Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Hauke Buhr

A proof of concept for an interoperable IoT platform

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Hauke Buhr

A proof of concept for an interoperable IoT platform

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Becke
Zweitgutachter: Prof. Dr. Lehmann

Eingereicht am: 09, November 2017

Hauke Buhr

Thema der Arbeit

A proof of concept for an interoperable IoT platform

Stichworte

Internet der Dinge (IoT), Interoperable IoT Plattform, Objekt-Beschreibungssprache, Verteilte Plattform, OMNet++, Nachrichtendesign, Kommunikationsdesign

Kurzzusammenfassung

Das Internet der Dinge (IoT) ist im Moment ein schnell wachsendes Themengebiet. Es findet Anwendung in vielen verschiedenen Bereichen wie zum Beispiel dem Smart Home, der Smart City, dem Gesundheitswesen [1].

Analysten prognostizieren, dass 50 Milliarden IoT Geräte im Jahr 2022 im Umlauf sein werden [2]. Es ist zu erwarten, dass diese große Anzahl von Geräten erzeugt eine sehr große Menge an Daten. Die generierten Daten müssen daher effizient verwaltet werden, um die Daten in angemessener Zeit verarbeiten und übertragen zu können.

Gegenwärtige Open-Source IoT Plattformen sind vertikal entworfen und decken die IoT Geräte Hersteller SDKs sowie auch Datenauswertungs- und Darstellungsdiensten ab. Indes decken gegenwärtige Plattformen nicht die Interoperabilität zwischen der eigenen und anderen Plattformen ab. Aus der fehlenden Interoperabilität kann gefolgert werden, dass die Daten nur innerhalb der eigenen Plattform gespeichert und verwertbar sind. Der Zugriff auf die Daten wird so jedoch nicht problemlos von außerhalb möglich.

Es gibt auch IoT Plattformen, die einen Ansatz zur Interoperabilität enthalten. Doch meist konzentrieren sich die interoperablen Plattformen nur darauf, die Kommunikation von IoT Geräten untereinander zu gewährleisten.

Diese Bachelorthesis schlägt ein interoperables IoT Plattformkonzept vor, welches durch sein Design die Eigenschaft der Interoperabilität unterstützt. Das Konzept besteht aus zwei Strukturelementen, zum einen die Kernkomponenten und zum anderen die „Connected Services“. Die Kernkomponenten sind die Management-, Registry- und Gatewaykomponente. Außerdem werden mehrere Interfaces für eine einheitliche Schnittstellenbeschreibung eingeführt. „Connected Services“ werden im Folgenden als „Objekte“ [1] bezeichnet. Diese Objekte sind „Services“[2] auf einer höheren Ebene, die Daten verarbeiten, speichern und, oder diese präsentieren.

[1] Objekte nicht im Kontext einer objektorientierten Programmiersprache.
[2] Services als angebotene Dienste, die Daten von Objekten nutzen.

Die in dieser Thesis vorgestellte Registry kann ihre Einträge mit einer anderen Registry teilen. Diese verteilten Registryeinträge und die angestrebte Plattform führen zu einer Hierarchie der IoT Plattformen, deren Daten auf jeder freigegebenen Ebene verfügbar sind. Dieser Ansatz fördert somit einem heterogenen und offenen Plattformdesign.

Außerdem braucht eine interoperable Plattform eine Objektsuchfunktion. Die vorgestellte Suchfunktion benötigt eine generische Beschreibungssprache. Die in dieser Thesis vorgestellte Beschreibungssprache ist objektorientiert.

Zu dem werden in dieser Thesis die Ergebnisse aus einer Proof of Concept Simulation in OMNeT++ [3] bereitgestellt und ausgewertet. Es zeigt sich, dass die Komplexität der Objektbeschreibung minimiert wird, da nur wenige Merkmale beschrieben werden müssen. Jedoch besteht durch den erweiterbaren Aufbau der Objektbeschreibung eine effektivere Möglichkeit der Beschreibung.

Auch die vorgeschlagenen Kommunikationsmuster in dieser Thesis sollen das Nachrichtenaufkommen reduzieren und eine Priorisierung von Nachrichten ermöglichen [2].

Teil dieser Bachelorthesis ist die Analyse von Anforderungen an eine IoT Plattform, mit anschließender Analyse einiger bestehender Plattformen. Des Weiteren werden Kommunikationsmuster definiert und die Objektbeschreibungssprache wird designt. Am Ende wird ein Entwurf der Plattform angeboten und mit OMNet++ demonstriert und evaluiert.

Hauke Buhr

Title of the paper

A proof of concept for an interoperable IoT platform

Keywords

Internet of Things (IoT), Interoperable IoT Platform, Object-Description Language, Distributed Platform, OMNet++, Message-Design, Communication-Design

Abstract

The Internet of Things (IoT) is currently a fast developing subject. It takes place in areas like smart home, smart city, health care and many more [1].

Analysts predict an amount of 50 billion IoT devices by 2022[2]. This tremendous amount of devices is generating even bigger amounts of data then the ones today. The produced data has to be processed in an efficient way, to be able to compute the data in a proper time.

Current open source IoT platforms are vertical designed and cover everything from sensor platform SDKs up to data evaluation and presentation services. But current IoT platforms do not cover interoperability between itself and any other platform. This means the data is stored and accessible only from within the system. Thus, there is no convenient way to access the data from a different platform. There are also approaches heading to an interoperable platform design. However, they mostly only focus on platform intercommunication and not on an interoperable discovery of IoT devices (further also referred to as objects) and data.

This bachelor thesis proposes an IoT platform concept which is interoperable by design. The concept consists of two types of components, first the core components and second the connected services. The proposed core components are the management-, registry- and gateway component. Also several interface specifications are introduced, as displayed in Figure 6.1. Connected services are here referred to as objects, upper level services like data processing and data bases or presenting services. Furthermore registries can share their dictionaries with other registries. Shared dictionaries and the resulting platform are leading to a stacked IoT platform that can be accessed at any level. This approach counteracts a closed and monolithic platform design.

Furthermore an interoperable platform needs an object discovery functionality. The proposed discovery mechanism needs a generic description scheme, that proposed description scheme is an object oriented description scheme. Moreover, this thesis discusses the experimental results of a proof of concept simulation in OMNeT++ [3]. Also this thesis describes the results which were made with a proof of concept simulation in OMNeT++. The object description scheme uses some required description fields and the descriptions can be adapted by some optional descriptions. The communication pattern, that are proposed, reduce the messages [2] and use different priorities for data processing [2].

The first step of this thesis is to define some requirements for an interoperable IoT platform and afterwards some existing platforms are compared against the requirements. The second step is to design the communication patterns and the object description scheme is designed. The third step is to design the platform, to implement a simulation in OMNet++ and to evaluate the simulation results.

Contents

List of Tables

List of Figures

Listings

1. Introduction

The motivation of this thesis, to develop a platform that is truly interoperable and an object context is provided for all parties, is provided in this Chapter. Furthermore the role of IoT platform is discussed and requirements for an IoT platform are presented.

This Chapter provides the motivation of this thesis, to develop a platform that is truly interoperable and provides object context for using parties. Furthermore the role of IoT platform is discussed and requirements for an IoT platform are presented.

1.1. Motivation

The Internet of Things (IoT) is an important and currently fast developing topic in both technology and engineering perspectives. There is a vast amount of different definitions what the IoT is, which work is done by it and which benefits are provided by it. The broad consensus is, that the IoT consists of objects, which are fitted with an Internet connection and their dedicated purpose is fulfilled with enough computing power. Communication, Interaction and Coordination between different objects is done without human intervention. A unique identifier is generated and assigned to each object [4]. The added value, generated for all participants, is the core improvement done by the IoT. The added value is a benefit for companies and customers, either a financial one or the efficiency is increased.

Build on this consensus many different kinds of IoT platforms were developed. A specific purpose is served by each of these platforms. These different approaches are further discussed in Chapter 2.

The idea of an IoT is not new. Earlier the devices were limited by their size and computing power. With these limitations an IoT was not feasible. Nowadays it is feasible due to the fact, that electronics became minimized, have a better energy consumption and the computing power is been enhanced [5]. An early adopter of the IoT-idea was the IP-enabled toaster which could be controlled over the Internet. That toaster was featured in 1990 at an Internet conference [6].

The next question is how many devices will be used in the future. Many companies and research organizations have offered their projections. A massive rise in the use of IoT devices is projected by different companies and organizations. A projection of 24 billion inter-connected IoT objects in the year 2019 was announced by Cisco [7] and 75 billion networked devices overall by 2020 are projected by Morgan Stanley [8]. In the more distant future the numbers are massively increasing. An amount of 50 billion IoT devices by 2022 is predicted by analysts [2], 100 billion IoT connections by 2025 are projected by Huawei [9] and an overall IoT revenue of $3.9 to $11.1 trillion by 2025 is suggested by McKinsey Global Institute [10]. The predicted numbers show a big variation, which makes every specific number questionable. So every specific number is questionable. A significant growth of the IoT itself can be seen if these numbers are analyzed and evaluated together.

Tremendous amounts of data and traffic are generated by huge numbers of IoT devices. The produced data has to be processed in an efficient way, so that data can be computed in a proper time. Therefore scaling structures are needed.

1.1.1. Use Cases

Several different areas for the IoT can be found on Site [1]. Some examples for areas are smart home, smart city and health care. The reason for the projected fast growth can be found in the huge range of possible areas. Hereinafter some IoT-use cases are shown.

1.1.1.1. Smart Home

The area smart home is the first meeting point for consumers who are interested in IoT devices and corresponding services. Consumers are affected by the IoT in areas like home automation components, health care, entertainment and energy management. A higher comfort at home could be reached by smart home systems. Furthermore a home with higher security standards and better energy consumption could be accomplished [5].

A practical example for a smart home use case is music streaming. An audio device could be used by someone in the room to stream music. The user is connected to the IoT system and is now able to stream the music via a connected device, that is capable of music streaming.
A user presence monitoring system is another example for an IoT system. All installed devices could be turned off to save energy, if they do not have to operate at this time.

1.1.1.2. Smart City

Todays cities are frequently congested by cars and commuters. Smart cities are a possibility to help minimize these problems. Networked vehicles, intelligent traffic systems and sensors embedded in roads and bridges are examples for sensing and interacting interfaces of a smart city [5].

A city is frequently congested by cars or people and traffic lights are a limiting factor for these groups to move effectively. With a monitoring system for cars and people, the traffic could be controlled in a more efficient way and the cities could become less congested.

To monitor the availability of parking space is another possible use case for a smart city system. In such scenarios a free parking lot could be searched and perhaps reserved for a specific amount of time. The time that is needed to search for a free parking lot could be reduced and the issue of congested streets could be reduced.

1.1.2. IoT platform

An IP-communication is installed in most devices today or they are connected via an IP-gateway. The question is, how advantages of devices like the IP-capable toaster could be used to create better services for the users. Philips Hue [11] for example, is a proprietary platform for IoT devices, built by the vendor. Proprietary platforms are created by each vendor to support their own products. The results are many different and separated proprietary platforms. Devices from different vendors can be controlled only by the usage of several different platforms. The alternative is, to use devices from one single vendor only.

To overcome this situation, IoT platforms were developed and are still being developed. The connectivity between IoT devices and users is established by IoT platforms. The upcoming traffic is handled and the communication between all participants is guaranteed by these platform. The platform can be accessed in different ways by the participants. Software Development Kits (SDKs) are delivered by these platforms. The platform can be used by different devices with the specified SDKs. Individual credentials are created to connect each device to the platform. Different devices are combined by these platforms, so they can be used in one environment. Kaa is a good example for such an open source platform and is further discussed in Section 2.1.2.

Several benefits like security and scalability are offered by platforms like Kaa. Interoperability is missed by all previously discussed platforms. That means the processed data is accessible inside the platform, but not from the outside and intercommunication between devices of different platforms is not possible in a convenient way.

Interoperability is the core value and a corner stone of the current Internet. Current platforms are built like so called "walled gardens" by most vendors today. It is impossible with todays platforms, to share data in a convenient way with each other. Users are forced to choose a platform and stick with this specific system. The transition to another system is made difficult by the vendors which have no intentions to open their "walls" to ease the transition to another system.

Devices and services from different platforms can be used by each other as desired, if a fully interoperable IoT environment is established. To establish an interoperability over all devices is not feasible or even necessary in every case, but the advantages and benefits for many use cases are increased by interoperability. Benefits can only be generated when the interoperability is well defined. Devices and services are defined by a common and well known description scheme.

1.2. Requirements

In the following Section requirements for an IoT platform are proposed and defined. These requirements should be discussed and are used for further IoT platform debates.

Interoperability Nowadays IoT devices are developed and manufactured by several different vendors. These IoT devices are normally connected to the vendors own platform. The users are enabled by these platforms to monitor and control connected devices of that specific vendor.

An interoperable platform is needed to counteract the segmentation of the IoT. Devices from different vendors should be able to be used within a single platform. By that, users are able to use devices from different vendors in their own system. Furthermore, the devices should be able to interact with each other. Machine readable and interpretable descriptions of services and devices should be provided or supported by that platform. These descriptions should be well-defined, so they could be read by different consumers with different systems. Also the descriptions should be discoverable, so specific services

and devices could be found within different platforms. Open standards should be used by an open and interoperable platform and proprietary protocols should therefore being avoided.

Independence Some platforms are built as a interoperability frameworks. The interoperability is added, to compatible existing platforms which are not interoperable, by these frameworks. A unified environment is established by these frameworks but it is not possible to connect devices directly to these frameworks. A core platform is needed to overcome the need for a platform to connect a device.

Any IP-capable device should be able to establish a connection to the platform. In order to do that the platform should be independent and ready to use without other platforms.

Message Reduction An estimation of high amounts of IoT devices was published. An even higher message amount will be produced by these devices. The networks could be under high pressure by the estimated amount of messages.

In knowledge of estimated IoT device numbers, the platform should be scalable. Efficient communication patterns should be used to reduce the messages by design. Further strategies should be established to reduce the send messages even more.

Dynamic Nowadays a IoT device is subjected to evolving requirements. The requirements are influenced by mutating security, performance and function requirements.

To overcome that issue updates for devices should be done at runtime. The function of the device should not be subjected by that. Dynamic mechanisms should be offered by the platform to fix issues, edit and change the purpose of the devices and services.

Administration The IoT lives from a great distribution. People with different kinds of technology knowledge are forced to use these systems. These systems have to be set up and maintained while being operated.

An IoT platform should be easy to use for everyone. Furthermore these kinds of platforms should be easy to set up and maintained.

1.3. Goals

The goal of this thesis is to present an interoperable IoT platform design, that is simulated in order to check the functionality. The design is evaluated with an OMNet++ simulation and the results are discussed. The object description language is a core feature of the proposed

interoperable IoT platform. This description language is empirically examined and the results are discussed.

1.4. Organization

To achieve these goals this thesis is organized as follows:

Chapter 2 - Related Work - the state of the art related work is described and some IoT platforms are presented.

Chapter 3 - Overview - the proposed interoperable IoT platform is introduced. The measures that are used to match the requirements are introduced, a short overview on behalf of security issues is given and the proposed communication patterns are introduced.

Chapter 4 - Description Language - describes how an object is represented in a description and some object descriptions are discussed.

Chapter 5 - Rules - introduces the rules in general. Furthermore, it is described how to work with rules to generate usage benefits, how the agent runtime environment can be used and usage examples of both cases are discussed.

Chapter 6 - Interoperable Platform Design - the created IoT platform design is discussed and the associated key components are presented. Also, the proposed IoT platform is classified.

Chapter 7 - Communication Sequences - It is defined how the defined components are communicating with each other. Furthermore, it is shown, which messages are used.

Chapter 8 - Platform Simulation - the simulations that are made with OMNet++ are described and the test results are discussed.

Chapter 9 - Conclusion - The research findings are summarized and an outlook for future work is provided.

2. Related Work

In the following Sections an overview of the current state of the art is presented. Also a small range of current IoT platforms is shown and their purpose is discussed. Furthermore the presented platforms are measured according to the previously defined requirements.

2.1. Current State of the Art

Currently the market is filled with different systems, that provide several services for different use cases and orientations. Hereinafter some platforms and their points of interest are described. Also the platforms are measured according to the previously described requirements (Section 1.2) and are briefly discussed. A more comparing discussion is presented in Section 2.2. The Sections are presented in the following steps. Some proprietary platforms are described in the first step. In the second step an open source variant is being explained and two research platforms are described in the third step.

2.1.1. Proprietary

IoT platforms are offered by many companies, some of them a part of them are being discussed in this Section. A specific use-case or environment is most likely to be focused by a company. Some example platforms and their specific orientations are shown in the following Sections.

2.1.1.1. Amazon

Different devices are connected to the cloud by Amazon's "AWS IoT" cloud-platform [12]. All components from device SDKs up to management tools, evaluation tools and the needed infrastructure are delivered by Amazon. This is shown in Figure 2.1.

On Amazon's consumer website the AWS IoT platform is shown as a consumer platform, but the infrastructure is focused on company sized platforms. Different technologies are used to exchange data between the devices and the platform. For example MQTT [13], WebSockets [14] and HTTP/1.1 [15] are used to establish and maintain connections. The platform is built as a monolithic and scaling system. One billion devices can be handled with the platform.

Furthermore secured and reliable data transport is provided by the platform.

Some basic IoT devices are provided by Amazon, one example is a simple button [16]. The button is used to trigger different events. Events are for example to order a cab, open the garage door or even control Philips Hue bulbs [11].

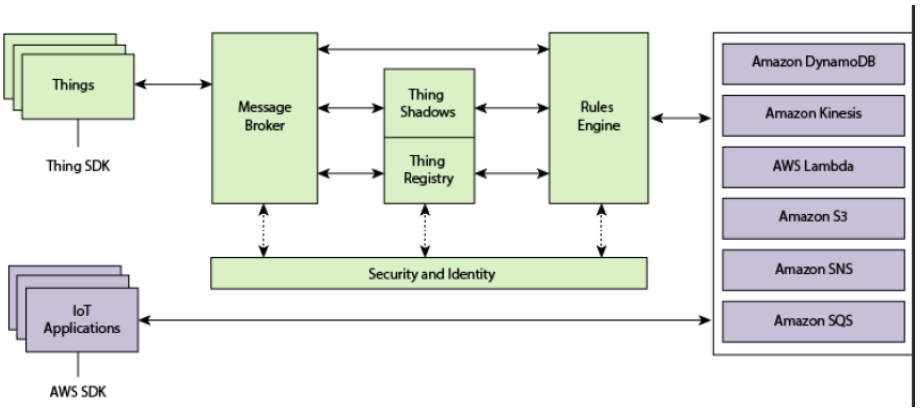

Figure 2.1.: Amazon AWS IoT core architecture from [17]

The Amazon AWS IoT platform is not interoperable. Inside traffic is supported by the platform but it has no interfaces for inter-platform data exchanges. Although meta-data is used to describe delivered data and device capabilities. These meta-data is only useful and usable inside the platform.

The platform is independent and operates solely. Furthermore the platform is connected to other Amazon AWS services.

The generated traffic is transported via the platform. A direct machine-to-machine communication is not featured. Some benefits are added by the centralized traffic handling. The traffic is monitored and added instances can handle traffic peeks. The data is protected against internal loss by centralized monitoring mechanisms. Messages can be reduced by the usage of rules. A behavior is defined by the rules and they are executed in a component that is called rule engine. Consequential the messages from the users to the devices are reduced.

A dynamic behavior of the devices is not supported by the platform. SDKs are used and the devices behavior is previously defined. If a new behavior is required the devices need to be updated manually.

The Amazon AWS IoT platform is hosted by Amazon, but the platform needs to be managed manually. Which means for example, new devices have to be added and registered devices have to be managed. [18] [19]

2.1.1.2. Microsoft

The Azure IoT Suite of Microsoft is focused on many use cases. Many of these use cases are part of the so called "Industry 4.0". But there are also use cases like connected cars and intelligent buildings [20].

In general, a connected plant is used to enhance the productivity and the profitability. The OPC Unified Architecture [21] standard is used to connect and monitor sites and devices. Due to this the fabrication line can be analyzed and the performance and efficiency can be increased. Furthermore it can be used to monitor the devices and to make remote analysis. The efficiency can be increased while the costs of the plant are lowered [22].

The Azure IoT Suite is a closed platform which is composed of different elements. The key components are the "IoT Hub" and the "Event Hub". A bidirectional connection with all registered devices is established by the "IoT Hub" [23] and the delivered data is processed by the "Event Hub" [24].

Microsoft is focused on data analysis and processing. Therefore an added value can be generated from the collected data.

Microsotf's Azure IoT Suite is not interoperable. It is a closed monolithic system which is built to process the data efficiently within the platform. Internally meta-data is used to describe devices and their properties. Third-party devices can be connected to the platform by different kinds of gateways [25].

The platform can be used by devices that are programmed with the delivered SDK. Many of Microsoft's own software products are used. One example is Cortana and Microsoft's machine learning system.

The traffic is channeled through the platform and a direct communication between devices is not allowed. So the efficiency is reduced in favor of higher reliability.

Moreover agents are used to add dynamic, but limited, capabilities to their devices [25]. Another dynamic capability or usage of the devices is not mentioned.

The platform is hosted by Microsoft in combination with their complete Azure software products. The IoT platform has to be managed manually. New devices are added and registered devices are maintained by the user.

2.1.1.3. SAP

The focus of SAP's IoT solution is pointed on companies which align themselves with the term "Industry 4.0". A system, that processes orders and theoretically manufactures the ordered goods automatically, is offered by them. In order with their SAP ERP system [26] it is not only

used to plan and manufacture the products, but also the storages are updated automatically and resources are tracked within the processes.

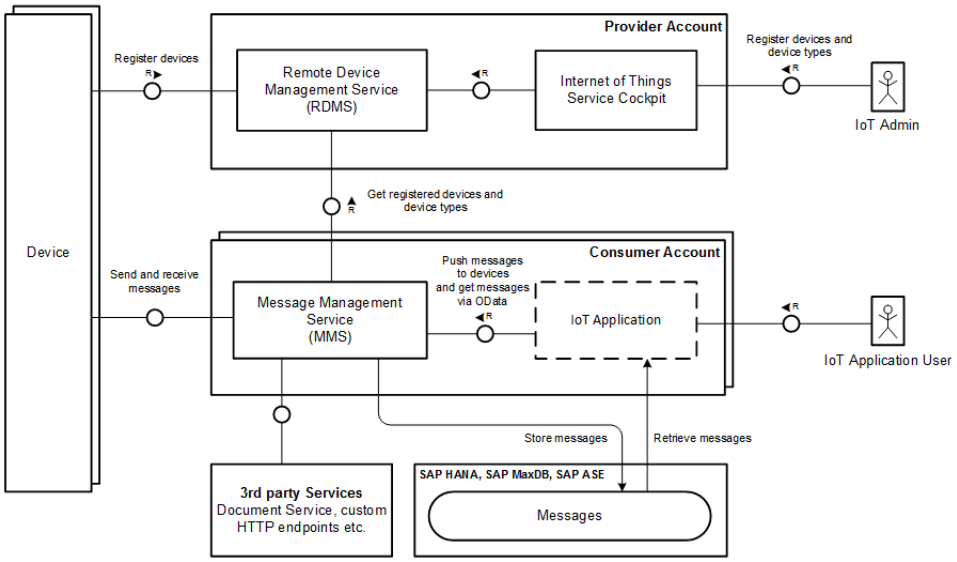

Figure 2.2.: SAP HANA IoT architecture from [27]

The architecture is shown in Figure 2.2 and third party service interfaces are defined. That interface is used to connect third party evaluation tools. A basic device description scheme is used internally. Furthermore, the possibility to define messages is given by the system.

The platform is strongly coupled to SAP's other systems and can be operated alone. However, third party access is also allowed by them.

All messages are delivered and received by the platform itself. A direct connection between devices is not mentioned, so a high-level control and monitor tool has to keep track and coordinate the system like it is used within the automatic manufacturing site.

A dynamic behavior of the devices is not mentioned. So in conclusion all the devices are connected to the platform via the SDK made available by SAP.

The IoT platform is a part of the HANA project and is hosted by SAP. The Internet of Things Service Cockpit (Section 2.2) is used to manual manage the platform. The IoT Application interface is used by the users.

2.1.2. Open Source

Kaa is an open source community project and it is basically a back-end machine-to-machine communication platform, where within some basic data presentation schemes are included. It

is a company grade IoT platform for private and project usage. The goal of Kaa is to provide a platform that is usable for all use cases which need machine-to-machine communication. Several usage examples are provided by the community.

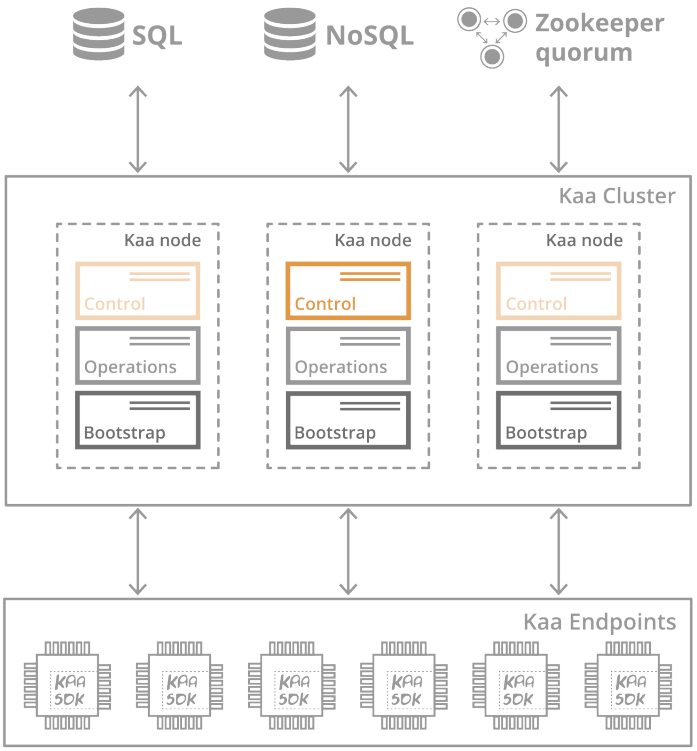

Figure 2.3.: Kaa cluster architecture from [28]

The data is processed internally by the closed system. Also the data is represented by a basic description scheme and is processed internally. The platform can only be accessed on the device level or at the application level. The system is not built to share information with other platforms easily.

The Kaa IoT platform works in an autonomic way. Other systems such as Apache Zookeeper [29] or SQL-databases are used by the system. Furthermore it is considered as a back-end solution.

The complete traffic is processed by the platform. A direct device to device communication is not supported. So it is a centralized and monolithic system.

Dynamic capabilities can not be found within the devices. Delivered SDKs are used to access and operate with the platform and changes of devices need to be made manually.

The platform is hosted and managed manually by the user. A system that is ready to use is delivered by Kaa. Eased first steps are promoted by the Sandbox that comes configured. A Kaa cloud solution [30] is currently in progress.

2.1.3. Research

Research projects called OpenIoT and BIG IoT are founded by the EU. OpenIoT was built to generate basic knowledge about interoperability and its benefits. BIG IoT is built upon the generated knowledge of OpenIoT. BIG IoT is a interoperability framework which is a marketplace for different IoT platforms. In the following Sections these platforms are further discussed.

2.1.3.1. OpenIoT

OpenIoT is an open source middleware with the purpose to collect information from sensor clouds, without having to worry about which exact sensor is used. Basic values can be measured by sensors or sensor systems. Vocabularies are used to describe specific values and units. Efficient ways to use and manage cloud environments for IoT devices and services are explored by OpenIoT. “Sensing-as-a-Service" is the concept that was chosen by OpenIoT. The platform is specialized to collect sensor data, but actuators are not part of the system [31].

Semantically annotated sensor data is provided by OpenIoT. The W3C Semantic Sensor Networks (SSN) [32] specification is used to semantically annotate the data. This annotation is standardized and is consequential machine understandable. Different vocabularies are used in the annotation scheme in order to support more data types [33].

The Global Sensor Networks (GSN) [34] is used to collect sensor data. Apart from that, the platform is independent.

The communication pattern poll and push are used by the platform, but the efficiency could significantly be increased, if the patterns are used properly [35].

The data is provided by several connected GSNs. The GSNs are only represented by their description. So if the description of the GSN is edited, the described data is used by the platform without further manual interaction.

The platform has to be hosted and managed manually. Furthermore, a Sandbox system like Kaa’s is offered.

2.1.3.2. BIG IoT

BIG IoT is the short form of Bridging the Interoperability Gap of the Internet of Things. It is a specific interoperable platform for IoT platforms and services. "BIG IoT API" and "BIG IoT Marketplace" are the two basic components of the platform. It is founded by the EU and developed in corporation with several companies like Bosch, Seat and Siemens [36].

Sensor data and services are provided via the interfaces of the BIG IoT API. Money can be earned with the BIG IoT Marketplace, If data and services are offered by the publisher and used by other consumers. For example, a service to request an available parking lot, is offered [37]. Money has to be spent in order to use this service. A list of available parking lots is then sent to the service user. Another feature would be a reservation service for a parking lot. Their business model is to create a revenue based on different payment models. Small fees on each payment, pay per use, for example API calls or even basic usage fees are an extract of the payment Models [38].

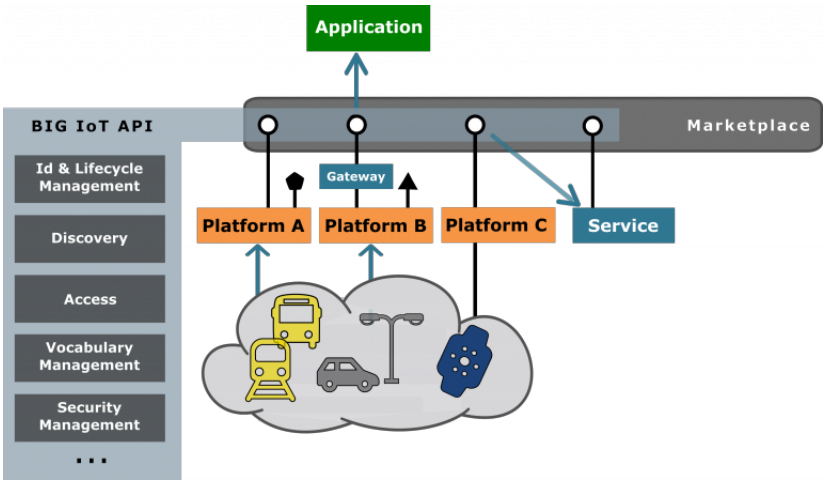

Figure 2.4.: BIG IoT architecture overview from [37]

The platform is interoperable. Vocabularies and another description language to describe data and services are used. The basic difference to OpenIoT's approach is basically that now a segment to make business is added in the description.

Because devices are not directly supported by the platform, BIG IoT is not an IoT platform. Adapters for other platforms and services are supported by BIG IoT. Through these adapters, data and services can be provided and used. BIG IoT is used to generate profit of your data and services. A marketplace for data and services is offered with the BIG IoT Marketplace.

All the data is transported via the platform and direct connections between platforms are not supported.

The chosen data can be offered by every user and therefore only the API has to be used. This means, that there is not any device level software involved.

The BIG IoT platform is hosted and data and services are provided by platforms and services. The usage of data and services can be feed.

2.2. Discussion

Different purposes are followed by current platforms. The machine-to-machine communication, security and data presentation is a critical feature of the platforms that are presented here. Several software tools to present and analyze the gathered data are presented accordingly to the proprietary platforms. Currently their main focus is on companies and their process digitalization.

The proprietary platforms and BIG IoT are built to make money. That is comprehensible, but also the focus on companies is illustrated by that.

All platforms that are presented here except OpenIoT and BIG IoT, are closed monolithic systems, but some kind of description language is used by all of them. A range from basic device descriptions up to data and billing descriptions are offered by them. All this is a good start, but by these descriptions only basic data types or specific devices are described. So the data and device representation is not as good when it comes to more complex objects. These complex objects for example are not only a temperature field, but contextual information about the usage of objects which is given by several sensors. This means, there is a disadvantage of current systems.

The platforms purpose is shown in Tabular 2.1 and 2.2. Furthermore the platforms are briefly classified accordingly to the requirements, that are presented in Section 1.2. The text in some fields of the Tabular is highlighted in the colors green, orange and red. An accordance to a requirement is highlighted in green, a partial accordance is highlighted in orange and a not sufficient accordance is signaled with the color red.

Platform	Amazon AWS IoT	Microsoft Azure IoT	SAP IoT
Purpose	Private and corporate usage	Corporate usage	Corporate usage
Interoperability	No	No	No
Independent	Yes	Yes	Yes
Message Reduction	No	No	No
Dynamic	No	Partial, they use agents for dynamic behavior	No
Administration	Hosted and self managed	Hosted and self managed	Hosted and self managed

Table 2.1.: Comparison of all described proprietary platforms

Platform	Kaa	OpenIoT	BIG IoT
Purpose	Private communication platform	Open sensor networks	Interoperability framework
Interoperability	No	Yes	Yes
Independent	Yes	Yes	No
Message Reduction	No	Partial, push pattern is used	No
Dynamic	No	Partial, because of descriptions	Partial, because of descriptions
Administration	Self hosted and self managed	Self hosted and self managed	Hosted and self managed

Table 2.2.: Comparison of the described open source platform and the interoperable platforms

3. Overview

Currently the IoT is segmented by different kinds of platforms, proprietary systems and sometimes single device groups and a vast amount of different technologies is used. Interoperability in general is not properly supported by the platforms.

In Chapter 2 alternative platforms were presented. The presented platforms are focused on closed systems, that are improved to handle internal traffic, that is generated by devices and services. An approach was done by BIG IoT, to overcome the segmentation with an on-top platform that is an interoperability-layer for different other platforms. But BIG IoT is not an IoT platform, it is built as an interoperability-layer and with this aim, that precondition the platform is not built to handle devices directly. Furthermore no adequate description scheme is used. Either devices can only be described in their basic abilities without context or the devices are represented within the platform and the knowledge how the devices are built.

The IoT platform, that is proposed in this thesis, is designed as an interoperable IoT platform. Thereby it is presented as an alternative IoT platform, where interoperability is explicitly supported. The platform is designed for IP (IPv4 [39] and IPv6 [40]) based communication on Internet-like networks. Light control components like Domain Name Service (DNS) [41] are used in the Internet to establish connections. That idea of light control components is used and adapted to match the requirements defined in this thesis. At this point and because of the limited time, a security mechanism is not proposed in this thesis.

The platform is presented with an open design and a few control structures. Because only IP networks should be supported at first, devices that are connected via other protocols, for example sensor networks, are viewed beginning with their IP gateway. In order to that every device which is not capable of IP or not powerful enough, is connected via an IP gateway.

In this thesis devices are further described as objects. These objects are a basic component in the proposed design. Object implementations are executed by the Object Engine component. Objects, that are executed by an Object Engine, are handled at run-time by the Registry component. The access to the system is granted and managed by the Management component.

The system is accessed by objects and users of the system. The Gateway component is used when a direct connection to a component of the system is not possible. In that case, messages are sent to the Gateway and after that the message is forwarded to the destination component.

To create a connection between objects and users is a key goal of this thesis. The Management and the Registry are the core components of this platform. These two are needed for a working platform. The Gateway, Object Engine and the user can be added if they are needed.
In the following Sections the challenges of such a system are discussed, the core components of the proposed design are briefly outlined and discussed, the goals of this thesis are described and the steps to accomplish these goals are shown.

3.1. Challenges

If an IoT platform is designed, several challenges have to be faced. First, the interoperability issues have to be solved. Therefore an advanced object description language is proposed in Chapter 4. In that description inheritance structures are defined. The inheritance structure is built as a tree, like it is used in Java's object inheritance structure. That description language is used to bind the properties and methods to the object that is described. If an object is defined, so it inherits from another object, all properties and methods are inherited from the parent object. New methods and properties can be added but existing ones are not permitted to be overwritten.

Second, to easily access these methods and properties, three different basic communication patterns (Request-Response, Reactive, Stream) [42] are defined and further discussed in Section 3.3. These two measurements are defined as the cornerstone of the interoperability of the platform. These cornerstones are necessary, because a well-defined communication patterns and descriptions are needed, if a connection between two different devices and objects is wanted.

Third, rules and a rule-based behavior is defined in Chapter 5. The proposed rules are inspired by IFTTT (If This Then That) [43] and the paper [2] proposed roles. The efficiency and the flexibility of the system is improved by the rules. A continuous self-monitoring is needed, in order to check the rules conditions and timer. The state of an device can either be monitored by steady requests or by a reactive behavior of the device itself. If the rules are triggered, the reaction should follow predefined steps.

Fourth, the platform needs to be designed efficiently and decentralized. Therefore a handle-driven platform is proposed. By the handle-driven approach, the traffic should be reduced.

The design also needs a light structure of control services. These services are described in the following Section.

3.2. Requirements Transposition

In Section 1.2 some requirements are defined. These requirements are introduced as crucial properties of an interoperable IoT platform. The requirements and the subsequent measures are introduced in the following Section.

Interoperability is one of the requirements which have been described earlier. It is realized through several steps. First, two communication patterns are defined. Through this a well-defined way of communication is implemented and all participants are enabled to communicate with each other. Second, a description language is proposed, that is an advantage to the existing description mechanisms. Third, the Registry and the Gateway component are introduced. The Gateway is used to establish connections between devices, if a direct connection is not possible. The Registry is designed to easily distribute object information, within the local network and globally.

Independence is realized by the four components of the platform. The Management is introduced as the component, that is used to supervise the platform. The Registry is used to share the object information at run-time. To connect components, if they are not able to communicate directly, is added by the Gateway. A device can be connected to the platform if the Object Engine is used.

Message reduction is defined as a requirement, because it is an improvement for the scalability of the platform. An improvement that can easily be done at design time and not by an infrastructure that scales at run-time. By the direct communication that is introduced by the handle-driven system, the messages that are sent, can be reduced significantly. A reactive behavior is introduced to lower the amount of messages that are sent for status requests. That requests are done internally by the device, to check and execute rules that were previously defined.

A dynamic behavior is a key value and requirement for the IoT. The Object Engine is built to meet that dynamic behavior. Different properties are introduced with the Object Engine. The devices behavior can be manipulated at run-time and devices that are built with the Object

Engine should be connected autonomously to the platform. Also a run-time environment for a reactive behavior is added to the Object Engine.

The Administration of the platform is combined in one component. The Management component is defined to handle all access requests. These requests can be yielded by all parties that participate in a system.

3.3. Communication Patterns

Well-defined communication patterns are needed for an interoperable IoT platform. A communication between the objects and user can be enabled by these patterns.

Publish-subscribe systems are used by other platforms for their message delivery. These systems are loosely coupled by the usage of message servers [44]. The problem is, that messages are sent between device and platform or device and message server, even if the data is not subscribed. If the proposed patterns are used properly, the number of messages can be reduced significantly.

These patterns should be applicable to many different use-cases. The most basic way to communicate is the request response pattern. A request message is sent from host A to host B and the a response message is sent from host B to host A. The pattern is very basic and simple. No complex logic is needed and more important, no state has to be saved. But it is not very efficient. If the request response pattern is used for repetitive requests of a specific value or resources, the amount of request messages rises. In these cases it could become more efficient to use a reactive pattern.

A reactive communication is set up with a condition. If that condition is satisfied, an action will be triggered. For example, every time a status change is detected, all subscribers of the specified condition are notified. The self-monitoring of the objects is a crucial part of the mechanism. The system has to be monitored periodically by itself, so a status change can be detected. Furthermore, the connection needs to be monitored. That monitoring has to be done to ensure, that a subscribed condition is correctly recognized.

The amount of messages that are sent, could be reduced by the reactive pattern. Some efficiency issues are shown by the reactive pattern, if it is not limited. It could result in a flood of status messages. Therefore it has to be parameterizable. The reactive pattern is further

discussed in Chapter 5 as rules.

The stream pattern for continuous method invocation is proposed. This pattern should be used for tasks like video or music streams.

A priority scheme is proposed to support different kinds of urgencies of messages and their processing. By different priorities, the processing of for example medical devices, could be preferred.

Both the stream pattern and the priority scheme are proposed for further work, because of the limited time.

3.4. Design

Four components are defined as cornerstones of the platform design that is proposed here. These cornerstones are the Management, the Registry, the Gateway and the Object Engine. The components are briefly described in the following Sections.

3.4.1. Management

The Management of the platform has to be done at run-time. Therefore the Management component is introduced as the control component.

A control interface and the security system should be managed here. Furthermore, the user management should be done by this component. The list of connected Managements, Registries, objects and user should be controlled within the Management.

Purposed tickets or certificates should be offered by this system. Through this, the permission check for specific actions should be eased.

The Management is further presented and discussed in Section 6.3.

3.4.2. Registry

Object entries and their properties should be managed at run-time by the Registry. In the system, the Registry design is built as a handle-driven broker. Like domain names that are handled by the DNS in the Internet, object entries should be managed by the Registry.

Look-up strategies should be used to retrieve object information from the Registry. Thereby different kinds of information and relations between objects should be retrievable.

An object should can be published to another platform. In that case objects should be added to the connected Registry. In conclusion all objects are saved within the Registry and some of them are published to other platforms.

The Registry is further presented and discussed in Section 6.4

3.4.3. Gateway

The Gateway is defined as a component, because nowadays the networks are divided by network borders called NAT (Network Address Translation) [45]. To solve this problem, the Gateway component is introduced.

Networks which are normally not connected can be connected by Gateways. For example an object in a LAN (Local Area Network) [44] needs to be accessed from outside the LAN. Such objects can normally not be accessed.

In that case the connection should be established via the Gateway. If a message is sent to the Gateway, the message should be forwarded by the Gateway like defined in the message.

The Gateway is further presented and discussed in Section 6.5

3.4.4. Object Engine

Devices can be connected to platforms in several ways. Devices that are programmed with provided SDKs are the most common way today. A static installation of these devices is thereby done.

Object descriptions should be used within this platform, to define the behavior of an object. An object implementation should be the coded implementation of the corresponding object description. Furthermore the object implementation can be added to the Object Engine at run-time. Through this, the dynamic behavior of the device, running an Object Engine, should be significantly increased.

A valid object implementation should be runnable within the Object Engine. The Object Engine should be a well-defined run-time environment for object implementations. A lightweight management interface should be provided by the Object Engine. The interfaces should be used to add objects and to handle incoming and outgoing traffic. A run-time environment for agents and rules should be part of the Management. These run-times are needed for an efficient behavior and decentralized structures. The Object Engine should furthermore be a gateway for devices that are not capable of IP or not powerful enough.

The Object Engine is further presented and discussed in Section 6.6.

3.5. Security

The safety of the users and their assets has always been taken into account. Informations about objects and users are handled within the platform. A first security issues is, that the

platform can not only be used to gather information, but to trigger actions of objects. These actions can be safety-critical and therefore these informations have to be secured as best as they can. Another problem is, that by the proposed object description language the contextual informations are enriched significantly. If the dynamic composition 4.5.3 option is used, then even the placement of safety critical objects can be disclosed. Therefore the discussion has to be done, which information should be described with the offered description options.

There are two security mechanisms that could be taken into account. Authentication, encryption and decryption mechanisms could be provided by different security mechanisms.

An asynchronous cryptographic system in form of a PKI [46] infrastructure or a security system like Kerberos [47] is proposed. The described functionalities should be implementable by both proposals. In further work, a security system should be proposed.

3.6. Summary

The proposed platform is briefly discussed in Table 3.1. The amount of messages is reduced by the rules and the handle-driven platform design. Messages are directly sent between two objects and are no longer sent via the platform. The dynamic capabilities are increased by the Object Engine. Every object can be processed and makes object updates at runtime possible.

In comparison to the IoT platforms, that are examined in behalf of the requirements and are summarized in Table 2.1 and Table 2.2, some improvements are shown by the proposed interoperable IoT platform. Improvements are shown especially when measured against to the requirements interoperability, dynamic behavior and message reduction.

Category/ Requirement	Proposed Platform	Description
Purpose	Private and public	The platform is built to support a community that is willing to share information.
Interoperability	Yes	The interoperability is supported by the proposed communication patterns, the object description language and the components Registry and Gateway. The components are introduced to support the open data thought.
Independent	Yes	The platform is independent because all crucial components and the corresponding mechanisms are defined.
Message Reduction	Yes	The messages are reduced by the handle-driven approach and the reactive behavior (rules).
Dynamic	Yes	A dynamic behavior is supported by the variability of the Object Engine and the reactive behavior.
Administration	Self hosted or hosted and self managed	The administration is defined within a single component. The Management is designed to handle all access and security issues.

Table 3.1.: Discussion of the proposed platform.

4. Description Language

A common understanding of objects and their abilities is needed in an interoperable environment. The knowledge, which specific device is used in a scenario, is utilized to process the data. Such specifications can only be operated in environments where every device and its behavior as well as the specifications are known.

In an interoperable platform the knowledge has to be enriched, so everyone can use the device by its specification and without the need to know by whom the service is offered. Through the creation of description every object, this issue can be solved. One solution would be a description language. In this thesis a description language is introduced as a possible key to interoperability for interoperable IoT platforms. A well-defined description language is essential to establish an interoperable environment.

Vocabularies are used by OpenIoT (Section 2.1.3.1) and BIG IoT (Section 2.1.3.2) to identify specific data types. But the vocabularies are only used to describe basic values and not their semantic context. The temperatures at a specific GPS location is an example for basic values and their combination. But the context of sensor data is a major issue, that is not appealed by vocabularies. For the description of basic values this is not important but if more and more complex objects are described, the complexity can not be depicted.

The object description language, that is presented in the following Sections, is created to improve the context of defined objects and their data.

4.1. Object

A device or a service can be represented by an object. The common description language is chosen as the representation scheme, to establish a common understanding of devices and services. Different properties and abilities of objects can be modeled with the description language. Simple objects can be composed to a bigger complex objects, by combining them in the description.

The object, that is presented in this thesis, is a mixed view of two architecture orientations. Both the ROA (Resource Oriented Architecture) [48] and SOA (Service Oriented Architec-

ture) [44] approach are represented by a described object. First, the ROA approach is symbolized by an identifier that is called object identifier (OID) (Section 4.2). Furthermore some basic description fields are bound to the object. Second, the SOA approach is symbolized by the operations that can be executed by the object. The object is the context for the operation. In conclusion the object is identified by the OID and basically described (ROA) and operations that can be executed by the object (SOA).

Two different kinds of objects are described in this thesis. First the object descriptions and second the object implementations. The object description has to be seen as a contract that is defined before run-time and the object implementation is the implementation of the contract. These two object types are further described hereinafter.

4.1.1. Object Description

The huge amount of different devices that can be bought is a problem for the interoperability. At run-time these objects can not be identified and used, because a common interface is not supported by them. An implicit or a very basic description is used by todays systems.

The object description is introduced to solve these problems. At first, an object is defined before run-time. The description is identified by an OID. This OID is used at run-time to identify the object as an object that is implemented to fulfill a specific description. Furthermore at run-time only the run-time specific information has to be managed. The whole object description is bound to the used OID.

The description or the contract is used to define the capabilities and properties of the object. Furthermore, these objects should at minimum be annotated with an OID, a name and a description. In order to implement the object in the IoT platform, this is the minimum level, that a object should be described with. Nevertheless, the objects should be defined and described as particular as possible, to improve the usability.

An example of a object description is shown in Listing 4.1.

```
{
  "oid": "0.0.1.4",
  "name": "switch",
  "description": "Can be used as a switch"
}
```

Listing 4.1: Description object

At run-time the object is bounded to the previously described object by the OID. To distinguish objects at run-time, a **local_id** is added. In Listing 4.2 an example for an OID and local_id combination is shown.

```
{
  "oid": "0.0.1.0",
  "local_id": "123"
}
```

Listing 4.2: OID and local_id at run-time

4.1.2. Object Implementation

The object implementations are implemented to fulfill the contract which is defined by the description object. At run-time the object implementation is identified by the OID and an additionally generated local_id.

A description object can be implemented in several different ways and for each way, different implementations of an object description can exist. These object implementations should be stored and made available for users and manufacturers.

In order to make the implementations available in a public repository, they should be defined under a license, to regulate the usage of them. The implementations could also be sold via a marketplace that is connected to the public repository.

4.2. Object Identifier

At run-time objects are needed to be identified. The abilities and properties of the object are defined before run-time and at run-time the object description should be represented by and a unique identifier.

The identification can be done in different ways. First, the objects can be defined at run-time by descriptive arguments or second, they are identified by a description language that was previously defined. Descriptions that are defined before run-time, are used in this thesis. These descriptions are identified by the OID.

The object identifier (OID) is specified in Article [49] and is used originally in Simple Network Management Protocol (SNMP) [50] to identify every node in a network. An example for an OID is **1.2.2** . As shown before it is a string consisting of numbers and dots. The inheritance hierarchy is described by the string in which the numbers are separated by dots. Object inheritance trees can be built with this format and the relationship between the objects is shown too.

In SNMP these identifiers are used to identify nodes in networks. In this thesis these OIDs are used to describe inheritance relationships like they are used in the Java object inheritance structure. An example for such an inheritance structure is shown in Figure 4.1.

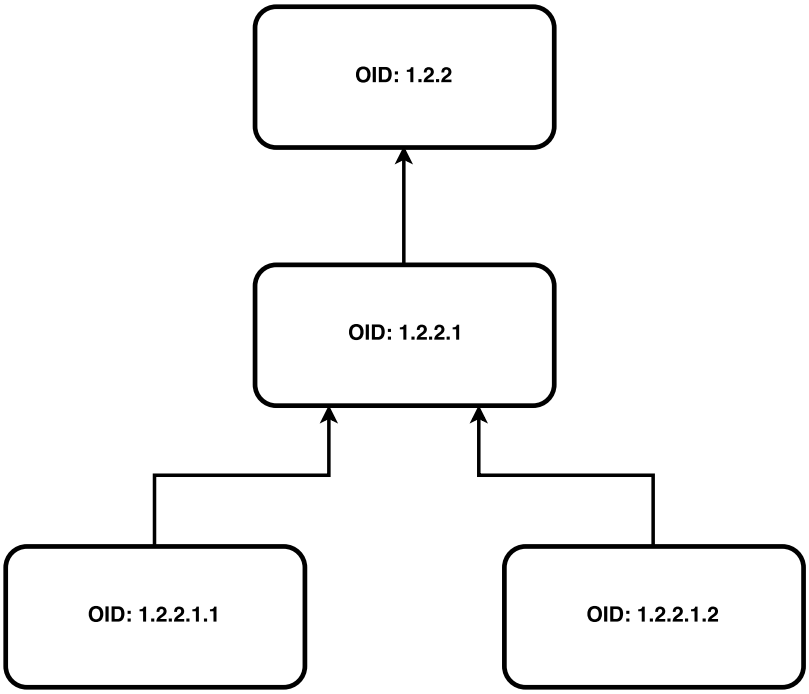

Figure 4.1.: Object identifier example

4.3. Object Hierarchy

Today, the IoT environment is separated and consists of many different objects. Some of these objects are built to fulfill the same task but can not be identified as these objects. An approach that is used by current platforms is to tag the data. But by these tags, the relations between different objects can not be displayed.

A hierarchy of objects is presented in this thesis. Information of abilities and properties of objects are contained in the object hierarchy. An inheritance mechanism, similar to the here introduced hierarchy, is used in the object oriented programming language Java.

In Java, the abilities and properties of the basic Java object are inherited by every Java object. A structure with properties and abilities is built with the inheritance model. In this system a group of objects can be selected to fulfill a task that it is defined according to their capabilities. Object identifiers (OID) are used to define which other object inherits from which specific object.

Like it is done in Java, the basic object is inherited by every object in this inheritance model. Private OID areas should be defined which should be used as a test environment. Furthermore,

vendor specific areas should also be considered.

An object identifier example, that is shown in Figure 4.1. In this example the object with the OID **1.2.2** is defined as a parent object for all other shown objects. The object **1.2.2** is inherited by object **1.2.2.1** which is again inherited by The objects **1.2.2.1.1** and **1.2.2.1.2**. This means, that if the object **1.2.2** is defined with a certain ability, this ability is provided by all other objects, which are inherited by object **1.2.2**.

Four basic description objects are the base of the object hierarchy. The first is the regular object which is the root of the inheritance tree (in Figure 4.2). The function of the other three description objects is to categorize the objects right from the beginning. Further categorization description objects are recommended and should be added to simplify the inheritance structure and make classes of objects easier to recognize. A first essential step to improve semantics for different kinds of objects is proposed with the categorizations in the inheritance model.

Physical Object A real-world object is represented by the physical object. Examples for physical objects are chairs, TVs and wearables.

Non Physical Object Non-physical objects are described as objects that do not exist in the real-world. These kinds of objects can be used in AR- and VR-environments.

Service A service describes services that have a specific behavior such as a video streaming. These kind of objects are not bounded to a specific existing object. Physical, as well as non phyisical, objects are meant by the term existing. The service can be seen as a micro service.

Private Object This area can be used to define own objects. So this number space is reserved for self created objects and objects that have to be tested in the later on defined environment. At run-time they can only be used in self defined scenarios, but the semantics are not readable by others.

4.4. Data Format

The representation of the description language is an essential aspect. There are different concepts of object notations and meta-languages that can be used for that purpose. Because of their popularity JSON and XML are chosen as possible candidates. JSON and XML are used in a wide range of use-cases. Both have different properties which are shown in the following description.

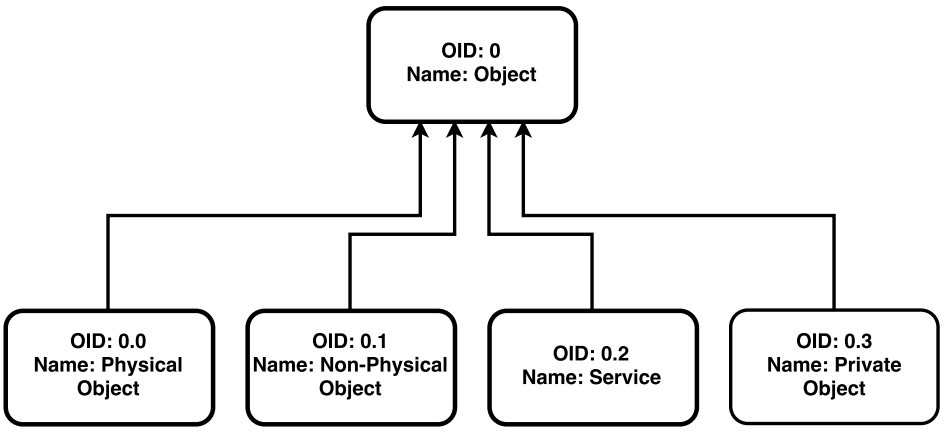

Figure 4.2.: Basic object inheritance scheme

JSON The JavaScript Object Notation (JSON) is a text-based, compact data storage and exchange format. JSON is used in JavaScript to describe objects internally. Externally JSON is used to exchange data. The notation is designed to be efficient, self-describing and easy to use. Furthermore the notation is a native part of JavaScript and therefore simple to process in a JavaScript environment. [51]

XML The Extensible Markup Language (XML) is a text-based format for the exchange of structured data. XML is built in order to be human readable and simple to use. It is applied in many use cases. A XML document can appear in two different states. These states are described as well-formed and valid. A document is called well-formed if it is conform to the syntactical rules of XML. A XML-formatted text is called valid if it is well-formed and it is conform to all rules defined in the DTD. The structure of the document is defined by the Document Type Definition (DTD). [52]

In this thesis, JSON is used for the prototypic descriptions, because it is easy to use and fast in development. In further work, XML should be used for the descriptions, because of the powerful validation feature.

4.5. Optional Descriptions

In order to improve the contextual knowledge the object should be described as precisely as possible. Therefore An object can be defined by several additive descriptions. In the following Sections the optional descriptions are presented.

4.5.1. Vocabularies

Vocabularies are used to establish a basic knowledge of words in an interoperable system. If the specified vocabulary is used, the words, that are defined in the vocabulary, can be understood and interpreted by every program.

An example for such a vocabulary is Schema.org. Schema.org was founded by Google, Microsoft and others. It is used to describe properties, entities, relationships between entities and actions. [53]

A vocabulary can be used to describe different types of data and locations in which the data is generated. This feature is essential for an interoperable platform. The purpose and context of different platforms can be made clear. By that, the usability of these platforms and services can be improved significantly. Vocabularies are identified by URIs. In Listing 4.3 an example for a vocabulary definition is shown.

```
{
  "vocabulary_count": 1,
  "vocabularies": {
    "vocabulary_0": {
      "prefix": "dcterms",
      "URI": "http://purl.org/dc/terms/"
    }
  }
}
```

Listing 4.3: Vocabulary definiton

The field **prefix** is introduced in the example from Listing 4.3. A unique identifier is needed for every vocabulary that is defined. An example with two vocabularies is given in Listing 4.4.

```
{
  "vocabulary_count": 2,
  "vocabularies": {
    "vocabulary_0": {
      "prefix": "dcterms",
```

```
      "URI": "http://purl.org/dc/terms/"
    },
    "vocabulary_1": {
      "prefix": "foaf",
      "URI": "http://xmlns.com/foaf/0.1/"
    }
  }
}
```

Listing 4.4: Vocabulary definition with two vocabularies

The number of used vocabularies is expendable, like it is shown in Listing 4.4. A new vocabulary can be added, as long the new prefix is different to the previously defined prefixes. An overview of vocabularies is given by the Open Knowledge Foundation [54].

4.5.2. Basic Data Types

Some basic data types are introduced along with the description language. The basic data types are added to simplify the usage of the platform. These data types can be used if the prefix **bsc** is used. **bsc** is the short form for basic. In the following description, the basic data types are defined.

bsc:boolean - true or false as a string

bsc:integer - 32 bit signed value

bsc:long - 64 bit signed value

bsc:double - 64 bit floating point value

bsc:string - a UTF-8 encoded text

The time unit is standardized to the value μs and the measurement unit for distances is standardized to the value μm.

4.5.3. Composition of Objects

The issue that a complex device has to be described, is not known by devices whose abilities and properties are known implicitly. Complex objects can not be described by current description structures, that are used by most platforms today.

The object composition is a key feature of the proposed description language. Complex objects can be described by other objects. So every complex object should be decomposed

to several simple object and then be reattached. Complex objects are called ambient objects. Objects without other objects bound to it are called simple objects. The divide and conquer principle is used with this approach. Simple objects are placed in coordinate systems. A coordinate system with a root point is defined by the ambient object. Simple objects are arranged in the coordinate system of the ambient objects. All composed objects are placed in relation to the root point of the object it is composed with. A point in this coordinate system is defined by the values of the coordinates in x-, y- and z- direction. Furthermore, an orientation is added with the corresponding angles, that are given with the coordinates. So six values are defined to describe a position and the orientation of a point in a coordinate system.

It should be noticed that only the ambient object knows about the attached objects. The simple and complex objects are not aware of their context outside their definition. Also, every ambient object can be a simple object that is composed to another ambient object.

More context can be generated by the composition and the produced data is now better to understand. The data is enriched by given information, to increase the processing outcome of the enriched data.

There are two kinds of compositions. On the one hand are the static members, on the other the dynamic members. Further descriptions can be found in the following Sections.

4.5.3.1. Static Members

Static members are defined as objects that are composed to an ambient object before run-time. If an object has static members, it is defined as a complex object.

The object that is composed to the ambient object should be positioned in the coordinate system of the ambient object. If it is not positioned, the object is bounded to the ambient object but does not have a specific coordinates. The possibilities of this language and the context of the objects could be maximized, if the object is positioned in the coordinate system. It is highly recommended to position the object.

A static member is a kind of composition, that requires a specific position of the member in the coordinate system of the ambient object. The members are positioned referring to the root coordinates of the ambient object. As described before, the mounting point is defined relative to the object root. A static member description is shown in Listing 4.5. If the composition is done without a specific position, the field **mounting_point** is defined as an empty object.

```
1 {
2   "member_count": 1,
3   "static_members": {
```

```
    "member_0": {
      "description": "seating",
      "oid": "0.0.23.0",
      "mounting_point": {
        "x": 0,
        "y": 60000000000,
        "z": 0,
        "angle_x": 0.0,
        "angle_y": 0.0,
        "angle_z": 0.0
      }
    }
  }
}
```

Listing 4.5: Static member definition example(with coordinates)

4.5.3.2. Dynamic Members

A dynamic member is described at run-time. These are dynamic compositions, that should be defined to enrich the context informations of the objects.

For example, if a *table* object is combined to an ambient object *room*. Then the *table* object would be a dynamic member of the ambient object *room*. These composition can either be defined with or without a position in the coordinate system of the ambient object.

The composition of objects is recommended to increase the context of the produced data and functions, but the composition is not required. A dynamic member is defined equally to a static member only at run-time

4.5.4. Description

The object is described by a text. In this text, the object is shortly described. Furthermore, the abilities and the purpose of the specific object should be illustrated.

An example for such a description is shown in Listing 4.6. The description should be used in other contexts too.

```
{
 "description": "A bulb that can be switched on and off.
                 It is used in systems like Smart Home."
```

```
}
```

Listing 4.6: Description example

4.5.5. Methods

An object is built and described to fulfill a purpose. The abilities that are needed should be defined.

These abilities are called methods. Methods are defined as actions which can be performed in the context of the object. First the amount of methods is defined by the field **method_count** and second the methods themselves are defined by the **methods**. In this term the methods are defined like shown in Listing 4.7.

```
{
  "method_count": 1,
  "methods":{
    "method_0": {
      "name": "setHeightInPercent",
      "is_status_request": false,
      "parameter_count": 1,
      "parameters": {
        "parameter_0": {
        "type": "bsc:int",
        "name": "newHeightInPercent"
        "precond": "newHeightInPercent >= 0
            && newHeightInPercent <= 100"
      },
      "return_values_count": 1,
      "return_values": {
        "return_value_0": {
          "type": "bsc:boolean",
          "name": "successful",
          "postcond": "successful == true || successful == false",
        }
      },
      "description": "sets the height of the seating in percent.",
      "option_count": 1,
      "options": {
        "option_0": {
          "description": "sets the height of the seating in percent.
```

```
28                              Faster height correction than operation_2"
29         }
30       }
31     }
32 }
```

Listing 4.7: Methode definition example

The fields, that are used in Listing 4.7, are further discussed in the following description.

is_status_request is defined as true, if no state change is triggered by the invocation of the specific method. The field is defined as false, if a state change could be triggered.

parameters are defined in their position by the parameter numeration. In order to that, the first parameter is called parameter_0. **type** and a **name** are used to define the parameter.

return_values are equally described as parameters.

conditions are defined either by conditions or by text. Conditions should be used for values of the typ bsc:integer, bsc:long, bsc:double and bsc:float. In case of a bsc:string the validation possibilities are limited. The validation of the length of a string is an example for a validation capability for bsc:strings. The conditions are not furhther described in this thesis.

options Different implementations of the same method should be described by an option. These options are defined by a string. This string is used to define the differences between the default implementation and the option.

4.5.6. HAL

The Object Engine is defined and introduced as a gateway for devices. The HAL in this design is the link between the Object Engine and the device with its special interface.

The hardware interface between the Object Engine 6.6 and the device itself is defined by a Proxy [55]. That Proxy is called Hardware Abstraction Layer (HAL) in further discussions. Operations are introduced as actions that the HAL is able to perform. The field **methods** is used to describe the operations, that are defined accordingly to the object methods. Operations, that are not previously defined, are named by the field **operations**. An operation is defined equally to methods, that are defined in Section 4.5.5. A HAL description is shown in Listing 4.8.

```
1 {
2   "hal": {
```

```
    "methods_count": 3,
    "methods": ["method_0", "method_1", "method_2"],
    "operations_count": 1,
    "operations": {
      "operation_0": {
        "name": "isInitialized",
        "parameter_count": 0,
        "parameters": {},
        "return_values_count": 1,
        "return_values": {
          "return_value_0": {
            "type": "bsc:boolean",
            "name": "initialized",
            "postcond": "initialized == true || initialized == false"
          }
        }
      }
    }
  }
}
```

Listing 4.8: HAL definition example

The efficiency and the response time on events can be improved by a reactive mechanism, that should be implemented. This mechanism should be used to get new information as fast as possible. The normal way is, that an action is triggered by the HAL. These different interaction pattern for the HAL and the device should be defined in further work.

4.5.7. Shape

The shape is an image of the object and could be used to picture the object in other programs. A possible use-case would be a design software for objects or furthermore, the shape could be used to show and interact with these objects in VR- and AR-environments.

Every shape is defined by three anchor coordinates within the 3 dimensional space and a corresponding angle for every dimension. This means, the object's anchor point is defined by a point with 6 dimensions.

The measures of the shape are referred to the anchor point coordinates. The shape could be edited at run-time but that would be part of further work. A dynamic shape would be interesting in AR- and VR-environments.

Four different shapes are supported by the platform. These shapes are adapted from the JavaFX terminus [56] and are listed in the following description.

box represents a box with the attributes

- value_x - width
- value_y - height
- value_z - depth

sphere represents a Sphere

- value_z - radius

cylinder represents a Cylinder

- value_y - height
- value_z - radius

triangle_mesh represents a Triangle Mesh

- vertex_array - an array of 3d coordinates(x,y,z), where vertex_array[i], vertex_array[i+1] and vertex_array[i+2] with $((i \equiv 3) = 0)$ are defining a vertex.
- face_array - an array of vertex_arrray indices, where a face is defined by three vertices. face_array[i], face_array[i+1] and face_array[i+2] with $((i \equiv 3) = 0)$ are defining a face.

Because the shape is no crucial part of the platform, it is not further mentioned.

4.5.8. Location

An object location is described by a GPS-position. The location of an object in a global context is not accurate and the support of indoor positioning is not given by GPS-like systems. For an exact location or even a better composition, the object should be composed with other objects. GeoJSON [57] is an encoding format for GPS-positions formatted in JSON. To support different types of encodings, the GeoJSON object is wrapped in the format. The encoding style is identified by the **type** field.

These compositions are described in Section 4.5.3.

```
1 {
2   "location": {
3     "type": "GeoJSON",
```

```
    "obj": {
      "type": "Feature",
      "geometry": {
        "type": "Point",
        "coordinates": [53.557067, 10.023116]
      },
      "properties": {
        "name": "HAW Hamburg BT7"
      }
    }
  }
}
```

Listing 4.9: Location definition example with GeoJSON

4.5.9. Descriptions Summary

The description of an object at run-time is different to the description that was done before run-time. At run-time a specific object is described, a whole group of objects was described previously. The descriptions at run-time are stored and managed by the Registry (6.4). The descriptions are summarized and shortly described in Table 4.1.

4.6. Standardization

A form of standardization is highly recommended to control the hierarchy of described objects. OID spaces are distributed by the standardization organization to all kinds of vendors and participants in general. An example for a distribution by a standardization organization is firstly the IPv4 address spaces [58], that are managed by the IANA and secondly the Structure of Management Information [59] also supervised by the IANA.

The object hierarchy should be supervised by a standardization organization. Because of this the proposed open object hierarchy is prevented from being used to serve only one participant. Furthermore the valid object hierarchy should be supervised by the organization, in order to keep the hierarchy logical consistent.

Descriptions	Before Run-Time	At Run-Time	optional
OID	Used to define inheritance	Used to identify an object	No
local_id	No	Defined and used	No
Description	Yes	Possible	before run-time no, at run-time yes
Vocabularies	Used, to define values	Implicitly now because of description	Yes
Composition of Objects	Static member	Dynamic member	Yes
Methods	Yes, to describe objects abilities	Previously described methods are used	Yes
HAL	Defined as a contract between description object and real device	Used by the object implementation	Yes
Shape	Described	Used to present the object in different environments	Yes
Location	Not defined	Defined as a GPS-position	Yes

Table 4.1.: Summary of proposed descriptions

4.7. Experiment

An object is defined in the proposed object description language. In this experiment a simple object is described before run-time, thereby the potentials of the description language are shown. The object, that is represented here, is a physical object, with an inheritance tree, that is shown in Figure 4.3.

The results are discussed later on.

4.7.1. Setting

Single components of the description are discussed here and the complete description is shown in Appendix A.1.

The description, of the box that was earlier mentioned, has an OID. With the OID in Listing 4.1 the object can be identified as an object that is built subsequent to the specified contract. The **internal_timer** field is used, to specify the internal timer interval. Through this, the value rate for internal refreshes is specified.

```
1 {
2   "oid": "0.0.8080",
```

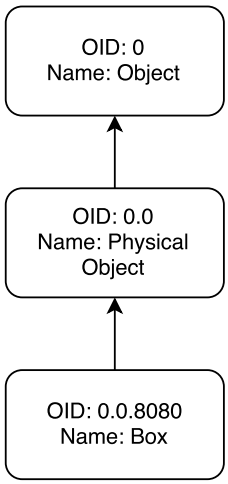

Figure 4.3.: Example object inheritance

```
  "name": "Box",
  "description": "A box that can be opened and closed.
    Furhtermore it can be monitored,
    if something is inside that box."
  "internal_timer": 1000
}
```

Listing 4.10: Description object

Because only the basic data types are used, no additional vocabulary is needed. The basic data types are now usable by adding the prefix bsc: to the type definition.

```
{
  "vocabulary_count": 0,
  "vocabularies": {
  }
}
```

Listing 4.11: Vocabulary definition

In Listing 4.7 example definitions for methods are shown. The field **is_status_request** is used to identify the method as a non-status changing method. The method can be called, without any concern of status changes. If the field is_status_request is defined as true, then the method has to be a status retrieving method.

```
{
  "method_count": 3,
```

```
  "methods":{
    "method_0": {
      "name": "openBox",
      "is_status_request": false,
      "parameter_count": 0,
      "parameters": {},
      "return_values_count": 0,
      "return_values": {},
      "description": "Opens the box.",
      "option_count": 1,
      "options": {
        "option_0": {
          "description": "Open the box faster."
        }
      }
    },
    "method_1": {
      "name": "closeBox",
      "is_status_request": false,
      "parameter_count": 0,
      "parameters": {},
      "return_values_count": 0,
      "return_values": {},
      "description": "Opens the box.",
      "option_count": 0,
      "options": {}
    },
    "method_2": {
      "name": "isBoxOpen",
      "is_status_request": true,
      "parameter_count": 0,
      "parameters": {},
      "return_values_count": 1,
      "return_values": {
        "return_value_0": {
          "type": "bsc:boolean",
          "name": "open",
          "postcond": "open == true || open == false"
        }
      },
```

```
43        "description": "State of the box.",
44        "option_count": 0,
45        "options": {}
46      }
47    }
48  }
```

Listing 4.12: Methode definition example

The object that is executed in the Object Engine, is just a gateway for the device. To have a common device interface for the object, the objects hardware methods are defined by the field **hal**. In this example, all needed device methods are defined alike the previously ones.

```
1  {
2    "hal": {
3      "methods_count": 3,
4      "methods": ["method_0", "method_1", "method_2"],
5      "operations_count": 0,
6      "operations": {}
7      }
8    }
9  }
```

Listing 4.13: HAL definition example

Two objects are composed to the box. First, a switch is attached as a member with position coordinates to the box. Second, a LED is attached to the box without a definite position.

```
1  {
2    "member_count": 2,
3    "static_members": {
4      "member_0": {
5        "description": "switch",
6        "oid": "0.0.1024",
7        "mounting_point": {
8          "x": 0,
9          "y": 60000000000,
10         "z": 0,
11         "angle_x": 0.0,
12         "angle_y": 0.0,
13         "angle_z": 0.0
14       }
15     },
```

```
16      "member_1": {
17        "description": "LED",
18        "oid": "0.0.2048",
19        "mounting_point": {}
20      }
21    }
22  }
```

Listing 4.14: Static member definition example

4.7.2. Results

In the example an object description before run-time is shown. The object is described as a box.

Additional **vocabularies** are not defined, because only the basic **bsc:** vocabulary is needed. But it is shown, how these vocabularies would be used.

Some abilities in form of **methods** are added to the object. There are two types of methods yet. First, the ones a status change can be triggered with. Second, the ones no status change can be triggered with.

The **hal** is defined by 3 operations that have the same syntax,like the methods that have been defined in the **methods** part.

A switch and a LED are composed to the box object. The switch is attached through a position. A huge benefit is generated by that. The context of the object and its position can be interpreted by that.

Another advantage is the modularity of the description language. Only those descriptions have to be defined, that are wanted. A disadvantage is, that a new object has to be defined and added to the description repository for every little edit of positioning. Furthermore, the description language needs to be improved in further work.

4.8. Discussion

The object description language is highly scalable. The numbering scheme of the OID is not limited and therefore the descriptions can be highly scaled. But in deep inheritance hierarchies the number sequences can get very long. The number spaces could be separated in company specific number spaces and organization specific number spaces, to establish a common system for all users. A fragmentation of the number areas would be done when different organizations

get their own number spaces. These issues have to be discussed in further work.

Objects are described in two states. These states are before run-time and at run-time. A kind of a gateway is implemented by the implementation of an object. The device methods are defined by the **hal** field. The methods which are offered by the device are used by the object. So the object is an abstraction layer for the actual device. By the abstraction of the device a generalization of devices that are built to do the same tasks, can be done. Due to this generalization, the complexity of different devices can be reduced. Furthermore, the devices are identified by the OID. Because of that, the amount of data, that has to be transferred, stored and evaluated at run-time, is significantly reduced. Because the definition of the objects is mostly done before run-time, additional structures for described devices have to be added. Also, by the added abstraction of the devices, the error rate and the communication delay are increased.

IP-capable devices are needed by this platform. Devices, that are not IP-capable or constrained, are connected to the platform via a gateway, like mentioned before. Like it was described before, the abstraction has some positive and non positive properties.

A core classification for devices is provided by the object hierarchy. The core classification is defined by the object, the physical object, non-physical object, the private object and the service. The core classification that is defined here, is very basic and needs to be adapted and enhanced in further work. The defined objects can be defined by the usage of the optional descriptions that are presented earlier. At run-time further descriptions could be added, but that mechanism has to be evaluated in further work.

The composition of objects is used to minimize the problem size to small pieces instead of design a complex object at once. A complex object is described as a composition of simple objects. These simple objects are called static member of the complex objects at description time. At run-time these compositions are called dynamic member of the complex object. The principle divide and conquer is followed by this and the re-usability of simple objects is increased. The dynamic member relations have to be managed at run-time. A big effort for the Registry could be produced by that, if the complexity of possible objects is considered. But through the memberships a big advantage is offered. The context of different objects can be symbolized now. At first a connection can either be unlocalized or localized. Further relations have to be identified in further work.

Vocabularies are used by the proposed description language to describe the types of values. Not types but objects and composed objects are defined by the proposed description language, but vocabularies are also used to define types.

The vocabulary idea is lifted to a whole new level by the object descriptions. These objects could be described by data storage formats JSON and XML. Both formats are able to save and exchange all properties of the described objects. JSON is easier to use and read, but XML could be validated against the DTD. Therefore XML is the better choice, and therefore should be used in further work.

The object description language should be standardized and the defined objects should be managed by a form of authority organization. To generate the maximum benefit of the language, it should be used globally and therefore an object identified by an OID should everywhere be identified as the same object with the exact same description.

The object should therefore be as well described as possible. Through this, the benefits of the description language can be maximized. But assumingly the objects will not be better described as the producer or description editor needs it to be. Due to this, only a few requirements for a minimum specifications are introduced and can be seen in Table 4.1.

If no optional descriptions are used, the memory footprint at run-time is very low. The devices are only managed by their OID and the local_id. If additional descriptions are used the memory usage is increased. That behavior has to be evaluated in further work.

Due to the OID scheme, a high complexity can be described at run-time, while providing the fewest amount of information. In conclusion the description before run-time is complicated and at run-time the benefits are generated by the description.

In Table 4.2 the proposed description language is compared with the description language of BIG IoT and OpenIoT.

Some improvements can be seen in terms of complexity of the described devices and the efficiency at run-time. A disadvantage is the longer development time, that is needed for the description. The descriptions have to be published before run-time.

Descriptions	BIG IoT & OpenIoT	Object Description Scheme
Complexity of descriptions	Basic services and capabilities are described. Fields like a location or pricing can be added. Vocabularies are used by them, to describe data.	The description of highly complex objects is possible by the composition of objects. Vocabularies are used to describe data types.
Context	Contextual information is given by extra fields, in a limited way.	The context of objects is enhanced, if they are placed in a coordinate system. Relations between objects are machine readable and can be used to gather contextual information.
Data before run-time	Before run-time only the used vocabularies and the used description language are defined	The language needs well defined objects before run-time in addition to the defined vocabularies.
Data at run-time	High need of data storage for all kinds of descriptions. Messages are bigger when the description is exchanged	The data at run-time is limited to the needed informations. The object is identified by the OID and a local_id. More storage is needed, when optional descriptions are used.
Faster in development	Fast, because the data only has to be described by the vocabularies	Not so fast, the objects needs to be created and registered before they can be used at run-time.
Faster in usage	At run-time the description has to be interpreted	Only the OID has to be searched. After that additional contextual knowledge could be gathered, if needed.

Table 4.2.: Comparison of the in Chapter 4 introduced description language with the schemes used by BIG IoT and OpenIoT.

5. Rules

Rules are used to give the objects a well defined mechanism for autonomous actions. They should be used to reduce messages and even reduce the computing power of central computing instances. In this case central processing units are less used because simple decisions and reactions can be triggered directly by the objects. In order to that fewer messages are needed to monitor the status updates and to react accordingly. This kind of rules is used in some platforms and is also proposed by [2] and [60].

The idea of rules is taken up in this thesis. Two kinds of rules are proposed, periodic rules as well as conditional rules. Actions can be triggered by both, conditional and periodic rules. The properties of conditional rules, periodic rules and actions are described in the following Sections.

5.1. Periodic

The periodic rules are used to trigger actions periodically. A periodic rule is set up with an **interval** value. That value is used to define the period in that the rule is triggered. The defined time depends on the internal object timer that is defined in the description.

The periodic rule example described in Listing 5.1 triggers the defined action every second.

```
{
  "rule_type": "periodic",
  "interval": 1000000,
  "action":{
    /**  **/
  }
}
```

Listing 5.1: Periodic rule definition example

5.2. Conditional

A conditional reaction pattern is proposed by both [2] and [60]. Reactions based on less-equal, greater-equal, equal and span are introduced by their patterns. To complete this range of patterns, the patterns **less** and **greater** should be added.

These kinds of systems are already in use. That idea is adopted by IFTTT(If-This-Then-That) [43]. Triggers are described in receipts and they are used by them to fulfill actions when the specified event occurs. The rules that are defined here are different because of their basic value approach, but the idea is the same.

All introduced conditional rule patterns are described in Table 5.1. In the table the operands named **op1** and **op2** are used. These operands are described to clarify the meaning of the pattern. The order of the operands is described as **op1** first, **op2** second and so on.

Pattern name	True if	Operands
less	op1 < op2	op1 is the return value of a method. op2 can be a method return, a specified value or the previous value.
less_equal	op1 <= op2	op1 is the return value of a method. op2 can be a method return, a specified value or the previous value.
equal	op1 == op2	op1 is the return value of a method. op2 can be a method return, a specified value or the previous value.
greater_equal	op1 >= op2	op1 is the return value of a method. op2 can be a method return, a specified value or the previous value.
greater	op1 > op2	op1 is the return value of a method. op2 can be a method return, a specified value or the previous value.
inside	op1 <= op3 <= op2	op1 and op2 are specified values. op3 is the return value of a method.
outside	op3 <= op1 <= op2 OR op1 <= op2 <= op3	op1 and op2 are specified values. op3 is the return value of a method.

Table 5.1.: Summary of defined conditional pattern and their properties.

The idea of level and edge sensitive rules is added to the system. Level sensitive rules are triggered as long as the condition is true and edge sensitive rules are triggered int the moment the transition of the logical value from not true to true is done. If an edge sensitive rule

is triggered by the transition, the rule is disabled until the result of the condition is false. Afterwards the rule is enabled again.

Furthermore the **timeout** field is added. The **timeout** is defined to disable the rule for the specified amount of time, after the rule was triggered. An example for a conditional rule definition is shown in Listing 5.2.

The fields **timeout** and **sensitivity** are optional. The default for field **timeout** is 0 and the default of **sensitivity** is "level".

```
{
  "rule_type": "conditional",
  "interval": 1000,
  "sensitivity": "level",
  "timeout": "20000",
  "pattern":"less",
  "operands":{
    "operand_1":{
      "type": "method_return",
      "method":{
        "name":"method_example_name",
        "parameter_0": "example_string"
      }
    },
    "operand_2":{
      "type": "specified_value",
      "specified_value":{
        "type":"example_type",
        "value": "example_string"
      }
    }
  },
  "action":{
    /**  **/
  }
}
```

Listing 5.2: Conditional rule definition example

5.3. Actions

Actions are defined as the reaction to triggered rules. The behavior that is defined by the actions, is characterized to act on behalf of central processing components. The message amount is reduced because the action reduces the amount of status- and action-messages. Three different kinds of actions are proposed and described in the following Sections.

5.3.1. Messages

A message is a predefined set of data that is sent when the previously defined rule is triggered. The text is directly defined in the JSON string, like shown in Listing 5.3. The message is delivered to the defined host.

That could be used to trigger a behavior if the action is triggered. In that case a message is sent to a defined host. With this message a method can be triggered, or another step could be initiated. Through this, a device to device communication is added to the platform.

```
{
  "action":{
    "type": "message",
    "message": "exampleText"
  }
}
```

Listing 5.3: Message action definition example

5.3.2. Methods

A method is defined as a method invocation. The method invocation is defined by the method **name** and the **parameters**. Both have to be used to describe a method action. The object where the method invocation takes place is identified previously, so here only the method **name** has to be mentioned. An example for a method invocation is given in 5.4.

```
{
  "action":{
  "type": "method",
  "method": {
    "name": "method_x.name",
    "parameters": {
      "parameter_0": {
        "value": 0
```

```
      },
      "parameter_1": {
        "value": true
      }
    }
  }
}
```

Listing 5.4: Method action definition example

5.3.3. Agents

An agent [44] is defined as a program that is loaded into an object to execute it in that environment. This kind of agent is used here. The agent can only be executed within the Object Engine. The execution is only allowed to take actions that are contained by the Object Engine.

Agents can be used for the preprocessing of gathered data. The central processing of this data is reduced by that. An example for an agent description is given in 5.5.

```
{
  "action":{
    "type": "agent",
    "description": "descriptionExample",
    "agent":"exampleCode"
  }
}
```

Listing 5.5: Agent definition example

5.4. Discussion

The number of messages can be reduced by the rules. The status monitoring is done internally. Therefore, no messages have to be sent. Furthermore the delay for a reaction is reduced, because the reaction is triggered directly by the object. To enable the sensitivity option of conditional rules, an internal monitoring of the status of each rule is needed.

There are several difficulties that have to be handled in order to enable rules. Basic questions are, who is allowed to add or delete rules. Who is allowed to observe rules and their behavior. Actions can be triggered by objects that are not even wanted to be connected. Due to the limited time of this thesis, these challenges have to be addressed in further work.

The typing of variables, makes it possible to use conditional rules. Parameter for condition checks could be added and proofed by their type.

Furthermore the agent code has to be checked if it is defective, but also it has to be checked if its memory footprint and communication activities are normal and as estimated. The agents also need to be checked in their run-time properties.

The direct communication feature that is added by the action message needs to be defined in further work. The message itself can not be used to trigger an action in a more secured system. But because of this feature the behavior of the platform and the connected devices would be enriched significantly.

The benefits of rules are described in detail in the Paper [2] and therefore they are not further discussed here.

6. Interoperable Platform Design

The IoT will consist of a huge amount of devices. Affected by that, the amount of data that has to be transported will increase significantly. Current IoT-ecosystems are proprietary built solutions for vendor specific goals. Because of that an interoperable function is not provided. In order to maximize the positive outcome of the IoT, the requirement interoperability, which was defined in Section 1.2, needs to be established. Furthermore all requirements from Section 1.2 are needed to be implemented.

In Chapter 2 some proprietary platforms are discussed. An outcome of that discussion was, that interoperability is not supported by the proprietary platforms, but some of the other requirements are implemented. An overview of the implemented requirements can be seen in the Tables 2.1, 2.2 and 3.1

The proposed platform design is introduced shortly in Chapter 3. In Figure 6.1 the basic interfaces and core components are shown. In the following Sections the proposed platform is introduced and the core components are further described.

6.1. Introduction

The proposed platform is a distributed system with some elementary management and distribution components. The platform is designed as an open communication platform for interoperable objects. The proposed solution is introduced as a handle-driven platform, in order to reduce central components. Direct connections between two components are preferred and gateways are used only if otherwise a connection it is not possible.

The core of the proposed platform is created by some well-defined components and messages, which are introduced later on. The interoperability is supported by the open platform design and the description language introduced in Section 4. The context awareness is improved by well-defined objects and the objects should be as well described as possible. The efficiency and usability is increased by the communication patterns and the description language defined in Chapters 3 and 4.

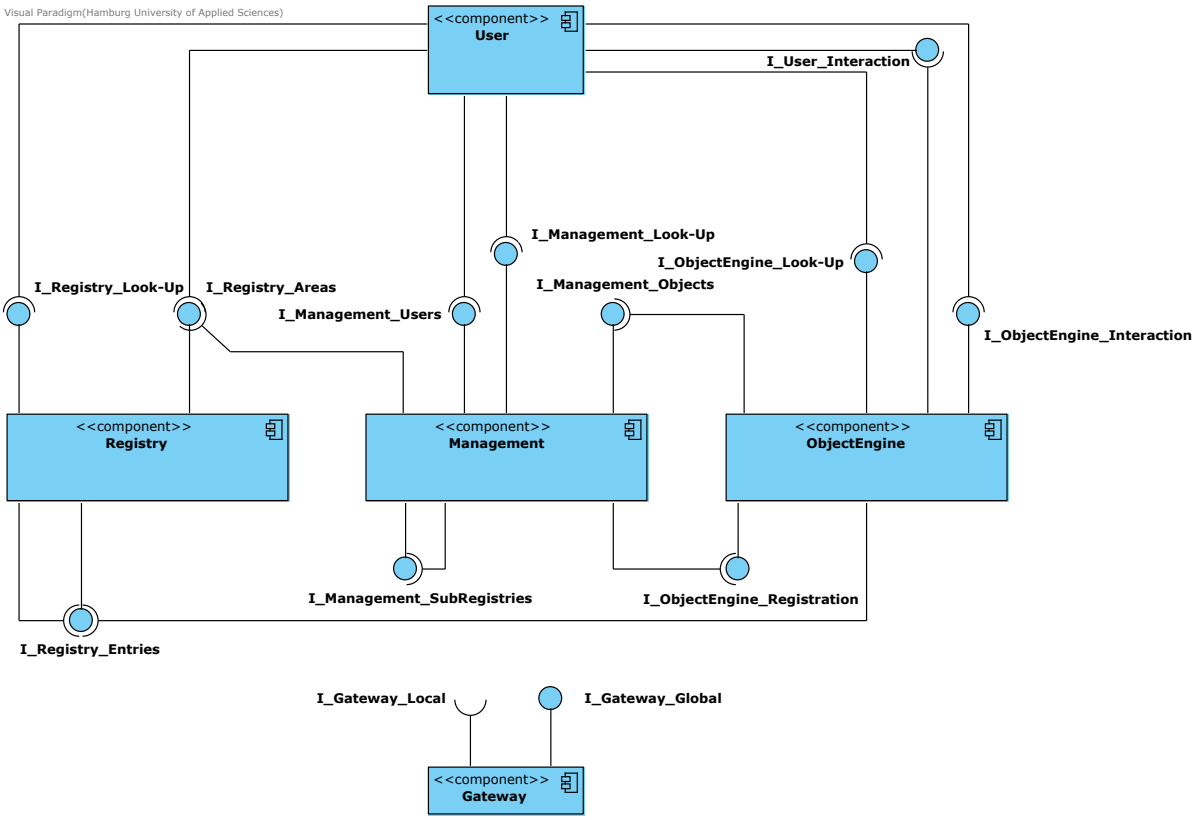

Figure 6.1.: Toplevel view

In order to accomplish the defined goals, the following components are defined and described afterwards.

6.2. User

A User in this context is defined as a person or program that interacts with the platform. The interaction is done by the, in Chapter 7 characterized, messages and sequences of messages.

One interface is offered by the User component, like showed in Figure 6.2. The interface *I_User_Interaction* is used for responses that are not sent immediately. An example for this is the added rule. A response of a rule can be sent at different points in time.

The User uses interfaces of the other components like described in Figure 6.1.

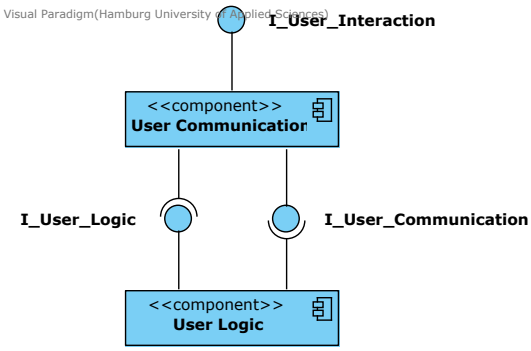

Figure 6.2.: User component

6.3. Management

An efficient control component is needed by an IoT platform. Such components are built to manage users, devices and published devices.

Current platforms are built as monolithic systems with build-in services. Users and devices are managed by these services. Published data is not a concern for current platforms and therefore not mentioned by them.

A Management component is introduced with the platform. This Management component is built to execute the defined authentication policies and to perform the access management as well. An access management system could be built with an authentication system, similiar to a PKI [46] or Kerberos [47]. The access management system has to be done in further work, because it would go beyond the scope of this thesis. In conclusion the security-, access- and user-management is held inside of the Management component and lists of objects and connected platforms are controlled by this component.

Proposed Architecture

In Figure 6.3 a proposed architecture of the Management component is shown. The registration of objects should be done at run-time with authentication mechanisms. Due to this, plug and play-solutions can be supported. If an object is accepted by a User or by a rule, then the object is signed in. Afterwards the handle for the Registry is sent to the previously accepted object.

Registered objects are available in the local Registry. These objects can also be shared with other Registries. Therefore the remote Management is connected by the local Management, to

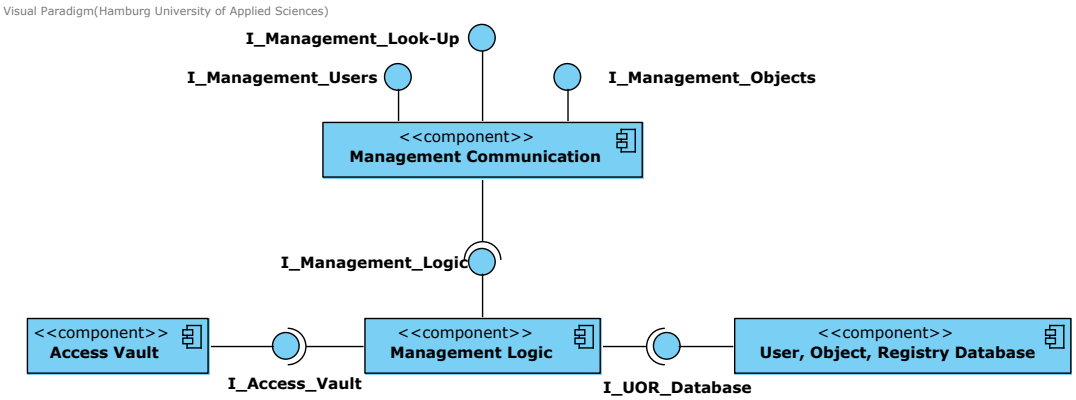

Figure 6.3.: Management component proposed architecture

get access to the remote Registry. After that, local objects can be offered to the remote Registry. These sequences are further described in Chapter 7.

The *I_Management_Look-Up* interface is used for look-ups of the internal state of the Management. Users and objects can be added to the platform via the *I_Management_Objects* and *I_Management_Users* interfaces.

The healthiness of the Registry should be monitored by the Management.

Discussion

Some advantages are introduced by the Management. For example the work-load of central instances can be reduced. Furthermore, the access management could be done by the Management component.

A disadvantage is, that in a decentralized system the access is granted and is valid for the time span that was defined before. The system could be improved in order to reduce that risk, by added access checks.

The risk of granted access which are declined later on, could be reduced by added access checks, which are performed by the objects, that would increase the amount of messages significantly.

6.4. Registry

A kind of handle-driven broker [44] is needed by an open and interoperable platform. The dynamic handling of entries is done by that broker. Entries in this context are defined in Chapter 4. A huge amount of object variants can be specified with the object description.

Therefore the object entries have to be treated as generic as possible.

A simple internal data/object representation is used by the other platforms. These representations are defined with an implicit understanding of the objects and produced data. In a monolithic system a handle-driven Registry is not needed. Only a mapping of data types is done by the Registry.

The Registry of the proposed platform is built as a handle-driven broker, like it is used in the Internet in the form of a DNS-Server [41]. Object entries are managed by their defined OID and a local_id which is defined at run-time. The object entries have to be as generic as possible. Therefore the Registry is built to handle the generic information, additional options are not handled yet. A handling of additional options should be introduced in further work.

The Registry is defined as a local entry-management component. Local entries can be shared by the Registry. In that case some defined entries are published to a specified remote Registry. The shared entries are organized in areas in the local Registry. If the local area is edited, then the entries are handled as described in the edit A.2.17 message.

The object entries have to be stored in an efficient way. There are two relevant types of databases. These databases are SQL- and NoSQL-databases. SQL-databases are classic relational databases. They have the advantage, if fixed data schemes are used. NoSQL-databases are available in different variations. Some of these variations are document-, key-value- and graph-databases. They all have their application area. Because of the tree-like OID numeration model, a graph-database is proposed. The selection of a use-case suitable database has to be done in subsequent observations. [61]

Proposed Architecture

In Figure 6.4 a proposed architecture of the Registry component is shown.

Authentication checks are done and the internal state is represented by the *Registry Logic* sub-component. The *Graph Database* sub-component is decoupled from the *Registry Logic* sub-component by the *Database Adapter* sub-component. By that the database can easier be replaced.

The interface *I_Registry_Areas* is used to setup and manage ares for shared objects and I_*Registry_Entries* is used to manage the local object entries. Look-up messages are processed via the *I_Registry_Look-Up* interface.

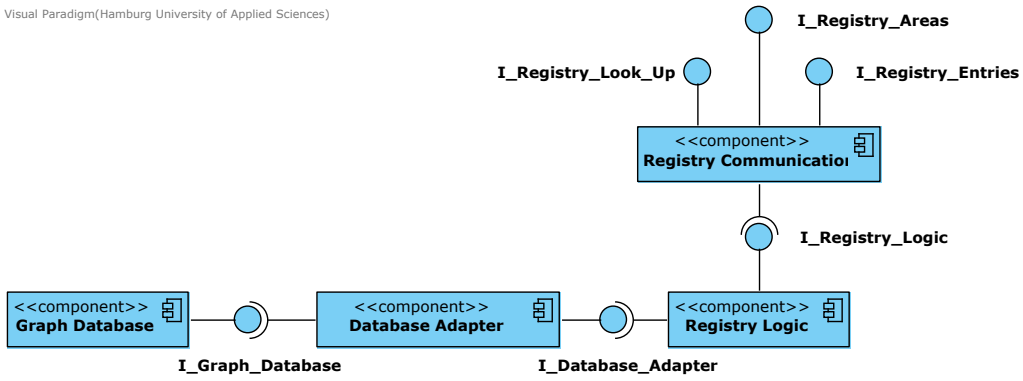

Figure 6.4.: Registry component proposed architecture

Discussion

The proposed Registry is defined as a component of an open platform. A loose coupling is an essential aspect of the proposed platform. The efficiency is also increased by the handle-driven approach. The dynamic abilities of the platform are increased by the possibility of fine granular areas in the Registry.

The Registry as the core component of the platform has to be very reliable and always available. Object context information are hold by the Registry. Because of that, the information has to be secured to protect additional context information.

6.5. Gateway

Todays networks are highly segmented. The network is segmented by global and local address spaces. Local address spaces are needed, because of the limitation of the IPv4 address space. IPv6 was introduced as the solution for limited IPv4-address space and simultaneously prepared the way for an end to end communication. But current networks are still divided. Outgoing traffic is resolved by Network-Address-Translation(NAT) [45] but for the incoming traffic a mechanism has to be found. NAT translates the local address to a global address. In some cases an end-to-end communication is not possible because of the segmented networks.

Central gateways are used by current systems. The gateway is reachable by a global address and is a part of the platform. Incoming messages are forwarded by the platform to the receiving component.

The proposed Gateway component uses the same mechanism as the existing technologies. The differences are, that the messages are forwarded by the Gateway component in the local network. The receiving component is identified by the global gateway address and the local address. The Gateway is designed as a stateless component. That can be done because no addresses have to be saved by the Gateway. Local addresses are sent within the gateway message.

Proposed Architecture

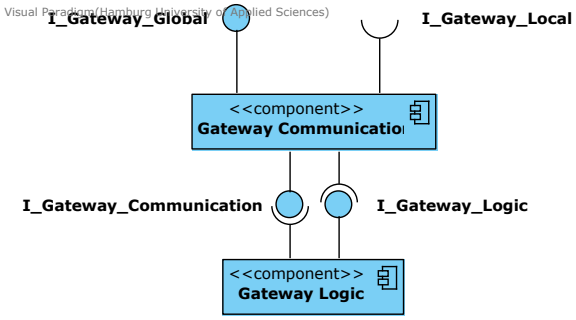

Figure 6.5.: Gateway component proposed architecture

In Figure 6.5 a proposed architecture of the Gateway component is shown.

The message forwarding is computed in the Gateway component. A message is sent to the global Interface, computed in the *Gateway Logic* sub-component and sent via the generic message interface. The generic message interface is defined as the message interface of all local reachable components.

Incoming messages are processed via the *I_Gateway_Global* interface and forwarded via the *I_Gateway_Private* interface.

Discussion

The advantages of this approach is, that the Gateway is stateless and therefore a scaling performance can be estimated. The scalability can be done by adding more instances. The incoming traffic can be distributed to all Gateway instances.

One of the disadvantages is, that the Gateway can be used to access the internal network. Furthermore the local network addresses are published to all Users that are connected to a public Registry. This component has a high need for security measures. The Gateway is used as a man in the middle to establish a connection. Thereby the property of an end-to-end

connection is unsatisfied. The current setting allows only one Gateway to deliver the message. This issue has be solved in subsequent observations.

6.6. Object Engine

Current IoT devices can be built in different shapes and have different functionalities. Thus all devices are built with their own interfaces. These interfaces are designed to fulfill an exact defined purpose. The semantics are provided by the manufacturer. That leads to a high range of interfaces.

Currently devices are connected via a SDK. The software for the IoT-devices is implemented using the platform's SDK. Because of this, the devices are bounded to the platform. Most times this process is static. The program can not be edited at run-time to change the device's purpose. The device's semantics are well-known and are simply described inside the platform and therefore are not further elaborated.

A well-defined Object Engine is proposed. The Object Engine should be seen as a device gateway. Different devices are bound to the platform by devices abstracted with an Object Engine. The in Section 3.3 defined communication pattern and the rules defined in Chapter 5 should be supported by the Object Engine. Furthermore an object implementation should be executable inside of the Object Engine. The object implementation should be bounded to the hardware via a Proxy [55]. That Proxy is called Hardware Abstraction Layer (HAL) in further discussions.

The Object Engine itself is an abstraction for different kinds of objects. It is provided with a well-defined interface. The same interface is used by every object in the system. The semantics of each object are represented by the object implementation, which for example is provided by the manufacturer.

The object implementation is an implementation of a description object, like it is defined in Chapter 4.

Proposed Architecture

In Figure 6.6 a proposed architecture of the Object Engine component is shown. The *ObjectEngine Control* sub-component should be used to control object implementations and their assigned HAL. The *Device HAL* sub-component should be used to bind the functionality, which is defined by the hardware, to the object implementation. The *ObjectEngine Object Runtime*

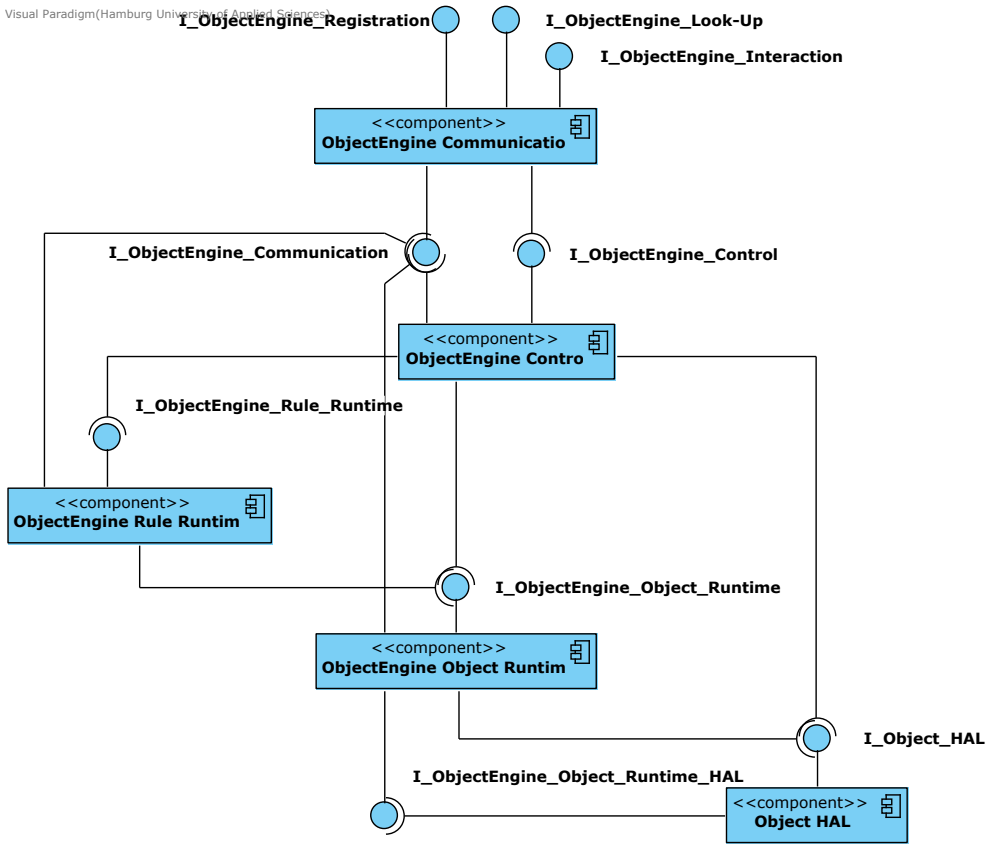

Figure 6.6.: Object Engine component proposed architecture

sub-component should be used to run the added object implementations. A common interface should be used by these object implementations to achieve a compatibility between the run-time environment and the object implementations. The *ObjectEngine Rule Runtime* sub-component should be used to run the defined and added rules. The actions defined in Section 5.3 should also be executed by the *Rule Runtime*. The *ObjectEngine Communication* sub-component is used to communicate with other platform components and Users of the platform.

Discussion

The Object Engine, which is defined in this thesis, is used as an IP-abstraction of a previously defined device. Due to this, several advantages, such as the use of the same interface by all objects within the system, are offered. Also, the object run-time is able to execute all kinds of compatible objects. New objects can be added at run-time, thereby the dynamic capabilities can be increased significantly. At the same time the Object Engine acts as a hardware abstraction

and as a result the IoT platform is decoupled from the actual devices by the Object Engine. Through the dynamic capabilities the possibility for updates at run-time is added.

But some disadvantages are also introduced by the proposed approach. Objects that are not capable of IP-communication are still connected via some kind of gateway. In this case an Object Engine is proposed, whose Object HAL is used to connect with devices which are using other protocols. An additional delay is added by the extra abstraction layer and in result of the extra abstraction more memory, energy and computing power is consumed.

6.7. Discussion

Some advantages are introduced by an open and interoperable system such as the proposed one. By the handle-driven approach the amount of sent messages is reduced in some scenarios. Dynamic capabilities are added by the Object Engine and the rules. A new kind of agility and decentralized computation and data preprocessing features are introduced by the agents. Furthermore object updates are possible at run-time. A system like the proposed platform is scalable because no central components are established. When a handle is requested, the handle can be used to communicate with the object directly. If the Registry is not reachable anymore. The object can further be used via the handle. That is a great improvement for decentralized systems.

However a centralized access management is not possible and some advanced mechanisms have to be used. An authentication with every component has to be used instead of one authentication with the platform. A centralized data store can not be achieved, but data-storage can be added by object implementations which are added to the Object Engines themselves. A great possibility is introduced with the agents but there have to be security measurements to check them and run them in a safe non harming way.

7. Communication Sequences

The proposed platform is defined as an interoperable IoT platform. The platform is designed as a distributed system and therefore all functionalities between the components have to be implemented with messages.

Theses messages in general are also used by the previously in Chapter 2 introduced and discussed IoT-platforms.

The messages that are sent within the context of the platform are JSON formatted. The advantages and disadvantages of JSON are discussed in Section 4.4. JSON is chosen because it is efficient, easily human readable and supported in several programming languages.

A basic message body is defined to have a common message organization.

7.1. Basic Message Body

Currently a version of a message body is used that allows only one type per message. The current message body version is shown in Listing 7.1. Several commands of the same type can be added to the **command** field. The commands can be added with a key that is called "entry_x" whereby x is incremented with every entry. The entries are sorted in the order they are added to the commands object. At the end the total amount of added entries is to be written to the field with the key "numOfEntries".

```
{
 "version": "0.1"
 "authentication": "example_authObject",
 "token": "example_tokenObject",
 "type": "example_type",
 "command": {
       "entries":{
         "entry_0": {
         }
       },
       "numOfEntries":1
 }
```

```
}
```

Listing 7.1: Basic message body version 0.1

The proposed new message body is created to take advantage of JSON's structural features. The new message body version is shown in Listing 7.2. Build in properties of JSON are now used. The **commands** are added to a JSON array which keeps the order of the entries by definition. The type now is bounded to the specified commands. Due to this definition, the entries can and should be executed in the order they are defined to allow ordered sequences of commands.

The new message body should be used in further works and papers.

```
{
  "version": "0.2"
  "authentication": "example_authObject",
  "token": "example_tokenObject",
  "commands": [
    {
      "example_type": {
        "description_of_example_Command": "example"
      }
    }
  ]
}
```

Listing 7.2: Basic message body version 0.2

7.2. Gateway Message Body

In local networks or if the other participant is directly reachable, the previously introduced basic message body can be used. If the devices are not directly reachable, then the message is sent, enriched with the Gateway message body. A message that is defined with a Gateway message body is send to the global Gateway address of the wanted component. The message is sent to the component that is defined in **destination** by the Gateway. The **address** and the **gateawayAddress** fields are defined for the case, that the communication scheme reactive 3.3 is used. If push is used, a new connection has to be established and therefore the information for Gateway messages have to be delivered. If IPv4 addresses with a corresponding port are used, the symbolic addresses should be formatted as "[ip-address]:port". That format is introduced in CoAPs URI scheme [62]. An example for that scheme would be "[127.0.0.1]:5000".

```
{
  "gatewayMessage": {
    "destination": "example_Destination_Address",
    "source": {
      "address": "example_Source_Address",
      "gatewayAddress": "example_Source_Gateway_Address"
    }
  }
}
```

Listing 7.3: Gateway message body version 0.1

7.3. Basic Sequences

In the following Sections message sequences for basic functionalities are defined. All used messages can be found in Appendix A.2 and are appended in brackets to each message type.

7.3.1. Heartbeat

In a distributed system, components are not integrated in a single process with shared memory by the same machine. Because of this limitation, the current state of a process can not be internally monitored by another process. A lightweight message is introduced to offer the possibility to check another part of the system. The check is simply a request and if a response is sent by the other part a correct function can be assumed but not be guaranteed.

The heartbeat sequence can be used to either check if the component is reachable or if the component is alive. If a **heartbeatResponse** (A.21) is sent back, than the component is defined by the term alive.

The messages that were used are listed as follows:

1. heartbeat (A.20)
2. heartbeatResponse (A.21)

7.3.2. Object Registration

If this sequence is used, objects can be added to a specific platform. The sequence starts with the message **registerObject** (A.42). The message is either sent directly to the Management, if the Management address is defined, or the message is sent via broadcast to the local network.

After the first message is sent all other messages of this sequence are sent directly to the specified destination.

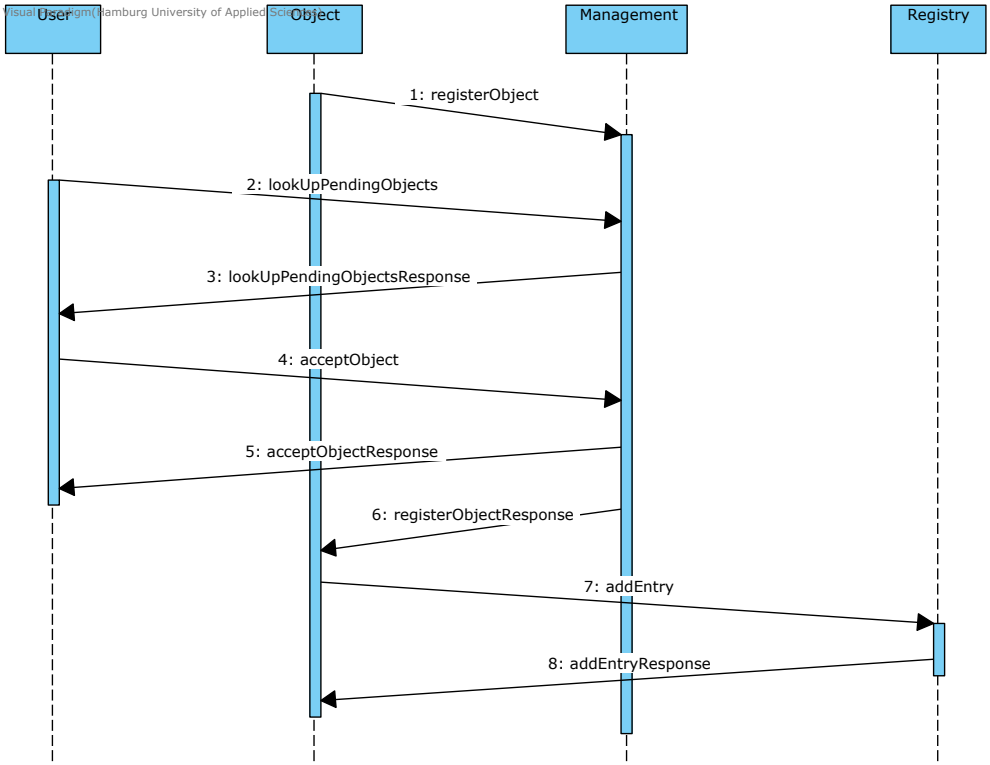

Figure 7.1.: Object registration sequence diagram

The object registration sequence is shown in Figure 7.1 and the messages that were used are listed as follows:

1. registerObject (A.42)
2. lookUpPendingObjects (A.29)
3. lookUpPendingObjectResponse (A.30)
4. acceptObject (A.1)
5. acceptObjectResponse (A.2)
6. registerObjectResponse (A.43)
7. addEntry (A.5)
8. addEntryResponse (A.6)

7.3.3. Register as Sub-Registry

The sequence that is specified in Figure 7.2 is used to establish a connection between two Registries. This means two different platforms are connected. In this relation, one Registry is the remote Registry and the other is the local Registry. During the sequence an area is defined in the local Registry. Objects, that are added to the specified area in the local Registry, are shared with the remote Registry. The process is triggered by the User. The **registerAsSubRegistry** (A.40) message is sent by the User to the local Management.

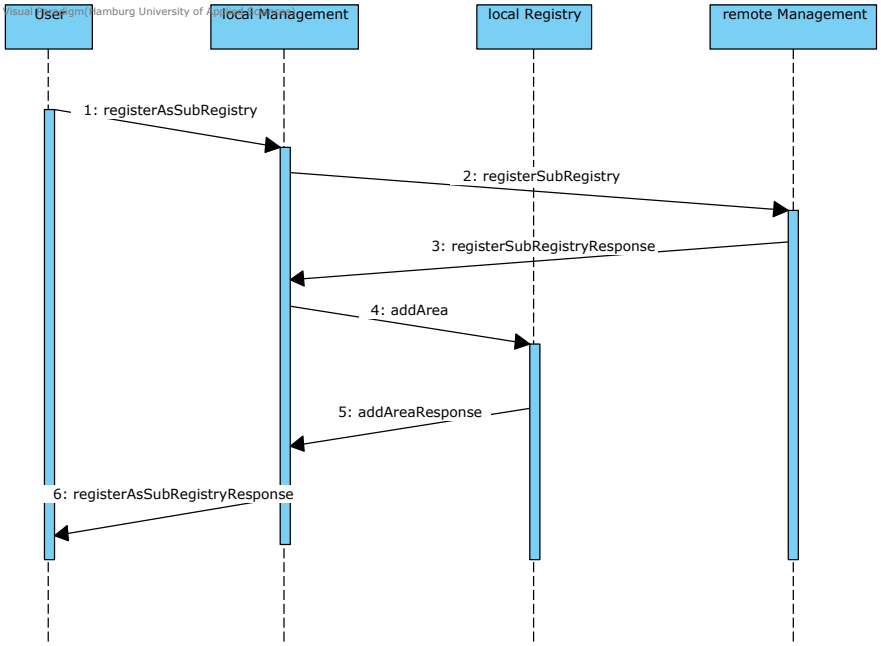

Figure 7.2.: Register as sub-Registry sequence diagram

1. registerAsSubRegistry (A.40)
2. registerSubRegistry (A.44)
3. registerSubRegistryResponse (A.45)
4. addArea (A.3)
5. addAreaResponse (A.4)
6. registerAsSubRegistryResponse (A.41)

7.3.4. Edit Registry Area

The edit Registry area sequence is used to edit shared objects, which are specified in an area in the local Registry. The possibility to easily share objects with other platforms is shown with the edit Registry sequence. That means new objects are added and deleted by this sequence. First the share status of the objects is altered in the area of the Registry and then the **addEntry** and **deleteEntry** messages are sent to the remote Registry. Both, objects to add and delete, are described in the message **editArea** A.17.

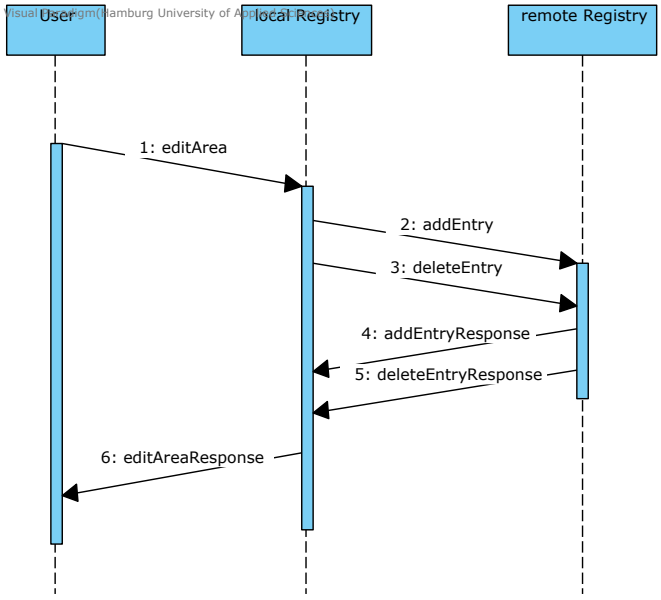

Figure 7.3.: Edit Registry area sequence diagram

The edit Registry area sequence is shown in Figure 7.3 and the messages that were used are listed as follows:

1. editArea (A.17)
2. addEntry (A.5)
3. deleteEntry (A.11)
4. addEntryResponse (A.6)
5. deleteEntryResponse (A.12)
6. editAreaResponse (A.18)

7.3.5. Object Interaction

The object interaction sequence is designed to show different kinds of interaction possibilities with objects.

The add/delete object rule sequence is used to edit the behavior of an object at run-time based on rules. These rules are described as defined in Chapter 5 and can be added and deleted.

At run-time for example measurement data has to be requested from an object. That object is designed to respond to the defined requests.

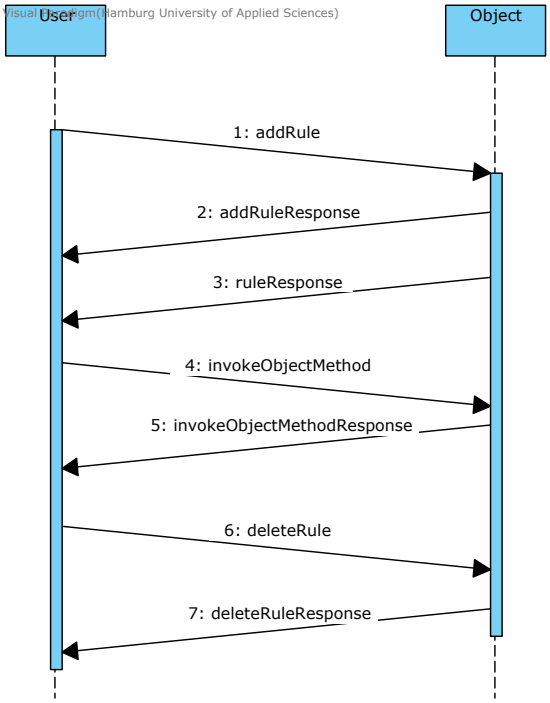

Figure 7.4.: Edit Registry area sequence diagram

The object interaction sequence is shown in Figure 7.4 and the messages that were used are listed as follows:

1. addRule (A.9)
2. addRuleResponse (A.10)
3. ruleResponse (A.46)
4. invokeObjectMethod (A.22)

5. invokeObjectMethodResponse (A.23)
6. deleteRule (A.15)
7. deleteRuleResponse (A.16)

7.3.6. Look-Up

The possibility to keep the overview of the status of the entries of the specified component is offered by the look-up sequences. Entries and functionalities are managed by the Management, the Registry and the Object Engine. The status of the functionalities and entries can be monitored with the later introduced look-up sequences. The messages which are proposed, are defined with a high granularity.

Management

The look-up sequences of the Management are used to keep track of different registered components. Objects and connected Registries are the registered components.
The look-up sequences of the Management are shown in Figure 7.5 and the messages that were used are listed as follows:

1. lookUpRegisteredAsSubRegistry (A.31)
2. lookUpRegisteredAsSubRegistryResponse (A.32)
3. lookUpRegisteredSubRegistry (A.35)
4. lookUpRegisteredSubRegistryResponse (A.36)
5. lookUpPendingObject (A.29)
6. lookUpPendingObjectResponse (A.30)
7. lookUpRegisteredObject (A.33)
8. lookUpRegisteredObjectResponse (A.34)

Registry

The look-up sequences of the Registry are used to keep track of registered objects and connected Registries. Furthermore the properties of the objects and Registries can be displayed.
The look-up sequences of the Registry are shown in Figure 7.6 and the messages that were used are listed as follows:

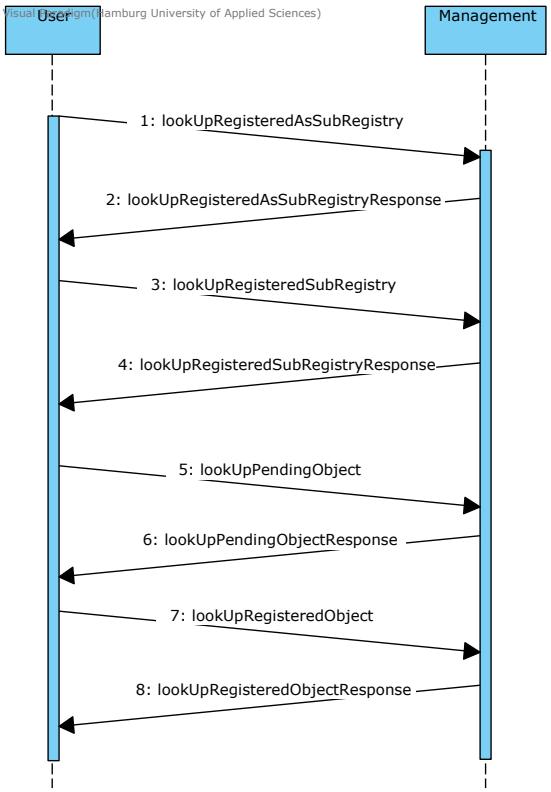

Figure 7.5.: Management look-up sequence diagram

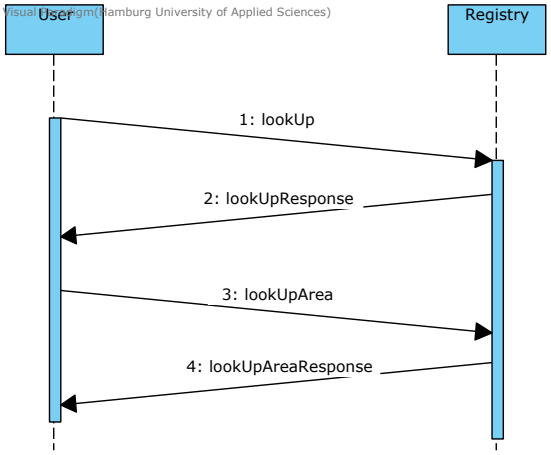

Figure 7.6.: Registry look-up sequence diagram

1. lookUp (A.24)

2. lookUpResponse (A.37)
3. lookUpArea (A.25)
4. lookUpAreaResponse (A.26)

Object Engine

The look-up sequences of the Object Engine are used to keep track of the managed objects. Furthermore the rules can be monitored by these sequences too.

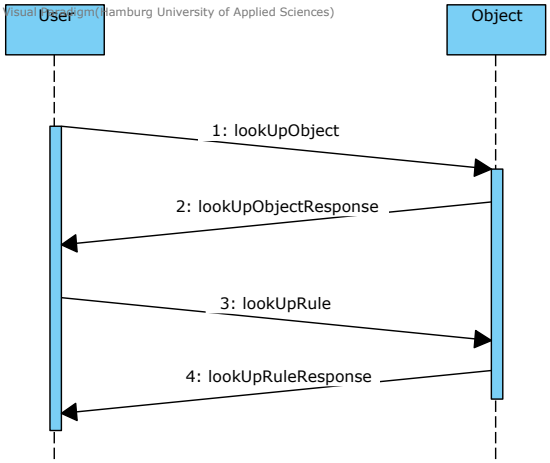

Figure 7.7.: Object look-up sequence diagram

The look-up sequences of the object are shown in Figure 7.7 and the messages that were used are listed as follows:

1. lookUpObject (A.27)
2. lookUpObjectResponse (A.28)
3. lookUpRule (A.38)
4. lookUpRuleResponse (A.39)

7.4. Discussion

The sequences that were presented in this Chapter are used to make the basic functionalities of the proposed platform available. The message bodies that are currently used have to be

improved and an improvement for the basic message body is already proposed and multiple Gateways have to be supported within the Gateway message body.

Also the proposed messages, which are shown in Appendix A.2, have to be tested in real-life scenarios and improved to fulfill as many use-cases as possible.

The look-up sequences are defined in a high granularity. The alternative is a generic look-up message, that defines the information that is wanted in an option field. In future work it has to be worked out, which granularity is appropriate for an efficient work-flow and which access is needed to retrieve information.

User information can not be retrieved with the currently defined look-up sequences. This functionality and the complete access management has to be added in future work. The Registry look-up sequences are currently built to give all provided information. Special search options and dependency options have to be added to allow an efficient and improved work-flow.

The defined sequences are also used as test routines for the components separately and also the collaboration of the components.

8. Platform Simulation

One goal of the proposed design is to create a more efficient system than the existing monolithic and centralized platforms. As a metric for efficiency the amount of messages that are sent is being evaluated. Following the idea that it is more efficient to send only one message instead of two or more.

This measurement can be done in two different ways. One way is to implement the proposed system as a real world test platform and the other way is to build a simulation model. A real world test platform comes with several difficulties. Since the measurements are done in an uncontrollable environment, the test results have to be handled with caution. If a test with more instances is built, there has to be enough computation power to run them. Also separated networks are difficult to establish.

A simulation has some advantages towards a real world test system. The simulation environment that is chosen has to generate reproducible results. Furthermore several instances of components have to be executable by it and networks have to be established easily.

Because of the advantages that are mentioned before, the usage of a simulation environment was chosen. The simulation was used firstly to measure the efficiency of the proposed system and secondly the messages and message sequences were checked for their correctness.

The messages and message sequences were tested and improved in an iterative process. The results are presented in Chapter 7 and in the Appendix A.2. Failures like missing arguments and misspelled keys were found while the simulation was used. Another part of the simulation was to evaluate needed messages. Namely some response messages were not defined or not tested before.

Due to the the limited time of this thesis, a more informative simulation scenario could not be found. For a more significant simulation more time is needed.

Thus the simulation of the distributed approach and a monolithic approach that uses the same message sequences was chosen. The differences between both simulation scenarios are further explained in Section 8.2.

8.1. Tool Chain

There are several different types of simulators. For the simulation of this platform a network simulator is chosen. A set of them is listed in the paper [63].

In that paper two relevant simulators are named. First the ns-3 simulator and second the OMNeT++ simulator. Both are discrete event simulators and a generic approach is pursued. Network simulations can be performed by both of them. Because OMNeT++ is used at the HAW Hamburg (Hochschule für Angewandte Wissenschaften Hamburg) by several workgroups, the OMNeT++ discrete event simulator is chosen for the platform simulation.

OMNeT++ is an object-oriented modular discrete event network simulation framework. A generic architecture is implemented, so it can be used for different problem domains. It can be used to model wired and wireless communication networks, model protocols, evaluate performance aspects of complex distributed systems and in general to model and simulate systems where the discrete event approach is suitable, and can be conveniently mapped into entities communicating by exchanging messages.

One of the fundamental parts of this infrastructure is a component architecture for simulation models. Models are put together from reusable components called modules. Modules can be connected with each other via gates (ports). Message passing is used by the modules to communicate and messages are passed by modules along predefined paths via gates and connections, or directly to their destination. Modules at the lowest level of the module hierarchy are called simple modules, and the model behavior is encased by them. The OMNeT++ hierarchy is shown in Figure 8.1. [3]

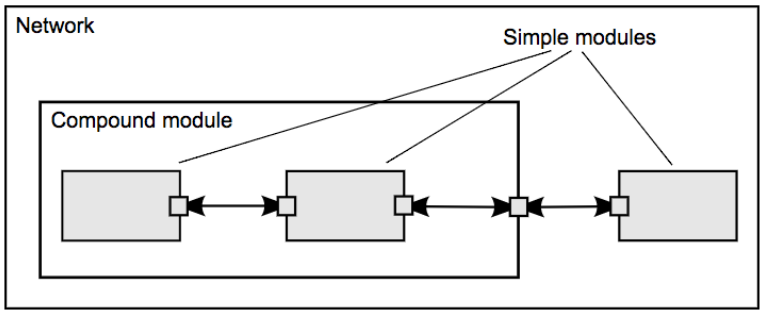

Figure 8.1.: OMNeT++ module hierarchy from [3]

The platform components that are defined in Chapter 6 are represented in the simulation as simple modules. The simple module is used as an interface for the component implementation. An example for the Registry is given in Figure 8.2. All components are connected via a self

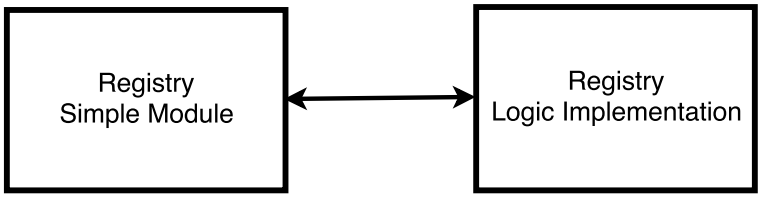

Figure 8.2.: OMNeT++ simple module implementation

defined simple module called router. All modules of a virtual network are connected to the router of the network. To connect networks the routers of different networks are connected with each other.

The address scheme that is used in this simulation is self defined. An address is defined by a network address and a module address. The network is identified by the network address and the host is identified by the module address. "Net1_Registry" is an example for an address like previously described. "Net1" is the network address and "Registry" is the address of the Registry simple module in the specified simulation. The network address and the module address are separated by the "_" sign.

The concept of the interoperable IoT platform is defined in the Chapters before. To evaluate the model and in order to do that the efficiency, the components of the platform are implemented in C++ [64]. C++ is the programming language that is used in OMNeT++. All platform components are connected via a nearly similar simple module. The messages which arrive at the simple module are dispatched by the message type. For the message type evaluation the Dispatcher pattern [65] is used. A common interface is provided by the Dispatcher. The messages are dispatched within the Dispatcher and then the corresponding methods are executed.

Due to time limitations, the functionalities are implemented in a very basic way. That means the implementation is not optimized in relation to computing power or anything else. The simulation implementation is built to be as adaptable as possible and as easy to work with.

8.2. Scenarios

Some of the previously in Section 7 defined sequences are tested and discussed in the subsequent Sections. Both scenarios are simulated under the following conditions.

- The same message-sequences are used in the monolithic and the distributed scenario.
- The messages are sent directly to the receiver, if the proposed platform is used.
- The messages are sent via the central platform, if a monolithic platform is used.
- The Gateway, Management and Registry are combined to one component in the monolithic approach and therefore no messages between those three components are measured.
- A message is sent for every request. The messages are not buffered and not sent together or not sent in smaller groups.
- Initial message rhythm.

  0s - 10s set-up time.

  10s - 30s 10 initial messages per second were sent with equally temporal distance.

  30s - 70s 50 initial messages per second were sent with equally temporal distance.

  70s - 110s 20 initial messages per second were sent with equally temporal distance.

8.2.1. Scenario Object Engine Interaction

The communication between a User and an Object Engine is chosen as a relevant scenario. In this scenario the Object Engine is already registered with the platform. The registration was done with sequence 7.3.2. The interaction that is measured, is done with messages described in sequence 7.3.5.

The message **invokeObjectMethod** A.2.22 is the initial message for the sequence. The messages are sent in the rhythms, that are described before. The address and the information of the Object Engine are prescribed and in order to that a look-up was not necessary.

In the diagram, that is shown in Figure 8.3, the results of the measurements, that were done in scenario of the Object Engine interaction, are presented.

The data was collected as an OMNeT++ vector, that was exported to a python array. The data in the array was evaluated and reprocessed. Afterwards the data was illustrated in the diagram.

The horizontal axis is defined as the time in seconds and the vertical axis is defined as messages per second. In the distributed line graph, four Sections can be seen. First, from zero seconds to ten seconds, second from ten seconds up to 30 seconds, third from 30 seconds up to 70 seconds and fourth from 70 seconds up to 110 seconds. In the subsequent description the amounts of messages per second, that are indicated by the distributed line graph, are summarized.

Section one nearly zero messages per second

Section two approximately 20 messages per second

Section three approximately 100 messages per second

Section four approximately 40 messages per second

In the monolithic line graph the same four time-sections can be seen. In the subsequent description the amounts of messages per second, that are indicated by the monolithic line graph, are summarized.

Section one nearly zero messages per second

Section two approximately 40 messages per second

Section three approximately 200 messages per second

Section four approximately 80 messages per second

Like it is mentioned before, the same four time-sections are shown by both line graphs. It can be seen, that the amount of messages per second is twice as high with the monolithic platform. The amount of messages with the monolithic platform is factor two higher as in the distributed approach.

The reason for the difference can be seen in the monolithic platform. With the monolithic platform all messages are sent not only between the User and the Object Engine, but the User to the central instance to the Object Engine. That behavior can be seen in Figure 8.4. With the distributed platform only path A (Figure 8.4) is used. In the monolithic platform the paths B and C (Figure 8.4) are used. Paths in this scenario are bidirectional connections between two components, that can be used to send a message.

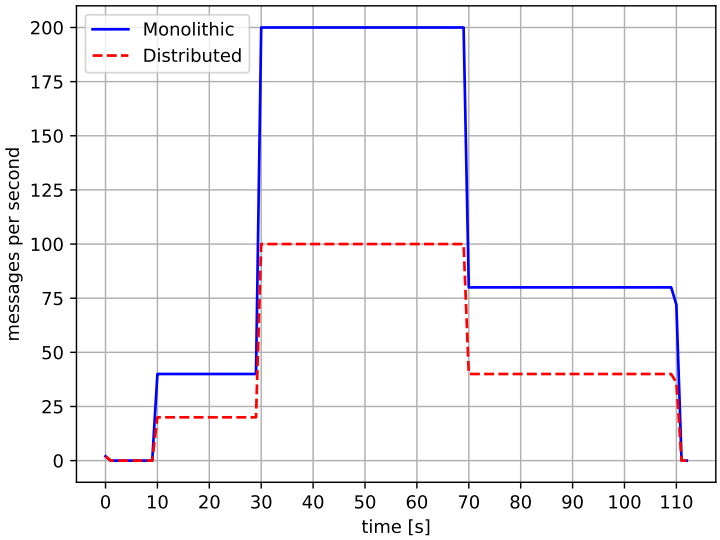

Figure 8.3.: Scenario Object Engine interaction measurements

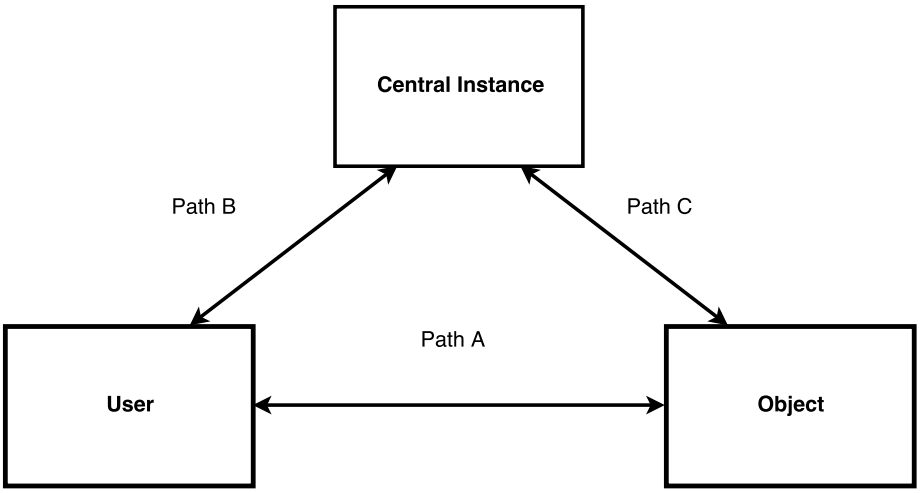

Figure 8.4.: Path usage in monolithic and distributed scenarios

8.2.2. Scenario Sub-Registry Registration

The registration as a sub-Registry is chosen as another relevant scenario.

The interaction that is measured, is done with messages described in sequence 7.3.3. The message **registerAsSubRegistry** A.2.40 is chosen to be the initial message for the sequence.

The messages are sent in the rhythms, that are described before. The address and the information of the Object Engine are prescribed and because of this a look-up was not necessary.

In the diagram, that is shown in Figure 8.5, the results of the measurements, that were done in the sub-Registry registration scenario, are described. The data was collected as an OMNeT++ vector, that was exported to a python array. The data in the array was evaluated and reprocessed. Afterwards the data was illustrated in the diagram. The horizontal axis is defined as the time in seconds and the vertical axis is defined as messages per second. In the distributed line graph four Sections can be seen. First, from zero seconds to ten seconds, second from ten seconds up to 30 seconds, third from 30 seconds up to 70 seconds and fourth from 70 seconds up to 110 seconds. In the subsequent description the amounts of messages per second, that are indicated by the distributed line graph, are summarized.

Section one nearly zero messages per second

Section two approximately 60 messages per second

Section three approximately 300 messages per second

Section four approximately 120 messages per second

In the monolithic line graph the same four time-sections can be seen. In the subsequent description the amounts of messages per second, that are indicated by the monolithic line graph, are summarized.

Section one nearly zero messages per second

Section two approximately 40 messages per second

Section three approximately 200 messages per second

Section four approximately 80 messages per second

Like it is mentioned before, the same four time-sections are shown by both line graphs. It can be seen, that the amount of messages per second is only two thirds with the monolithic platform. The amount of messages with the distributed platform is factor one and a half higher then with the monolithic platform.

The amount of messages that was measured with the distributed approach is higher. This is the result of the distribution of the the components Management and Registry. With the monolithic platform those messages are part of the internal communication and are therefore not taken into account.

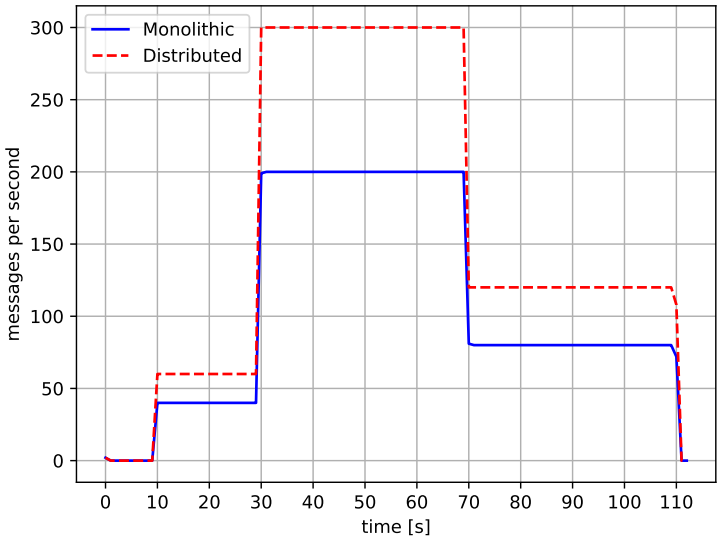

Figure 8.5.: Scenario Object Engine registration measurements

8.3. Discussion

In both scenarios a different behavior of the two platform constructions can be compared. But two different assumptions can be drawn from these two scenarios. In the first scenario of the distributed platform, only half the messages are needed for the same behavior. In the second scenario, the monolithic platform is proofed to be more efficient, because only two thirds of the messages are needed.

So the efficiency of the distributed and importantly handle-driven approach is not more efficient in every case. The distributed approach is equally or more efficient when a scenario is chosen that includes a direct communication with a component. If a communication is directly between a User and an Object Engine, then the efficiency is increased. If a scenario is chosen that includes communication between a Management, a Registry or a Gateway of the same local platform, then the efficiency in the monolithic platform is obviously higher.

It is more likely that the bigger part of the communication within a platform is used for the communication between Object Engines and Users. Registrations of sub-Registries for example are estimated not to happen so often in comparison to the device Object Engine communication.

Although the efficiency is increased by the message scheme, some disadvantages are included with the distributed platform approach.

First, the state of all components can not be monitored internally. That has to be done with other mechanisms, that should be introduced in further work.

Second, if an Object Engine is requested by several Users, then a big work-load can be imposed. The work-load for a small component is harder to handle, than for the monolithic platform. So the Object Engines should either be able to handle that work-load or mechanisms for a work-load balancer should be introduced.

Third, a centralized data-storage for the data is not part of the distributed approach. The data storage should be added directly to every Object Engine. Through this the distribution is further advanced.

There are some next steps that have to be discussed and done.

The simulation should be improved with IP-communication interfaces. Because of this improvement the simulation could be used to evaluate the function of other components. Also a real component could be tested against and with simulation components.

Furthermore to ease the test and evaluation process, test routines should be further defined and implemented in an appropriate environment.

9. Conclusion

A concept for an interoperable IoT-platform is introduced by this thesis. An object description scheme, a rule based behavior and different interaction schemes have also been defined. The proposed platform is built to support the Management of objects that are described by the early mentioned and introduced description scheme 4. Some advantages and disadvantages of the proposed platform are presented and discussed earlier. The proposed platform has to be seen as a work in progress approach.

A central component that handles all upcoming traffic is not needed. A direct communication paradigm is introduced by this platform. Due to this the messages which are sent between the components are reduced in total and the workload of central components is significantly reduced. The platform is managed by two central components. The Management is used to manage users, connections to other platforms and objects. The Registry is used to save the objects and their additional information. Also the shared objects are exchanged between different platforms by the Registry. A central database for historical measurement data is not implemented due to the decentralized approach.

The error rate can be reduced by this approach. Because no central instance is implemented, the traffic between the User and the object can not be interrupted. Furthermore an object that was earlier found via the Registry, can be used even if the Management and the Registry are not reachable later on.

The platform is designed to work in IP-networks. Based on the limitations of IP-networks the Quality of Service (QoS) is described as a best-effort platform. Some limitations can be overcome by integrated mechanisms like responses.

By the focus on IP-networks many IoT-devices can not be directly connected to the platform. In cases like that, an Object Engine has to be used as a gateway for such devices. In this case the connected devices are defined in terms of the proposed platform and offered to the Registry by the Object Engine. If a device is capable of IP but is limited otherwise, the same scenario

with an Object Engine as a gateway is applied.

Several use-cases are supported by the open architecture of the proposed platform. In particular the interoperable design with an easy way to share objects is a benefit and is needed in many use-cases. But the open architecture in its current state is not suitable for use-cases that strictly need a closed and secured systems. Issues concerning the security are discussed in the following Section.

9.1. Future Work

The proposed platform should be built to support IPv4 and IPv6. IPv6 is an essential part of the IoT development and is therefore a must. But, the concept is not limited to IP-networks and can be adapted for other communication media. With the focus on IP-networks the QoS is limited to best-effort. Because of that the support of other transmission technologies should be examined.

All kinds of optional and additional information should be managed by the Registry. These information should be used to filter the objects in the look-up scenarios. Thereby the efficiency of the communication can be further improved.

A central database for objects is not part of the proposed platform. Due to this fact a centralized data collection and storage is not supported. A mechanism to solve this issue would be a decentralized database, added to the Object Engine. Therefore a specification for historical data requests is needed.

Furthermore, strategies for connection checks are essential. The heartbeat message (A.2.20) and the corresponding sequence was introduced in Chapter 7.3.1. The heartbeat should be used to check the reachability and healthiness of the components in the system. The own IP-address should be monitored and in case of a change, the new address should be published. Currently a address reconfiguration message is not proposed but should be introduced. A work-around is to delete the object with the old address and add the object with the new address to the Registry.

The Gateway component was introduced as a stateless gateway for messages and in this case Gateway messages. The Gateway concept needs to be tested in a productive scenario. A change of the messages and the concept may be required.

The communication patterns request-response and reactive are well defined. The stream pattern needs to be worked out further. The data has to be described further and stream

capabilities have to be defined. A stream capability for example with which the rate data is sent and processed.

The proposed description language is currently defined in JSON but has to be defined in XML in further work. XML has the advantage of the validation feature. That is a huge advantage in descriptions. With this feature it can be tested, if the description is technically fine. Because the description is defined once and is proposed to be locally stored, the higher extent of XML is not that important. Furthermore XML is already used in several object serialization technologies and should therefore be fitted to be used in this description scheme.

The rules like they are introduced, have to be tested further. Challenges in productive environments have not been discovered yet.

The advanced look-ip messages were mentioned before. These messages have to be adapted, when the request scheme is defined. Also a support of sequence numbers should be introduced, to be able to filter messages by their order. Another topic for further discussions are transactions. Another point that should be considered is a message checksum and a corresponding validation mechanism. Like mentioned in the Section before, the security infrastructure has to be implemented and tested.

Currently there is no security strategy implemented. Before a security mechanism is introduced and implemented, the system must not be used in productive systems with safety-critical applications.

In order to propose a security mechanism, a requirements analysis should be done. Afterwards, in further work, a security system should be proposed.

An interoperable IoT platform is presented in this thesis. Some existing techniques are referenced to the design. Mechanisms like vocabularies, rules and actual communication pattern are used. The description is further improved by a new object description language.

In order to use the new capabilities efficiently, a handle-driven platform design is proposed. Four components are the core parts of the platform. Each is defined to serve a specific purpose.

The efficiency of the communication, the dynamic behavior of objects, the interoperability between platforms and the context awareness of objects is improved by this approach.

A. Appendix

A.1. Object Description

A.2. Messages

The messages which are shown in the following sections are sorted in alphabetical order. Furthermore these messages should be seen as examples. Some of theses messages need to be redefined in future work.

A.2.1. acceptObject

```
{
    "command": {
        "entries": {
            "entry_0": {
                "accept": true,
                "objectId": "0"
            }
        },
        "numOfEntries": 1
    "type": "acceptObject",
}
```

Listing A.1: acceptObject Message

A.2.2. acceptObjectResponse

```
{
    "command": {
        "entries": {
            "entry_0": {
                "request": {
                    "accept": true,
```

```
                    "objectId": "0"
                },
                "successful": true
            }
        }
    "type": "acceptObjectResponse",
}
```

Listing A.2: acceptObjectResponse Message

A.2.3. addArea

```
{
    "command": {
        "entries": {
            "entry_0": {
                "options": {},
                "registryId": "3",
                "remoteRegistry": {
                    "address": "Net1_Registry2",
                    "gatewayAddress": "Net4_GlobalGateway"
                },
                "requestId": "1",
                "token": "test_grantedToken_Net3_Management3"
            }
        },
        "numOfEntries": 1
    "type": "addArea",
}
```

Listing A.3: addArea Message

A.2.4. addAreaResponse

```
{
    "command": {
        "entries": {
            "entry_0": {
                "areaId": "1",
                "remoteRegistry": {
                    "address": "Net1_Registry2",
                    "gatewayAddress": "Net4_GlobalGateway"
```

```
                },
                "requestId": "1",
                "successful": true
            }
        },
        "numOfEntries": 1
    "type": "addAreaResponse",
}
```

Listing A.4: addAreaResponse Message

A.2.5. addEntry

```
{
    "command": {
        "entries": {
            "entry_0": {
                "address": "exampleAddress_Object",
                "oid": "1.2.3.41.213",
                "options": {},
                "requestId": "1"
            }
        },
        "numOfEntries": 1
    "type": "addEntry",
}
```

Listing A.5: addEntry Message

A.2.6. addEntryResponse

```
{
    "command": {
        "entries": {
            "entry_0": {
                "localId": "3",
                "requestId": "1",
                "successful": true
            }
        },
        "numOfEntries": 1
    "type": "addEntryResponse",
```

```
}
```

Listing A.6: addEntryResponse Message

A.2.7. addObject

```
{
    "command": {
        "entries": {
            "entry_0": {
                "HALAddress": "example_HALAddress",
                "HALToken": "example_HALToken",
                "oid": "1.2.3.41.213",
                "options": {},
                "repository": "objectImplementationRepository_URL",
                "tempObjectId": "1123.temp"
            }
        },
        "numOfEntries": 1
    "type": "addObject",
}
```

Listing A.7: addObject Message

A.2.8. addObjectResponse

```
{
    "command": {
        "entries": {
            "entry_0": {
                "successful": true,
                "tempObjectId": "1123.temp"
            }
        },
        "numOfEntries": 1
    "type": "addObjectResponse",
}
```

Listing A.8: addObjectResponse Message

A.2.9. addRule

```
{
```

```
    "command": {
        "entries": {
            "entry_0": {
                "action": {
                    "method": {
                        "name": "User2_Test_Invocation",
                        "parameters": {
                            "parameter_0": 10,
                            "parameter_1": true,
                            "parameter_2": "Test_String_Parameter"
                        }
                    },
                    "receiver": {
                        "address": "Net1_User2",
                        "auth": "User2_test_auth",
                        "token": "User2_test_token",
                        "type": "other"
                    },
                    "respond": true,
                    "respondTo": {
                        "address": "Net3_User2",
                        "gatewayAddress": "Net4_GlobalGateway2"
                    },
                    "type": "method"
                },
                "interval": 1000,
                "rule_type": "periodic"
            }
        },
        "numOfEntries": 1
    "type": "addRule",
}
```

Listing A.9: addRule Message

A.2.10. addRuleResponse

```
{
    "command": {
        "entries": {
            "entry_0": {
```

```
                "request": {
                    "action": {
                        "method": {
                            "name": "User2_Test_Invocation",
                            "parameters": {
                                "parameter_0": 10,
                                "parameter_1": true,
                                "parameter_2": "Test_String_Parameter"
                            }
                        },
                        "receiver": {
                            "address": "Net1_User2",
                            "auth": "User2_test_auth",
                            "token": "User2_test_token",
                            "type": "other"
                        },
                        "respond": true,
                        "respondTo": {
                            "address": "Net3_User2",
                            "gatewayAddress": "Net4_GlobalGateway2"
                        },
                        "type": "method"
                    },
                    "interval": 1000,
                    "rule_type": "periodic"
                },
                "ruleId": "0",
                "successful": true
            }
        },
        "numOfEntries": 1
    "type": "addRuleResponse",
}
```

Listing A.10: addRuleResponse Message

A.2.11. deleteEntry

```
{
    "command": {
        "entries": {
```

```
            "entry_0": {
                "localId": "0"
            }
        },
        "numOfEntries": 1
    "type": "deleteEntry",
}
```

Listing A.11: deleteEntry Message

A.2.12. deleteEntryResponse

```
{
    "command": {
        "entries": {
            "entry_0": {
                "localId": "0",
                "successful": true
            }
        },
        "numOfEntries": 1
    "type": "deleteEntryResponse",
}
```

Listing A.12: deleteEntryResponse Message

A.2.13. deleteObject

```
{
    "command": {
        "entries": {
            "entry_0": {
                "localId": "0"
            }
        },
        "numOfEntries": 1
    "type": "deleteObject",
}
```

Listing A.13: deleteObject Message

A.2.14. deleteObjectResponse

```
{
    "command": {
        "entries": {
            "entry_0": {
                "localId": "0",
                "successful": true
            }
        },
        "numOfEntries": 1
    "type": "deleteObjectResponse",
}
```

Listing A.14: deleteObjectResponse Message

A.2.15. deleteRule

```
{
    "command": {
        "entries": {
            "entry_0": {
                "ruleId": "0"
            }
        },
        "numOfEntries": 1
    "type": "deleteRule",
}
```

Listing A.15: deleteRule Message

A.2.16. deleteRuleResponse

```
{
    "command": {
        "entries": {
            "entry_0": {
                "ruleId": "0",
                "successful": true
            }
        },
        "numOfEntries": 1
    "type": "deleteRuleResponse",
```

```
}
```

Listing A.16: deleteRuleResponse Message

A.2.17. editArea

```
{
    "command": {
        "entries": {
            "entry_0": {
                "areaId": "1",
                "options": {
                    "add": {
                        "localIds": []
                    },
                    "remove": {
                        "localIds": [
                            "3_0"
                        ]
                    }
                }
            }
        },
        "numOfEntries": 1
    "type": "editArea",
}
```

Listing A.17: editArea Message

A.2.18. editAreaResponse

```
{
    "command": {
        "entries": {
            "entry_0": {
                "areaId": "1",
                "successful": true
            }
        },
        "numOfEntries": 1
    "type": "editAreaResponse",
```

```
}
```

Listing A.18: editAreaResponse Message

A.2.19. gatewayAdvertisement

```
{
    "command": {
        "globalAddress": "Net4_GlobalGateway2"
    "type": "gatewayAdvertisement",
}
```

Listing A.19: gatewayAdvertisement Message

A.2.20. heartbeat

```
{
    "command": {},
    "type": "heartbeat",
}
```

Listing A.20: heartbeat Message

A.2.21. heartbeatResponse

```
{
    "command": {},
    "type": "heartbeatResponse",
}
```

Listing A.21: heartbeatResponse Message

A.2.22. invokeObjectMethod

```
{
    "command": {
        "entries": {
            "entry_0": {
                "localId": "test_requestId",
                "methodName": "global User test invocation",
                "parameters": {
                    "1": "test"
                },
                "responseId": "Net1_Management2"
```

```
            }
        },
        "numOfEntries": 1
    "gatewayMessage": {
        "destination": "Net1_Object2",
        "source": {
            "address": "Net4_GlobalUser"
        }
    "type": "invokeObjectMethod",
}
```

Listing A.22: invokeObjectMethod Message

A.2.23. invokeObjectMethodResponse

```
{
    "command": {
        "entries": {
            "entry_0": {
                "responseId": "testResponseId",
                "returnValue": {},
                "successful": true
            }
        },
        "numOfEntries": 1
    "type": "invokeObjectMethodResponse",
}
```

Listing A.23: invokeObjectMethodResponse Message

A.2.24. lookUp

```
{
    "command": {
        "limit": 500,
        "oids": [
            "1.2.3"
        ]
    "type": "lookUp",
}
```

Listing A.24: lookUp Message

A.2.25. lookUpArea

```
{
    "command": {
        "limit": 500
    "type": "lookUpArea",
}
```

Listing A.25: lookUpArea Message

A.2.26. lookUpAreaResponse

```
{
    "command": {
        "entries": {
            "1": {
                "objects": {
                    "0": {
                        "localId": "0",
                        "registeredInRemoteRegistry": false,
                        "removeRequestedInRemoteRegistry": false
                    }
                },
                "options": {
                    "add": {
                        "localIds": []
                    },
                    "remove": {
                        "localIds": [
                            "3_0"
                        ]
                    }
                },
                "registryId": "3",
                "remoteRegistry": {
                    "address": "Net1_Registry2",
                    "gatewayAddress": "Net4_GlobalGateway"
                },
                "requestId": "1",
                "token": "test_grantedToken_Net3_Management3"
            }
```

```
        }
    "type": "lookUpAreaResponse",
}
```

Listing A.26: lookUpAreaResponse Message

A.2.27. lookUpObject

```
{
    "command": {
        "limit": 5
    "type": "lookUpObject",
}
```

Listing A.27: lookUpObject Message

A.2.28. lookUpObjectResponse

```
{
    "command": {
        "limit": 5,
        "pendingObjects": {},
        "registeredObjects": {
            "3": {
                "HALAddress": "example_HALAddress",
                "HALToken": "example_HALToken",
                "localId": "3",
                "oid": "1.2.3.4.Object1",
                "options": {},
                "repository": "objectImplementationRepository_URL",
                "requestId": "1",
                "tempObjectId": "1123.temp"
            }
        }
    "type": "lookUpObjectResponse",
}
```

Listing A.28: lookUpObjectResponse Message

A.2.29. lookUpPendingObject

```
{
    "command": {
```

```
        "limit": 500
    "type": "lookUpPendingObject",
}
```

Listing A.29: lookUpPendingObject Message

A.2.30. lookUpPendingObjectResponse

```
{
    "command": {
        "entries": {
            "0": {
                "address": "Net3_Object3",
                "authentication": "default_authObject",
                "gatewayAddress": "Net4_GlobalGateway2",
                "localId": "0",
                "oid": "1.2.3.4.5.Object3",
                "requestId": "Object301",
                "token": "token_device0"
            }
        }
    "type": "lookUpPendingObjectResponse",
}
```

Listing A.30: lookUpPendingObjectResponse Message

A.2.31. lookUpRegisteredAsSubRegistry

```
{
    "command": {
        "limit": 5
    "type": "lookUpRegisteredAsSubRegistry",
}
```

Listing A.31: lookUpRegisteredAsSubRegistry Message

A.2.32. lookUpRegisteredAsSubRegistryResponse

```
{
    "command": {
        "entries": {
            "2": {
                "areaId": "2",
```

```
                "authentication": "test_authentication",
                "remoteManagement": {
                    "address": "Net1_Management2"
                },
                "remoteRegistry": {
                    "address": "Net1_Registry2",
                    "gatewayAddress": "Net4_GlobalGateway"
                },
                "requestId": "test_requestId",
                "requestSource": {
                    "address": "Net1_User"
                },
                "token": "default_tokenObject"
            }
        }
    "type": "lookUpRegisteredAsSubRegistryResponse",
}
```

Listing A.32: lookUpRegisteredAsSubRegistryResponse Message

A.2.33. lookUpRegisteredObject

```
{
    "command": {
        "limit": 500
    "type": "lookUpRegisteredObject",
}
```

Listing A.33: lookUpRegisteredObject Message

A.2.34. lookUpRegisteredObjectResponse

```
{
    "command": {
        "entries": {
            "0": {
                "address": "Net3_Object3",
                "authentication": "default_authObject",
                "gatewayAddress": "Net4_GlobalGateway2",
                "localId": "0",
                "oid": "1.2.3.4.5.Object3",
                "requestId": "Object301",
```

```
                "token": "token_device0"
            }
        }
    "type": "lookUpRegisteredObjectResponse",
}
```

Listing A.34: lookUpRegisteredObjectResponse Message

A.2.35. lookUpRegisteredSubRegistry

```
{
    "command": {
        "limit": 5
    "type": "lookUpRegisteredSubRegistry",
}
```

Listing A.35: lookUpRegisteredSubRegistry Message

A.2.36. lookUpRegisteredSubRegistryResponse

```
{
    "command": {
        "entries": {
            "2": {
                "authentication": "test_authentication",
                "registryId": "2",
                "token": "default_tokenObject"
            },
            "3": {
                "authentication": "test_authentication",
                "registryId": "3",
                "token": "default_tokenObject"
            }
        }
    "type": "lookUpRegisteredSubRegistryResponse",
}
```

Listing A.36: lookUpRegisteredSubRegistryResponse Message

A.2.37. lookUpResponse

```
{
    "command": {
```

```
        "entries": {
            "1.2.3": [
                {
                    "address": "Net1_Object1",
                    "gatewayAddress": "Net4_GlobalGateway",
                    "localId": "2_0",
                    "oid": "1.2.3.4.Object1",
                    "options": {}
                },
                {
                    "address": "Net3_Object3",
                    "gatewayAddress": "Net4_GlobalGateway2",
                    "localId": "3_0",
                    "oid": "1.2.3.4.5.Object3",
                    "options": {}
                }
            ]
        }
    "type": "lookUpResponse",
}
```

Listing A.37: lookUpResponse Message

A.2.38. lookUpRule

```
{
    "command": {
        "limit": 5
    "type": "lookUpRule",
}
```

Listing A.38: lookUpRule Message

A.2.39. lookUpRuleResponse

```
{
    "command": {
        "entries": {}
    "type": "lookUpRuleResponse",
}
```

Listing A.39: lookUpRuleResponse Message

A.2.40. registerAsSubRegistry

```
{
    "command": {
        "entries": {
            "entry_0": {
                "authentication": "test_authentication",
                "remoteManagement": {
                    "address": "Net1_Management2",
                    "gatewayAddress": "Net4_GlobalGateway"
                },
                "requestId": "test_requestId",
                "token": "default_tokenObject"
            }
        },
        "numOfEntries": 1
    "type": "registerAsSubRegistry",
}
```

Listing A.40: registerAsSubRegistry Message

A.2.41. registerAsSubRegistryResponse

```
{
    "command": {
        "entries": {
            "entry_0": {
                "areaId": "1",
                "remoteRegistry": {
                    "address": "Net1_Registry2",
                    "gatewayAddress": "Net4_GlobalGateway"
                },
                "requestId": "1",
                "successful": true,
                "token": "default_tokenObject"
            }
        },
        "numOfEntries": 1
    "type": "registerAsSubRegistryResponse",
}
```

Listing A.41: registerAsSubRegistryResponse Message

A.2.42. registerObject

```
{
    "command": {
        "entries": {
            "entry_0": {
                "address": "Net3_Object3",
                "authentication": "default_authObject",
                "gatewayAddress": "Net4_GlobalGateway2",
                "oid": "1.2.3.4.5.Object3",
                "requestId": "Object301"
            }
        },
        "numOfEntries": 1
    "type": "registerObject",
}
```

Listing A.42: registerObject Message

A.2.43. registerObjectResponse

```
{
    "command": {
        "entries": {
            "entry_0": {
                "localId": "0",
                "registry": {
                    "address": "Net3_Registry3",
                    "gatewayAddress": "Net4_GlobalGateway2"
                },
                "requestId": "Object301",
                "successful": true,
                "token": "token_device0"
            }
        },
        "numOfEntries": 1
    "type": "registerObjectResponse",
}
```

Listing A.43: registerObjectResponse Message

A.2.44. registerSubRegistry

```
{
    "command": {
        "entries": {
            "entry_0": {
                "authentication": "test_authentication",
                "requestId": "1",
                "token": "default_tokenObject"
            }
        },
        "numOfEntries": 1
    "gatewayMessage": {
        "destination": "Net1_Management2",
        "source": {
            "address": "Net3_Management3",
            "gatewayAddress": "Net4_GlobalGateway2"
        }
    "type": "registerSubRegistry",
}
```

Listing A.44: registerSubRegistry Message

A.2.45. registerSubRegistryResponse

```
{
    "command": {
        "entries": {
            "entry_0": {
                "registryId": "3",
                "remoteRegistry": {
                    "address": "Net1_Registry2",
                    "gatewayAddress": "Net4_GlobalGateway"
                },
                "requestId": "1",
                "successful": true,
                "token": "test_grantedToken_Net3_Management3"
            }
        },
        "numOfEntries": 1
    "gatewayMessage": {
        "destination": "Net3_Management3",
        "source": {
```

```
            "address": "Net1_Management2",
            "gatewayAddress": "Net4_GlobalGateway"
        }
    "type": "registerSubRegistryResponse",
}
```

Listing A.45: registerSubRegistryResponse Message

A.2.46. ruleResponse

```
{
    "command": {
        "auth": "User_test_auth",
        "returnValue": 0,
        "ruleId": "0",
        "token": "User_test_token"
    "type": "ruleResponse",
}
```

Listing A.46: ruleResponse Message

Bibliography

[1] Kaa. *IoT Use Cases*. URL: https://www.kaaproject.org/iot-use-cases/ (visited on 04/13/2017).

[2] Hoan-Suk Choi, Deok-Hee Kang, and Woo-Seop Rhee. *RISE: Role-based Internet of Things Service Environment*. Conference Paper. Department of Multimedia Engineering Hanbat National University 34158 Daejeon Korea, 2016.

[3] OMNeT++. *Simulation Manual OMNeT++ version 5.1.1*. URL: https://omnetpp.org/doc/omnetpp/manual/#sec:introduction:what-is-omnetpp (visited on 09/27/2017).

[4] Thomas Fischer. *Internet of Things-Buzzwords: Das bedeuten die Schlagwörter*. URL: https://www.expertenderit.de/blog/internet-of-things-buzzwords-das-bedeuten-die-schlagwoerter?hsCtaTracking=6d6f65d8-0d01-425c-8f98-e32d36d4f13c%7C1ec95a02-5646-43e0-87d2-51c2c320a19a (visited on 07/11/2017).

[5] Karen Rose, Scott Eldridge, and Lyman Chapin. *The Internet of Things: An Overview*. Tech. rep. The Internet Society (ISOC), 2015.

[6] Living Internet. *The Internet Toaster*. 2000. URL: http://www.livinginternet.com/i/ia_myths_toast.htm (visited on 07/11/2017).

[7] Cisco. *Complete Visual Networking Index (VNI) Forecast*. URL: http://www.cisco.com/c/en/us/solutions/service-provider/visual-networking-index-vni/index.html?stickynav=1 (visited on 07/11/2017).

[8] Tony Danova. *Morgan Stanley: 75 Billion Devices Will Be Connected To The Internet Of Things By 2020*. URL: http://www.businessinsider.com/75-billion-devices-will-be-connected-to-the-internet-by-2020-2013-10?IR=T (visited on 07/11/2017).

[9] Ltd. Huawei Technologies Co. *Huawei's Heavy Investment in Five IoT Solutions Leads to Impressive Breakthroughs*. URL: http://www.huawei.com/en/news/2016/4/wuda-IoT-jiejue-fangan (visited on 07/11/2017).

[10] James Manyika et al. *The Internet of Things: Mapping the Value Beyond the Hype*. Tech. rep. McKinsey Global Institute, 2015.

[11] Philips. *Philips Hue*. URL: https://www.philips.de/c-p/8718291241737/hue-persoenliche-smarte-led-beleuchtung (visited on 10/11/2017).

[12] Amazon. *Amazon AWS IoT*. URL: https://aws.amazon.com/de/iot-platform/ (visited on 05/27/2017).

[13] Andrew Banks and Rahul Gupta. "MQTT Version 3.1. 1". In: *OASIS standard* 29 (2014).

[14] I. Fette and A. Melnikov. *The WebSocket Protocol*. RFC 6455. http://www.rfc-editor.org/rfc/rfc6455.txt. RFC Editor, 2011. URL: http://www.rfc-editor.org/rfc/rfc6455.txt.

[15] Roy T. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. http://www.rfc-editor.org/rfc/rfc2616.txt. RFC Editor, 1999. URL: http://www.rfc-editor.org/rfc/rfc2616.txt.

[16] Amazon. *Amazon AWS IoT Button*. URL: https://aws.amazon.com/de/iotbutton/ (visited on 07/12/2017).

[17] Amazon. *Amazon AWS IoT Overview*. URL: https://aws.amazon.com/de/iot-platform/how-it-works/ (visited on 07/12/2017).

[18] Olawale Oladehin and Brett Francis. *Core Tenets of IoT*. Tech. rep. Amazon Web Services, 2017.

[19] . *AWS IoT Developer Guide*. Tech. rep. Amazon Web Services, 2017.

[20] Microsoft. *Microsoft Azure IoT Solutions*. URL: https://www.microsoft.com/de-de/internet-of-things/solutions (visited on 07/12/2017).

[21] OPC. *OPC-UA*. URL: https://opcfoundation.org/ (visited on 10/11/2017).

[22] Microsoft. *Microsoft Azure IoT Hub Documentation*. URL: https://docs.microsoft.com/de-de/azure/iot-hub/ (visited on 07/12/2017).

[23] Microsoft. *Microsoft Azure IoT Hub*. URL: https://azure.microsoft.com/de-de/services/iot-hub/ (visited on 05/27/2017).

[24] Microsoft. *Microsoft Azure IoT Event Hubs*. URL: https://azure.microsoft.com/de-de/services/event-hubs/ (visited on 05/27/2017).

[25] . *Azure IoT Reference Architecture*. Tech. rep. Microsoft, 2016.

[26] SAP. *SAP ERP*. URL: https://www.sap.com/germany/products/enterprise-management-erp.html (visited on 10/11/2017).

[27] SAP. *SAP HANA IoT Architecture*. URL: https://help.hana.ondemand.com/iot/frameset.htm?4ab3521d055f41e9bce8837d4abbc09d.html (visited on 07/12/2017).

[28] Kaa. *Kaa Cluster Architecture*. URL: https://kaaproject.github.io/kaa/docs/v0.10.0/Architecture-overview/ (visited on 07/12/2017).

[29] The Apache Software Foundation. *Apache ZooKeeper*. URL: https://zookeeper.apache.org/ (visited on 10/15/2017).

[30] KaaIoT Technologies. *IoT Cloud Products*. URL: https://www.kaaiot.io/products/cloud/ (visited on 07/12/2017).

[31] OpenIoT Consortium. *Open IoT Objectives*. URL: http://www.openiot.eu/?page_id=18 (visited on 07/12/2017).

[32] Michael Compton et al. "The SSN ontology of the W3C semantic sensor network incubator group". In: *Web semantics: science, services and agents on the World Wide Web* 17 (2012), pp. 25–32.

[33] OpenIoT. *Open IoT Wiki*. URL: https://github.com/OpenIotOrg/openiot/wiki (visited on 07/12/2017).

[34] GSN. *GSN Wiki*. URL: https://github.com/LSIR/gsn/wiki (visited on 07/12/2017).

[35] OpenIoT. *Linked Sensor Middleware (LSM)*. URL: https://github.com/OpenIotOrg/openiot/wiki/Data-platform-%28lsm%29 (visited on 07/12/2017).

[36] BIG IoT. *BIG IoT*. URL: http://big-iot.eu/ (visited on 07/13/2017).

[37] BIG IoT. *BIG IoT Projects*. URL: http://big-iot.eu/project/ (visited on 07/12/2017).

[38] Werner Schladofsky et al. *Business Models for Interoperable IoT Ecosystems*. URL: http://www.arne-broering.de/IoT-WS-Paper_BIG_IoT_Business_Models_v0.7.pdf (visited on 07/12/2017).

[39] Jon Postel. *Internet Protocol*. STD 0791. http://www.rfc-editor.org/rfc/rfc791.txt. RFC Editor, 1981. URL: http://www.rfc-editor.org/rfc/rfc791.txt.

[40] S. Deering and R. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. STD 8200. RFC Editor, 2017.

[41] Andrew S. Tanenbaum and Maarten van Steen. *Verteilte Systeme: Prinzipien und Paradigmen (Pearson Studium - IT) (German Edition)*. Pearson Studium, 2007. ISBN: 978-3-8273-7293-2.

[42] Alexandr Krylovskiy, Marco Jahn, and Edoardo Patti. *Designing a Smart City Internet of Things Platform with Microservice Architecture*. Conference Paper. Fraunhofer FIT, Sankt Augustin, Germany, Dept. of Control, and Computer Engineering, Politecnico di Torino, Italy, 2015.

[43] IFTTT. *IFTTT overview*. URL: https://ifttt.com/ (visited on 10/09/2017).

[44] Günther Bengel. *Grundkurs Verteilte Systeme: Grundlagen und Praxis des Client-Server und Distributed Computing*. Springer Vieweg, 2015. ISBN: 978-3-8348-1670-2.

[45] P. Srisuresh and K. Egevang. *Traditional IP Network Address Translator (Traditional NAT)*. RFC 3022. RFC Editor, 2001.

[46] R. Perlman. "An overview of PKI trust models". In: *IEEE Network* 13.6 (1999), pp. 38–43. ISSN: 0890-8044. DOI: 10.1109/65.806987.

[47] B. C. Neuman and T. Ts'o. "Kerberos: an authentication service for computer networks". In: *IEEE Communications Magazine* 32.9 (1994), pp. 33–38. ISSN: 0163-6804. DOI: 10.1109/35.312841.

[48] Leonard Richardson and Sam Ruby. *RESTful web services*. " O'Reilly Media, Inc.", 2008.

[49] Michael Mealling. "A URN Namespace of Object Identifiers". In: (2001).

[50] Jeffrey D. Case et al. *Simple Network Management Protocol (SNMP)*. STD 1157. http://www.rfc-editor.org/rfc/rfc1157.txt. RFC Editor, 1990. URL: http://www.rfc-editor.org/rfc/rfc1157.txt.

[51] T Bray. *The JavaScript Object Notation (JSON) Data Interchange Format Internet Engineering Task Force (IETF), 7159*. 2014.

[52] Tim Bray et al. "Extensible markup language (XML)." In: *World Wide Web Journal* 2.4 (1997), pp. 27–66.

[53] Schema.org. *Welcome to Schema.org*. URL: http://schema.org/ (visited on 07/25/2017).

[54] Open Knowledge Foundation. *Linked Open Vocabularies*. URL: http://lov.okfn.org/dataset/lov/ (visited on 07/25/2017).

[55] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley Professional Computing Series)*. Addison-Wesley Professional, 1994. ISBN: 0-201-63361-2.

[56] Oracle. *JavaFX: Working with JavaFX Graphics*. URL: https://docs.oracle.com/javase/8/javafx/graphics-tutorial/shape3d.htm#CJAHFAHA (visited on 07/26/2017).

[57] H. Butler et al. *The GeoJSON Format*. RFC 7946. IETF, 2016. URL: https://tools.ietf.org/html/rfc7946 (visited on 07/26/2017).

[58] IANA. *IANA IPv4 Address Space Registry*. URL: https://www.iana.org/assignments/ipv4-address-space/ipv4-address-space.xhtml (visited on 07/26/2017).

[59] IANA. *Structure of Management Information (SMI) Numbers (MIB Module Registrations)*. URL: https://www.iana.org/assignments/smi-numbers/smi-numbers.xhtml (visited on 07/26/2017).

[60] Konstantinos Vandikas and Vlasios Tsiatsis. *Performance Evaluation of an IoT Platform*. Conference Paper. Management and Operations of Complex Systems Ericsson Research Stockholm Sweden, 2014.

[61] Inc. MongoDB. *NoSQL Databases Explained*. URL: https://www.mongodb.com/nosql-explained (visited on 10/14/2017).

[62] Z. Shelby, K. Hartke, and C. Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252. http://www.rfc-editor.org/rfc/rfc7252.txt. RFC Editor, 2014. URL: http://www.rfc-editor.org/rfc/rfc7252.txt.

[63] E. Weingartner, H. vom Lehn, and K. Wehrle. "A Performance Comparison of Recent Network Simulators". In: *2009 IEEE International Conference on Communications*. 2009, pp. 1–5. DOI: 10.1109/ICC.2009.5198657.

[64] Bjarne Stroustrup. *The C++ programming language*. Pearson Education India, 1995.

[65] Benoit Dupire and Eduardo B Fernandez. "The command dispatcher pattern". In: *8th Conference on Pattern Languages of Programs (PLOP '01)*. Citeseer. 2001.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 09, November 2017 Hauke Buhr