that each log message gets written into a local database in

[3] https://nlog-project.org/

```
<?xml version="1.0" ?>
<nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <targets>
    <target xsi:type="BufferingWrapper" name="buffered" bufferSize="10" overflowAction="Flush">
      <target xsi:type="File"
          layout="${date:format=yyyy-MM-dd HH\:mm\:ss,fff} ${level:uppercase=true} ${logger} - ${message}"
          fileName="${basedir}/Logs/logfile.${date:format=yyyyMMdd}.log"
          keepFileOpen="false"/>
    </target>


    <target xsi:type="File" name="file"
          layout="${date:format=yyyy-MM-dd HH\:mm\:ss,fff} ${level:uppercase=true} ${logger} - ${message}"
          fileName="${basedir}/Logs/logfile.${date:format=yyyyMMdd}.log"
          keepFileOpen="false"/>

    <target name="db" xsi:type="Database"
         dbProvider="System.Data.SQLite.SQLiteConnection, System.Data.SQLite" keepConnection="false"
         commandType="Text"
         connectionString="Data Source=${basedir}/Logs/log.db3;Version=3;"
         commandText="INSERT into Log(Timestamp, Loglevel, Logger, Callsite, Message)
          values(@Timestamp, @Loglevel, @Logger, @Callsite, @Message)">
      <parameter name="@Timestamp" layout="${longdate}"/>
      <parameter name="@Loglevel" layout="${level:uppercase=true}"/>
      <parameter name="@Logger" layout="${logger}"/>
      <parameter name="@Callsite" layout="${callsite:filename=true}"/>
      <parameter name="@Message" layout="${message}"/>
    </target>
  </targets>

  <rules>
    <logger name="*" minlevel="Info" writeTo="buffered" />
    <logger name="*" minlevel="Info" writeTo="db" />
  </rules>
</nlog>
```

Figure 5.3: The NLog configuration file

addition to a file. An additional custom column named ”Synced” was added to mark previously synced log entries.

5.4.2 Log, Heartbeat and Command Service

The previously described database target can now be directly examined on the machine with a database software, but since these logs should be forwarded to the cloud, an additional service was needed - the ”primtec syncing service”. This company owned Windows service is usually used to synchronize data between two databases or files permanently, and therefore was extended by an MQTT and an AMQP destination.

As soon as the ”syncing service” starts, all entries in the database are traversed and sent to the RabbitMQ ”log-channel” by serializing the log message object to a JSON string, also called ”marshalling”. The user credentials were added to the RabbitMQ send method as properties, otherwise, these messages would be rejected by the RabbitMQ message broker. As mentioned above, a filter is used that excludes all database entries where the column ”Synced” is set to true. This assures that previously sent logs are not sent again. The synchronization service can be configured to synchronize the data once or infinitely often in a specified time interval. For this solution, a time interval of 30 seconds has been defined so that the data is sent in batches rather than individually. The prerequisite for a successful sync is, of course, an existing network connection. If this is not present, the sync service waits for the next trigger.

Another service was implemented, called the ”heartbeat service”, which sends messages to the ”heartbeat channel” in a given time interval. Implementing a service can be done by using a NuGet package made by *Topshelf*[4], which is used by primtec GmbH in many different projects. After extending an application with the Topshelf functionalities, it can either be started as a simple executable or installed as a service. The heartbeat message, sent similarly to the log messages as a JSON string, contains information about the device, customer identification, and timestamps and is used to check the client’s current state.

[4] http://topshelf-project.com/

The last service, the "command service", which does not send messages but listens to incoming command requests sent on the "command-channel".

The primtec application container has been extended such that these services get started in an individual background thread at application start. The server credentials are saved within a *App.config* file, which gets read on application start. Otherwise, no additional configurations are needed by the user, which meets the requirement "Transparent Integration to existing Products" 4.3.5.

5.5 The Orchestrator

The orchestrator is implemented as a windows service, as described in 5.2, and should be running on a separate machine at the customer's site but can also be hosted on the same device as the RabbitMQ message broker. Unlike the other components, it requires a connection to the Internet. In industrial settings, such access to the Internet is seen as a security issue and therefore has to be granted explicitly by the administrator according to the company policy.

As seen from the RabbitMQ side of view, the orchestrator is not different from a regular client except it uses the predefined orchestrator user credentials. This gives it the possibility to listen to the log-channel and the heartbeat-channel while also writing to the command channel. As soon as it receives messages in the log-channel, it "unmarshalls" the JSON string back to the original log message objects and saves it to a predefined remote database by using the *.Net Entity Framework*[5].

.Net Entity Framework is the open-source library made by Microsoft, used to work with databases. Most of today's standard database systems are supported and can be downloaded and used via the corresponding database provider NuGet package. primtec GmbH generally uses SQL Server, PostgreSQL, Oracle, and SQLite, all of which are supported. Switching from one database system to another can be done by swapping the database connection string found in the corresponding configuration file. This makes

[5] https://docs.microsoft.com/en-us/ef/

it straightforward to move from one database to another without getting stuck in the ”vendor lock-in”.

If heartbeat messages are received on the heartbeat-channel, they get ”unmarshalled” and saved to the remote database, similarly to the log-messages.

To handle and forward remote commands from the website, an RPC server class was implemented. *RPC* (remote procedure call) is a form of client-server communication, where procedures are provided by an RPC server, which can be called from clients over a shared network, similarly as a local procedure call would be implemented. Many open-source libraries are available on the Internet that could be used in this solution, but due to the fact the *primtec* used *gRPC*[6] in other projects, this particular open-source framework was chosen.

This class was integrated into the orchestrator, which means that from the view of the web-api, the orchestrator acts as an RPC server, whereas the web-api represents an RPC client. In the current state of development, only a single remote command was defined - the ”update command”. The update command utilizes a *UpdateRequest* object as message payload, consisting of an update-file-path and a client name. If this update command is called from the web-api at the orchestrator, this *UpdateRequest* object gets first serialized again applying JSON, and then sent to the clients using the command-channel. Based on the object type of the message data, the client can then start specific procedures, for example, an *update-procedure*, which, however, was not implemented during this work. The client’s name inside this message object could be used to reject remote commands if not matched with the client’s name. The gRPC communication is encrypted by using SSL/TLS, assuming certificates are used correctly.

5.6 The Cloud

The database described in 5.5 could be hosted on a server at the customer’s premises, but since the log messages should be accessible from around

[6] https://grpc.io/

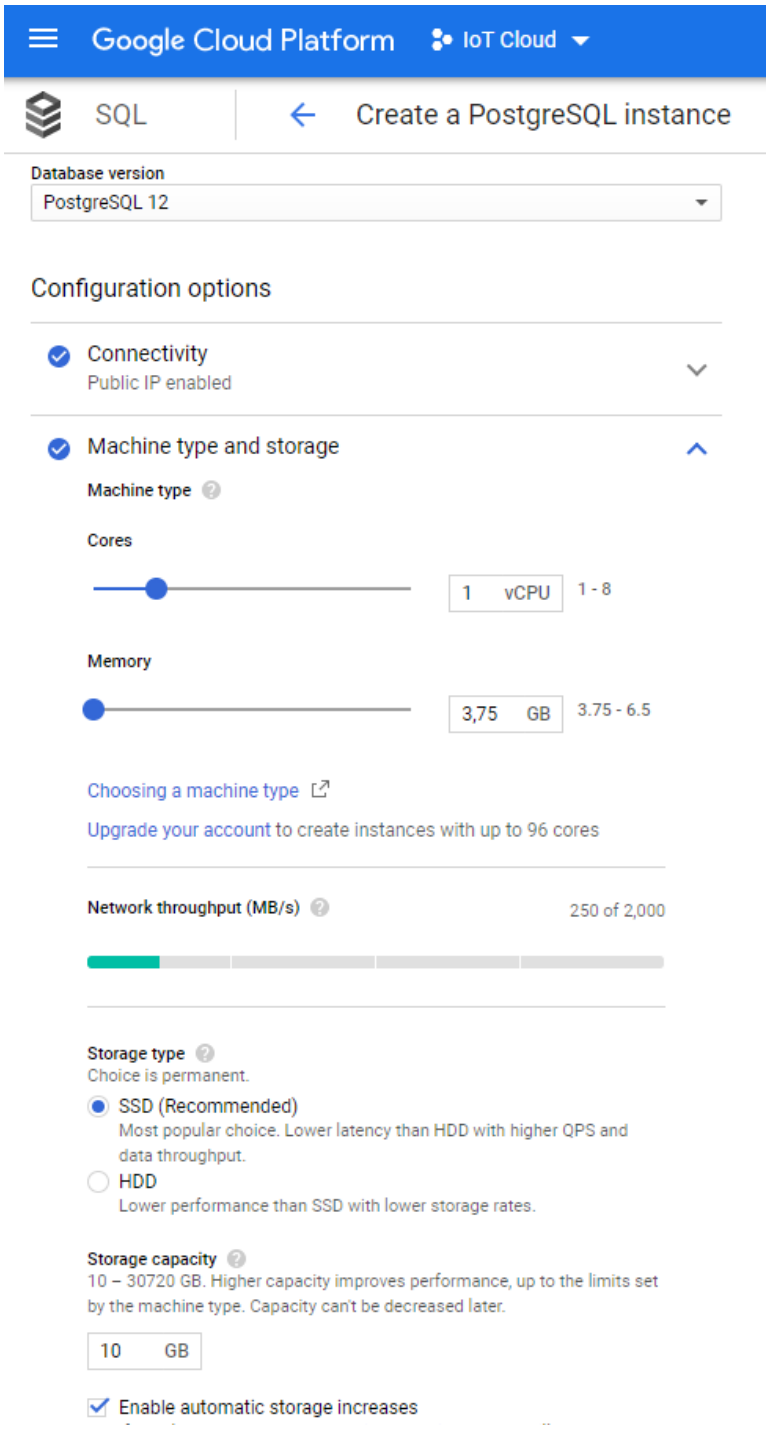

Figure 5.4: Configuration of the Google Cloud Database

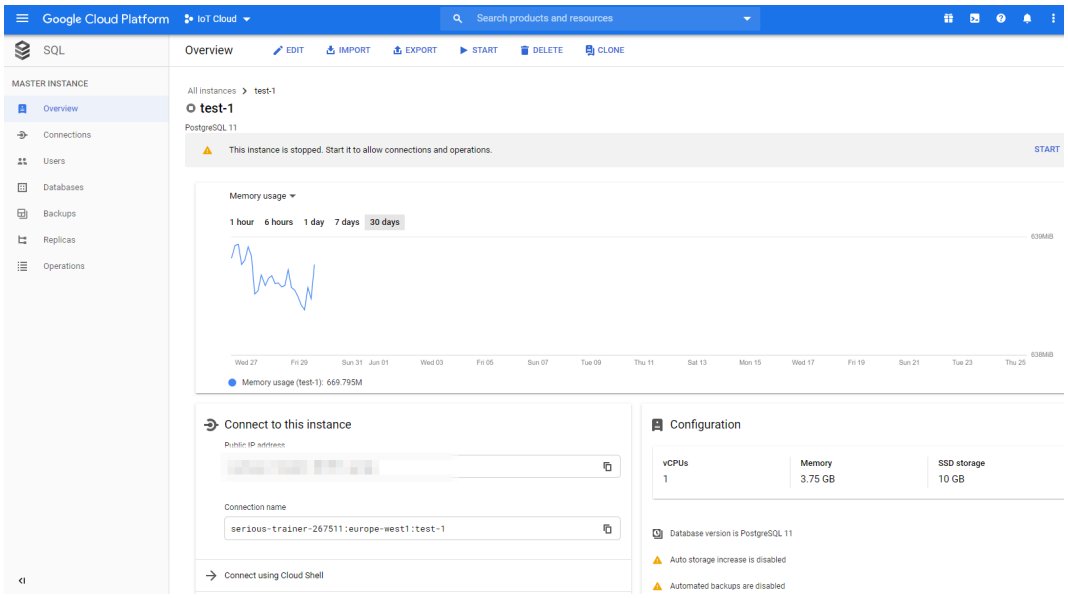

Figure 5.5: The Google Cloud Database Dashboard

the world, a cloud database was configured. To save costs in this proof of concept setup, the most cost-effective cloud database provider from the three providers presented in 2.1 was chosen.

At that time, this was the Google Cloud hosting a PostgreSQL instance, but since the database used by the orchestrator can be changed as described in 5.5, any other could have been used.

As a requirement for a Google Cloud database, a Google account is needed. After registration, a cloud product can be chosen, in this case, the PostgreSQL database, and then configured by defining the region of the database server, arranging the hardware components, and generating the admin password. The cloud configuration screen can be seen in Figure 5.4. Additionally, backup and maintenance time frames, if additionally purchased, can be specified.

After the setup, the cloud database is finally assembled and ready to use. The dashboard 5.5 shows the current memory usage, CPU utilization, as well as the public IP address. The database itself consists out of six tables: the *Log-table*, the *Heartbeat-table*, the *Device-table*, the *User-table*, the *Role-table*

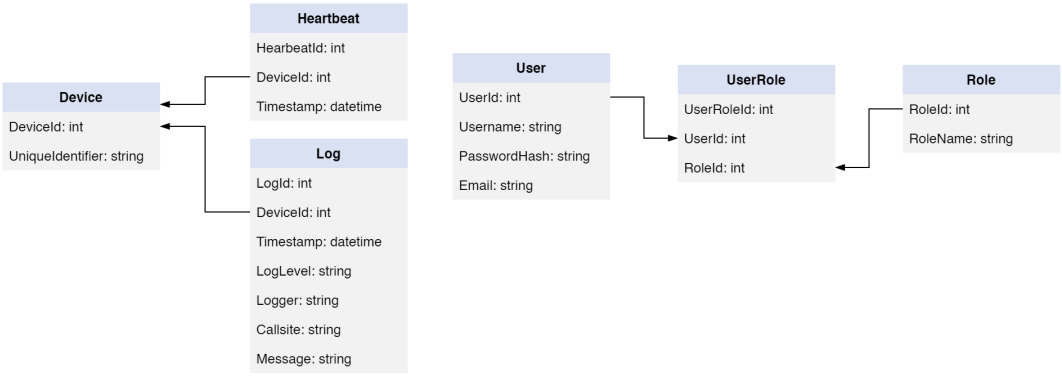

Figure 5.6: Database architecture

and a *User-Role-Mapping-Table*. The heartbeat- and log-table each reference a device using a foreign key. The user-table is needed for the user login on the website. Also, a role can be chosen for each user, mapped using a mapping table. An overview of the database tables and their relations can be seen in Figure 5.6.

5.7 The Web Service

The last component is the web service and its web view, shown in Figure 5.7. It was implemented in *ASP.Net*[7], an open-source web-application framework by Microsoft. Most of the web applications by primtec GmbH are implemented using the ASP.Net framework since they can be developed using C# and the entire primtec code base. Due to a previous primtec project, *Vue*[8] combined with *Vuetify*[9] was used for the front-end implementation. The correct corporate design of primtec was achieved by utilizing Vuetify's UI components, which provide a predefined style according to the Google Material Design, and by using the primtec colors.

In the current expansion stage, the website has the following features:

[7] https://dotnet.microsoft.com/apps/aspnet
[8] https://vuejs.org/
[9] https://vuetifyjs.com/en/

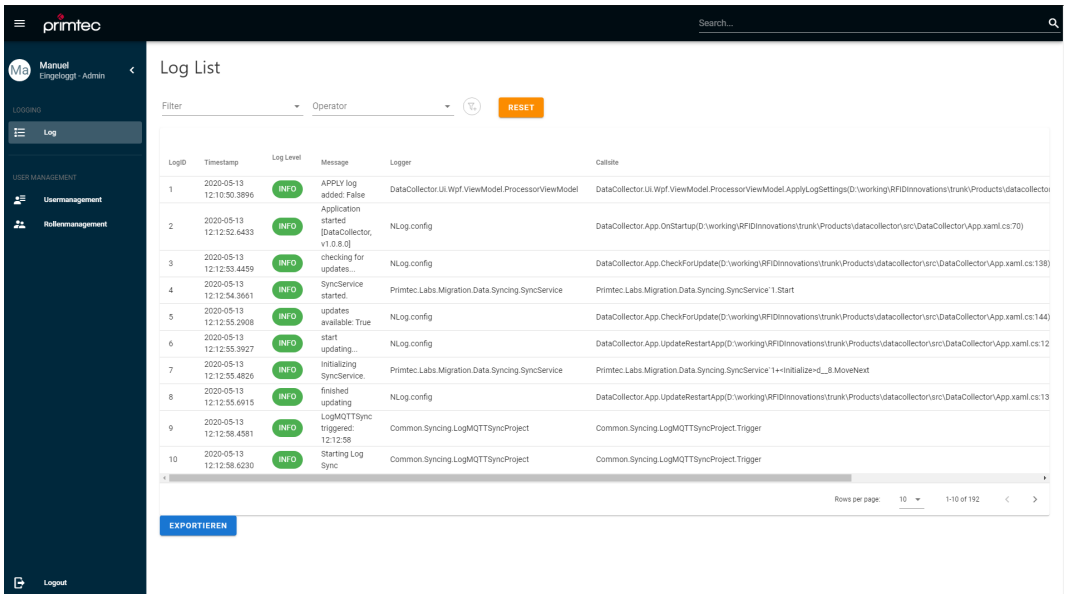

Figure 5.7: Screenshot of the Web User Interface

1. *User Management*: To access the website, a user account with a username/password combination is required. These accounts can be created and edited with the user management feature. In addition, a user can have different roles, such as an administrator or customer role, which can be used for certain restrictions on the website. The user objects and their hashed passwords are all stored on the same database as the logs.
2. *Log View*: The log view shows the received logs in detail. The log message, the client, the timestamp, and the log level are shown in different columns and can be filtered separately. An export feature makes it possible to download the whole table as an Excel file.
3. *Device Status View*: The device status view shows all devices which have sent log messages. An offline/online status visualizes if this device has sent a heartbeat signal in the last time-frame, which can be configured using a config file.
4. *Remote Commands*: The last feature is the remote command triggering using the according gRPC client class described in 5.5. Next to the device status in the view described above, buttons, representing these

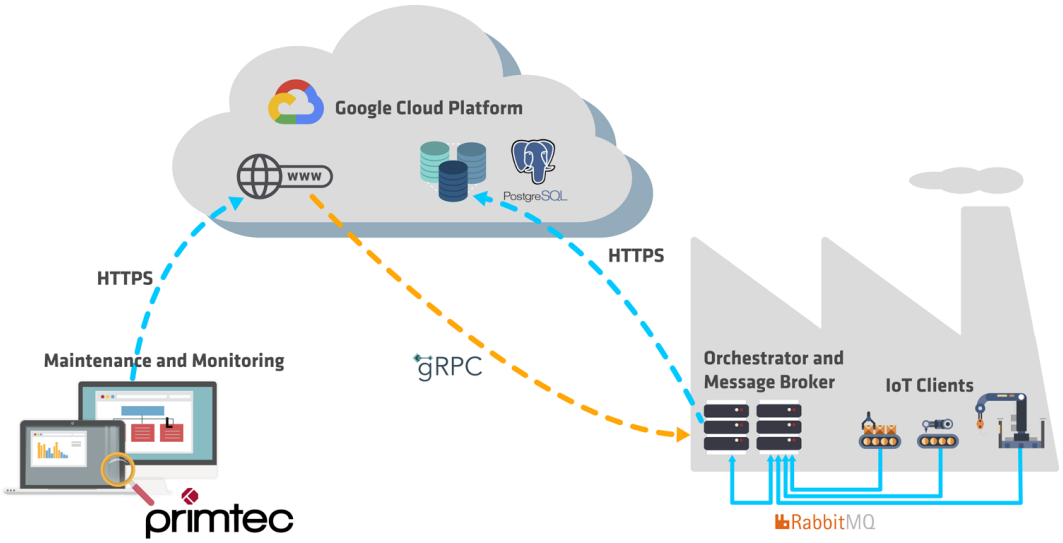

Figure 5.8: The final architecture of the implementation, *graphics taken from freepik*

remote commands, can be clicked. An information message is shown if the remote command was triggered successfully.

The website can be hosted in many different ways, starting with the local machine used during development. IIS (Internet Information Services), a web server software from Microsoft, can be used as a professional solution as well as a cloud provider. For this thesis, the Google Cloud was used, as it was already configured to host the PostgreSQL database. The whole project can be published to the "Google App Engine" by defining a configuration file in the "PublishProfiles" folder created by Visual Studio.

An illustration of the final architecture, including all used components, can be seen in Figure 5.8

6 Discussion and Evaluation

The solution presented in chapter 5 was implemented based on the constraints and requirements shown in 4.2 and 4.3. This chapter analyzes the implementation and shows how each requirement was met. For a detailed evaluation, the *N-P-L-F* system, standardized with ISO/IEC 15504 (Mesquida, Mas, and Amengual (2011)), was used assigning each requirement one of the four values:

- *not achieved*: $(0 - 15\%)$
- *partially achieved*: $(15 - 50\%)$
- *largely achieved*: $(50 - 85\%)$
- *fully achieved*: $(85 - 100\%)$

An overview of each requirement, its importance and the corresponding N-P-L-F rating can be seen in Table 6.1.

Requirement Evaluation		
Requirement	Importance	NPLF Rating
R01: Reliability and Quality	High Importance	fully achieved
R02: Confidentiality, Integrity and Authenticity	High Importance	partially achieved
R03: Restricted access to the World Wide Web	High Importance	fully achieved
R04: Web Interface	High Importance	fully achieved
R05: Easy Integration to existing Products	High Importance	partially achieved
R06: Logging to the cloud	High Importance	fully achieved
R07: Monitoring Data	Medium Importance	partially achieved
R08: Remote Commands	Medium Importance	largely achieved
R09: Statistics	Low Importance	not achieved
R10: Notification System	Low Importance	not achieved
R11: Availability	Low Importance	fully achieved

Table 6.1: Requirements and their NPLF rating

R01: Reliability and Quality

Reliability is one of the most essential properties of an industrial solution. Therefore, the implementation was designed in a fail-safe manner where, when each module fails due to software bugs, connection problems, or any other technical failures, there is always a local database installed on each device containing all data which was sent and is to be sent. Therefore, this requirement was **fully achieved**.

R02: Confidentiality, Integrity, and Authenticity

HTTPS was used for sending data over the Internet as well as within the customer's network. This approach can be considered as secure as long as HTTPS is used correctly, including correctly signed certificates and a certificate revocation process. For this implementation, only self-signed certificates were used which must not be applied in a real-world scenario, and a revocation process was not implemented, resulting in a **partially achieved** requirement. Improvements in security and related future work are discussed in chapter 7.

R03: Restricted access to the World Wide Web

Restricted access to the Internet was the main reason why a self-made *IoT/Cloud* system was necessary. The advantage of this design is the communication which completely takes place within a local network except for the orchestrator-web service data exchange. This makes it more secure against attacks, easier to maintain and set-up regarding the Internet policies, which usually prohibit connections outside the local network at the customer's premises. This requirement was **fully achieved**.

R04: Web Interface

This requirement was **fully achieved**, as a complete website was implemented, offering all features which were primarily needed. An additional

user administration system, which is already used in other solutions of *primtec*, was integrated as well as roles that can be assigned to the users and consequently prohibit access to certain parts of the website.

R05: Easy Integration to existing Products

Logging, heartbeat, and remote commands where each implemented as services that can be started during the boot process of the program. Except for a few configuration parameters listed in the application config files, no additional setup steps are needed. Logs can be created as usual by calling the *NLog* functions, which makes the system entirely invisible to the programmer.

The only part missing is the user credential setup needed on the RabbitMQ side, which can be very time consuming - the requirement was **partially achieved**. More about this topic can be found in Chapter 7.

R06: Logging to the Cloud

Transferring the log messages to the cloud was the main goal of the implementation. In addition to the local database, where each log entry is saved, each log message gets additionally stored in a cloud database. This system works utterly invisible in the background, and no notable configuration is needed resulting in a **fully achieved** requirement.

R07: Monitoring Data

A heartbeat system was integrated, giving each client the opportunity to send heartbeat signals in a specified interval. Based on these heartbeat signals, an online/offline state can be shown on the website. Additional data, including software version, hardware load, or other operating data concerning the client machine, is currently not sent but can be added later by merely implementing an additional service next to the heartbeat-service and the logging-service. This requirement was **partially achieved**.

R08: Remote Commands

Remote commands have been implemented by using *gRPC*. Currently, only a single update-command can be sent containing a file path, which is not processed by the client yet. Due to the fact that *gRPC* is easy to use, the remote commands can be extended without much effort - this requirement was **largely achieved**.

R09: Statistics

Statistics were not implemented because they are outside of the scope of this thesis. However, *Vuetify*, the front-end framework used for the website does feature a framework containing graphs and charts. This requirement was **not achieved**. However, the website is capable, and statistics can be integrated in the future.

R10: Notification System

Since other data, except a heartbeat signal, is not sent, no notification system was implemented, and therefore the requirement was **not achieved**. Nevertheless, ASP.Net, the framework used for the implementation of the web service, does feature an *SMPT-Client* class providing all necessary functionalities for sending Emails.

R11: Availability

Availability is determined by the cloud providers, and since all three of the presented ones have an availability above 90% and additional availability settings, with data redundancy spread over multiple data centers, this requirement was **fully achieved**.

7 Limitations and Future Work

The solution presented in chapter 5 fulfills most of the requirements defined in section 4.3. Some of them, however, could not be met entirely, such as the requirement "Confidentiality, Integrity and Authenticity" (4.3.2) as well as "Easy Integration to existing Products" (4.3.5). This section lists improvements and work which needs to be done in the future to raise the implemented *IoT/Cloud* system to a professional level.

7.1 Security Improvements

The security of the implemented *IoT/Cloud* system relies on TLS (Transport Layer Security), which is used to encrypt the AMQP and MQTT messages and the communication to the web service. As a prerequisite, signed certificates are needed for the clients, the orchestrator as well as the message broker. In the implementation presented in 5, self-signed certificates were used, which must not be applied in a real-world scenario. Therefore a correct certificate should always be used created and maintained by the customer's system administrator.

Due to the risk that a correctly signed certificate, more specifically the private key, could be compromised, a certificate revocation procedure should be implemented. This procedure should invalidate the certificate and block any further message transactions. Since all devices must be equipped with newly generated certificates, and since manual certificate installation is not feasible, an automatic certificate distribution would be needed.

Finally, before deploying these systems at the customer's premises, a professional information security audit and analysis would be advisable, evalu-

ating potential vulnerabilities. This would exceed the scope of this master thesis and should be carried out by an IT security expert.

7.2 Usability Improvements

A *plug and play* experience was mentioned in the requirement description "Easy Integration to existing Products" (4.3.5). This means that after installing the software on the devices and after hosting the message broker and the orchestrator, the *IoT/Cloud* framework should be ready for launch. This has not yet been finally realized in the current expansion stage. Each client needs a specific RabbitMQ user with the correct permissions, which must be predefined and configured using either the RabbitMQ web interface or a command-line tool on the machine hosting the RabbitMQ message broker. This makes the initiation process cumbersome and not scalable for hundreds of clients. Ideally, client users should be created and assigned via the web interface dynamically, making it possible to add new devices later without much effort.

7.3 Scalability Limitations

One key feature and advantage of *IoT* and *Cloud Computing* is its scalability and expandability on demand. Thousands of these small "things" can communicate with each other while a cloud back-end administrates the dataflow. This was also the idea of this master thesis and its implementation; however, its scalability was not tested. It would be advisable to perform a stress test with simulated IoT-clients in order to measure the performance of the system and RabbitMQ.

Additionally, a requirements evaluation concerning the configuration of the cloud database should be made with the aim to configure it as efficient as possible.

7.4 Maturity of the Implementation

The implementation in this thesis represents a proof-of-concept implementation which means that it is not ready to be used at the customer. In addition to the security audit and analysis (7.1), a code review of a *primtec* software developer is needed to meet all industrial requirements. Numerous scientific work on how to perform theses code reviews and detailed analysis regarding its advantages was released in the past (e.g., Thongtanunam et al. (2015) and Czerwonka, Greiler, and Tilford (2015)), and these techniques should be applied.

7.5 Threats to Validity

This work was meticulously prepared and the topics covered were researched as thoroughly as possible. However, errors and gaps can not be excluded since the topics *IoT* and *Cloud* are very extensive. Especially the products of the cloud providers are challenging to research in their seemingly infinite number, which makes it possible that the requirements defined in section 4.3 could be met with one of their premade products. Nevertheless, the solution presented in this thesis as has been demonstrated in the evaluation chapter successfully solves *primtec*'s problems.

8 Conclusion

The goal of this thesis was to introduce the Styrian IT-company *primtec GmbH* to the concepts of *Internet of things* (*IoT*) and *Cloud Computing* by developing and implementing an *IoT/Cloud* framework for their software. This framework should be designed to provide the opportunity for remote controlling, monitoring, and maintenance.

After presenting two products of *primtec* and their current problems, which is the lack of remote access to the worldwide spread machines, the first part of this work focused on the *cloud* and *IoT*, their history, the corresponding terminology, and the fundamental ideas of these novel technologies. *Google Cloud Provider*, *Microsoft Azure*, and *Amazon Web Services*, the three biggest cloud providers, were described and compared regarding their different products, prices, and their general advantages and disadvantages. In consideration of the continually changing and evolving product portfolio and their overall high quality in terms of security and availability, no recommendation could be made.

Since *IoT* is all about the communication between machines, the different "languages" (*MQTT*, *AMQP*, *HTTP*, and *WebSockets*), better known under the generic term *Message Protocols*, which are commonly used in combination with *IoT*, were investigated and compared regarding their relevance for the *primtec* software framework.

In addition, current research and state of the art in the scientific field were presented and explained in these areas.

The second part of this thesis dealt with the practical implementation of the *IoT/Cloud* framework for *primtec*. First, the problems and limitations of the industrial sector, in which almost all *primtec* customers are located, were explained and broken down. From these constraints, requirements

were defined, which had to be fulfilled by the implementation. With these requirements in mind, a solution was developed consisting out of the following parts:

- *The Client Software*: The client software is an addition to the already existing software container, including a "heartbeat-signals service" and "log-message synchronization service" using *MQTT* and *AMQP*.
- *The Orchestrator*: This module receives the locally sent messages by the clients in the network and forwards them to a cloud database.
- *The Message Broker*: A message broker (*RabbitMQ*) was used to handle the *MQTT*/*AMQP* communication within the network.
- *The Cloud Database*: A *PostgreSQL* database hosted using the *Google Cloud Provider*, which was chosen for the sole reason of the lowest cost at that time, was used to store the log messages.
- *The Web Service*: A website based on *Vue* was implemented, showing the currently active devices and their corresponding log-messages and online states.

The entire implementation can be considered a proof-of-concept and is not yet ready to be deployed at the customer's site due to security reasons and various simplifications and features that still need to be implemented. These extensions were discussed in the final part of this thesis, and an outlook on future work was given.

Bibliography

Albalawi, A. et al. (2019). "A Survey on Authentication Techniques for the Internet of Things." In: *2019 International Conference on Computer and Information Sciences (ICCIS)*, pp. 1–5 (cit. on p. 26).

Ashton, Kevin et al. (2009). "That 'internet of things' thing." In: *RFID journal* 22.7, pp. 97–114 (cit. on p. 24).

Azure, Microsoft (2020). *SLA for Azure SQL Database*. Microsoft Azure. URL: https://azure.microsoft.com/en-us/support/legal/sla/sql-database/v1_4/ (visited on 04/17/2020) (cit. on p. 38).

Babovic, Z. B., J. Protic, and V. Milutinovic (2016). "Web Performance Evaluation for Internet of Things Applications." In: *IEEE Access* 4, pp. 6974–6992 (cit. on pp. 22, 27).

Buyya, R., C. S. Yeo, and S. Venugopal (2008). "Market-Oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities." In: *2008 10th IEEE International Conference on High Performance Computing and Communications*, pp. 5–13 (cit. on p. 27).

Chopra, K., K. Gupta, and A. Lambora (Feb. 2019). "Future Internet: The Internet of Things-A Literature Review." In: *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*, pp. 135–139. DOI: 10.1109/COMITCon.2019.8862269 (cit. on p. 6).

Chopra, Kriti, Kunal Gupta, and Annu Lambora (Feb. 2019). "Future Internet: The Internet of Things-A Literature Review." In: pp. 135–139. DOI: 10.1109/COMITCon.2019.8862269 (cit. on p. 25).

Czerwonka, J., M. Greiler, and J. Tilford (2015). "Code Reviews Do Not Find Bugs. How the Current Code Review Best Practice Slows Us Down." In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 2, pp. 27–28 (cit. on p. 65).

Dordevic, Borislav, Slobodan Jovanovic, and Valentina Timcenko (Nov. 2014). "Cloud Computing in Amazon and Microsoft Azure platforms: Performance and service comparison." In: pp. 931–934. DOI: 10.1109/TELFOR.2014.7034558 (cit. on p. 26).

Dorsemaine, B. et al. (Sept. 2015). "Internet of Things: A Definition amp; Taxonomy." In: *2015 9th International Conference on Next Generation Mobile Applications, Services and Technologies*, pp. 72–77. DOI: 10.1109/NGMAST.2015.71 (cit. on pp. 7, 8, 25).

Edan, N. M., A. Al-Sherbaz, and S. Turner (2017). "Design and evaluation of browser-to-browser video conferencing in WebRTC." In: *2017 Global Information Infrastructure and Networking Symposium (GIIS)*, pp. 75–78 (cit. on p. 18).

Fatemi Moghaddam, F. et al. (Aug. 2015). "Cloud computing: Vision, architecture and Characteristics." In: *2015 IEEE 6th Control and System Graduate Research Colloquium (ICSGRC)*, pp. 1–6. DOI: 10.1109/ICSGRC.2015.7412454 (cit. on p. 8).

Ghaffar, M. A. A. and T. T. Vu (2015). "Cloud computing providers for satellite image processing service: A comparative study." In: *2015 International Conference on Space Science and Communication (IconSpace)*, pp. 61–64 (cit. on p. 26).

Gigli, Matthew and Simon Koo (Jan. 2011). "Internet of Things: Services and Applications Categorization Abstract." In: *Adv. Internet of Things* 1, pp. 27–31. DOI: 10.4236/ait.2011.12004 (cit. on p. 25).

Gou, Q. et al. (2013). "Construction and Strategies in IoT Security System." In: *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*, pp. 1129–1132 (cit. on p. 26).

Guner, A., K. Kurtel, and U. Celikkan (2017). "A message broker based architecture for context aware IoT application development." In: *2017 International Conference on Computer Science and Engineering (UBMK)*, pp. 233–238 (cit. on pp. 17, 28, 40, 45).

Hoffmann, J. et al. (2018). "Towards a Safety and Energy Aware protocol for Wireless Communication." In: *2018 13th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pp. 1–6 (cit. on p. 29).

IBM (2020). *How TLS provides identification, authentication, confidentiality, and integrity*. IBM. URL: https://www.ibm.com/support/knowledgecenter/

SSFKSJ_9.0.0/com.ibm.mq.sec.doc/q009940_.html (visited on 06/18/2020) (cit. on p. 48).

IBM-Cloud-Education (2020). *Message Brokers*. IBM. URL: https://www.ibm.com/cloud/learn/message-brokers (visited on 07/04/2020) (cit. on p. 20).

John, Vineet and Xia Liu (2017). "A Survey of Distributed Message Broker Queues." In: *ArXiv* abs/1704.00411 (cit. on p. 23).

Kang, D. et al. (2017). "Room Temperature Control and Fire Alarm/Suppression IoT Service Using MQTT on AWS." In: *2017 International Conference on Platform Technology and Service (PlatCon)*, pp. 1–5 (cit. on p. 28).

Kaskade, Jim (2020). *What Does HA in the Cloud Mean?* Jim Kaskade. URL: https://jameskaskade.com/?p=448 (visited on 07/04/2020) (cit. on p. 38).

Kumar, K. and Y. Lu (2010). "Cloud Computing for Mobile Users: Can Offloading Computation Save Energy?" In: *Computer* 43.4, pp. 51–56 (cit. on p. 26).

Li, Shancang, Li Xu, and Shanshan Zhao (Apr. 2014). "The internet of things: A survey." In: *Information Systems Frontiers* 17. DOI: 10.1007/s10796-014-9492-7 (cit. on pp. 7, 24).

Luzuriaga, J. E. et al. (2015). "A comparative evaluation of AMQP and MQTT protocols over unstable and mobile networks." In: *2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC)*, pp. 931–936 (cit. on p. 28).

Madakam, Somayya, R Ramaswamy, and Siddharth Tripathi (Apr. 2015). "Internet of Things (IoT): A Literature Review." In: *Journal of Computer and Communications* 3, pp. 164–173. DOI: 10.4236/jcc.2015.35021 (cit. on pp. 24, 25).

MarketWatch (2020). *Amazon rules the public cloud*. MarketWatch, Inc. URL: https://www.marketwatch.com/story/amazon-rules-the-public-cloud-but-google-microsoft-alibaba-are-growing-faster-2017-12-20 (visited on 02/21/2020) (cit. on p. 19).

Mell, Peter and Tim Grance (2011). *The nist definition of cloud computing*. NIST. URL: https://csrc.nist.gov/publications/detail/sp/800-145/final (visited on 01/03/2020) (cit. on pp. 9–11).

Mesquida, Antoni Lluís, Antònia Mas, and Esperança Amengual (2011). "An ISO/IEC 15504 Security Extension." In: *Software Process Improvement and Capability Determination*. Ed. by Rory V. O'Connor et al. Berlin,

Heidelberg: Springer Berlin Heidelberg, pp. 64–72. ISBN: 978-3-642-21233-8 (cit. on p. 59).

Naik, N. (Oct. 2017). "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP." In: *2017 IEEE International Systems Engineering Symposium (ISSE)*, pp. 1–7. DOI: 10.1109/SysEng.2017.8088251 (cit. on pp. 12, 14).

OASIS-Open (2020). *MQTT Version 3.1.1*. OASIS Open. URL: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html (visited on 04/23/2020) (cit. on p. 13).

Panetta, Kasey (2020). *Gartner Top 10 Strategic Technology Trends for 2018*. Gartner. URL: https://www.gartner.com/smarterwithgartner/gartner-top-10-strategic-technology-trends-for-2018/ (visited on 02/07/2020) (cit. on p. 8).

ParkMyCloud (2020). *AWS vs Azure vs Google Cloud Market Share 2020: What the Latest Data Shows*. ParkMyCloud. URL: https://www.parkmycloud.com/blog/aws-vs-azure-vs-google-cloud-market-share/ (visited on 04/02/2020) (cit. on pp. 19, 20).

Peniak, P. and M. Franeková (2018). "Extended Model of Secure Communication for Embedded Systems with IoT and MQTT." In: *2018 International Conference on Applied Electronics (AE)*, pp. 1–4 (cit. on p. 28).

Prajapati, A. G., S. J. Sharma, and V. S. Badgujar (2018). "All About Cloud: A Systematic Survey." In: *2018 International Conference on Smart City and Emerging Technology (ICSCET)*, pp. 1–6 (cit. on p. 26).

primtec-IoT (2020). *primtec IoT Device*. primtec GmbH. URL: https://primtec.eu/produkte/iot-device/ (visited on 02/07/2020) (cit. on p. 2).

primtec-PackingTable (2020). *primtec Packing Table*. primtec GmbH. URL: https://primtec.eu/produkte/rfid-arbeitstisch/ (visited on 02/07/2020) (cit. on p. 4).

RabbitMQ (2020). *AMQP 0-9-1 Model Explained*. RabbitMQ. URL: https://www.rabbitmq.com/tutorials/amqp-concepts.html (visited on 06/26/2020) (cit. on p. 15).

Rimal, B. P., E. Choi, and I. Lumb (2009). "A Taxonomy and Survey of Cloud Computing Systems." In: *2009 Fifth International Joint Conference on INC, IMS and IDC*, pp. 44–51 (cit. on p. 26).

Saadeh, M. et al. (2016). "Authentication Techniques for the Internet of Things: A Survey." In: *2016 Cybersecurity and Cyberforensics Conference (CCC)*, pp. 28–34 (cit. on p. 25).

Sachs, Kai et al. (Apr. 2010). "Benchmarking Publish/Subscribe-Based Messaging Systems." In: vol. 6193, pp. 203–214. DOI: 10.1007/978-3-642-14589-6_21 (cit. on p. 21).

Singh, J. et al. (2016). "Twenty Security Considerations for Cloud-Supported Internet of Things." In: *IEEE Internet of Things Journal* 3.3, pp. 269–284 (cit. on p. 26).

Sreeraj, S., N. S. Kumar, and G. S. Kumar (2017). "A framework for predicting the performance of IoT protocols, a use case based approach." In: *2017 International Conference On Smart Technologies For Smart Nation (SmartTechCon)*, pp. 577–580 (cit. on p. 28).

Thongtanunam, P. et al. (2015). "Who should review my code? A file location-based code-reviewer recommendation approach for Modern Code Review." In: *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 141–150 (cit. on p. 65).

Uy, N. Q. and V. H. Nam (2019). "A comparison of AMQP and MQTT protocols for Internet of Things." In: *2019 6th NAFOSTED Conference on Information and Computer Science (NICS)*, pp. 292–297 (cit. on pp. 13, 14, 16).

Wikipedia (2020). *MQTT*. Wikipedia. URL: https://en.wikipedia.org/wiki/MQTT (visited on 04/02/2020) (cit. on p. 14).

Yassein, M. B., M. Q. Shatnawi, and D. Al-zoubi (2016). "Application layer protocols for the Internet of Things: A survey." In: *2016 International Conference on Engineering MIS (ICEMIS)*, pp. 1–4 (cit. on p. 27).

Zhang, Minghui, Fuqun Sun, and Xu Cheng (2012). "Architecture of Internet of Things and Its Key Technology Integration Based-On RFID." In: *2012 Fifth International Symposium on Computational Intelligence and Design* 1, pp. 294–297 (cit. on p. 25).

Zhang, Z. and J. S. Lim (2015). "Emotion Recognition Algorithm Based on Neural Fuzzy Network and the Cloud Technology." In: *2015 10th International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA)*, pp. 576–579 (cit. on p. 26).

Zhonggui Ma et al. (2013). "The architecture and key technologies of Internet of Things in logistics." In: *International Conference on Cyberspace Technology (CCT 2013)*, pp. 464–468 (cit. on p. 25).