Technische Universität Berlin (TU Berlin)
Fachbereich IV - Elektrotechnik und Informatik

Master Thesis:
Design and Development of a Fog Service Orchestration Engine for Smart Factories

Master Thesis
from

Markus Paeschke

Supervisor: Prof. Dr.-Ing. Thomas Magedanz
Dr.-Ing. Alexander Willner
Mathias Brito

**"Es ist nicht wenig Zeit, die wir haben,
sondern es ist viel Zeit, die wir nicht nutzen."**

- Lucius Annaeus Seneca -

Markus Paeschke
Trachtenbrodtstr. 32
10409 Berlin

I hereby declare that the following thesis "Design and Development of a Fog Service Orchestration Engine for Smart Factories" has been written only by the undersigned and without any assistance from third parties.

Furthermore, I confirm that no sources have been used in the preparation of this thesis other than those indicated in the thesis itself.

Berlin, July 9, 2017

_____

Markus Paeschke

Acknowledgments

thank your supervisors

thank your colleagues

thank your family and friends

Berlin, July 9, 2017

Abstract

Research Area - write about the research area Lorem ipsum dolor sit amet, consectetur adipisic- ing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Application Area - write about the application area Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Research Issue - write about the research issue Lorem ipsum dolor sit amet, consectetur adipisic- ing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Own Approach - write about the own approach Lorem ipsum dolor sit amet, consectetur adipisic- ing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Scientific Contributions - write about the scientific contributions Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Validation & Outlook - write about the validation and outlook Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Zusammenfassung

Forschungsbereich Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

wrEingrenzungite Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Problemstellung Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Eigener Ansatz Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Wissenschaftlicher Beitrag Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Validierung & Ausblick Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Motivation

The Internet of Things (IoT) is one of the biggest topic in the recent years. Companies with a focus in that area have an enormous market growth with plenty of new opportunities, use cases, technologies, services and devices. Bain & Company predicts an annual revenue of $450 billion for companies who selling hardware, software and comprehensive solutions in the IoT context by 2020.[0] In order to limit the vast area of IoT, more and more standards are defined and subtopics established. The European Research Cluster on the Internet of Things (IERC) divided them into eight categories: Smart Cities, Smart Health-care, Smart Transport and Smart Industry also known as Industry 4.0 to mention only a few. All of them are well connected, for example a Smart Factory, which is a part of the Smart Industry, can get a delivery from a self driving truck (Smart Transport) which navigates through a Smart City to get to the factory. Such information networks are one of the main goals of IoT. In the Industry 4.0 for example multiple smart factories should be interconnect into a distributed and autonomous value chain. Also the automation in a single factory will be increased which helps to have a more flexible and efficient production process. Currently a factory has a high degree of automation, but due to a lack of intelligence and communication between the machines and the underlying system, they can not react to changing requirements or unexpected situations. One solution to achieve that are Cyber-Physical Systems (CPSs). These are virtual systems which are connected with embedded systems to monitor and control physical processes.[cf. 0, p. 363] A normal Cyber Systems (CSs) is passive, means it could not interact with the physical world, with the appearance of CPSs things can communicate so the system has significantly more intelligence in sensors and actuators.[cf. 0, p. 1363 f.]

Another solution is to change the fundamental architecture of such a system from a monolithic to more distributed multicloud architecture. With Fog Computing the cloud moves away from centralized data centers to the edge of the underlying network.[cf. 0, p. 380] Such a network can have thousands of nodes with multiple sensors, machines or smart components connected to them. An "intermediate layer between the IoT environment and the Cloud"[0, p.236] enables a lot of new possibilities like pre-computation and storage of gathered data, which reduces traffic and resource overhead in the cloud, it keeps sensitive data on-premise[cf. 0, p.236] and enables real-time applications to take decisions based on analytics running near the device and a lower latency. On the other hand there are also a lot of challenges in these highly

heterogeneous and hybrid environment. As an example in some scenarios multiple low power devices have to interact with each other, lossy signals and short range radio technologies are widely used and nodes can appear and disappear frequently.[cf. 0, p. 325] Especially the last case is elaborated because the underlying system has to handle that. Furthermore the required applications running on these nodes can be change commonly and have to be deployed and removed in a dynamical way.

Virtualization with Virtual Machines (VMs) is a common approach in Cloud systems to provide elasticity of large-scale shared resources.[cf. 0, p. 117] A more lightweight, less resource and time consuming solution is container virtualization. "Furthermore, they are flexible tools for packaging, delivering and orchestration software infrastructure services as well as application"[0, p. 117]. Orchestration tools like Kubernetes[1] and Docker Swarm[2] that can deploy, scale and manage containers to clusters of hosts have become established in the last years. If this technology can be moved over to the IoT area, many challenges can be solved. Dynamically deployed applications at the edge of a network can store and preprocess gathered data even if a node have no connection to the cloud. Traffic can be reduced by only transmitting aggregated data back to the cloud. Lossy signals can be compensated due to an autonomous behavior of the different components. More often small low power devices with a limited computational power are used as IoT nodes which also profit rather from lightweight container solutions than from resource consuming VMs. This thesis shows the capabilities of container orchestration for the IoT and smart factories by creating a prototype which can be executed on fog nodes. Therefor an engine will be created which can orchestrate containers based on functional and non-functional constraints on a single fog node or between a cluster of fog nodes.

1.2 Objective

This thesis describes an approach to design and implement a fog service orchestration engine for smart factories. The aim of this work is to create a prototype for a fog node, which can deploy containers on the same node or on other network nodes. A fog Node is typically a low power device at the edge of a network, especially in the area of Industry 4.0 and smart factories. Furthermore the prototype should consider specific functional and non-functional constraints while deploying the containers. A condition can be a hardware requirement, a required software or a dependencies to another node. The technical prerequisite and detailed requirements for the prototype as well as the usability of a GUI, which could be for example Open Baton, an European Telecommunications Standards Institute (ETSI) Network Function Virtualistion (NFV) compliant Management And Orchestration (MANO) framework, have to be worked out. The cooperation with the Fraunhofer-Institut für Offene Kommunikationssysteme (FOKUS) plays a prominent role for this thesis, because they have a lot of knowledge and experience especially in this area of IoT and NFV. As the development method of choice a Kanban like process will be used. Kanban is very flexible but also straight forward. There is less meeting overhead then in Scrum, but it also helps to have an eye on the planed and spend resources.

[1] `https://kubernetes.io`
[2] `https://docs.docker.com/engine/swarm`

1.3 Scope

As mentioned before, the engine to be developed will be a prototype for a fog node, this means the creation of a MANO engine on the server side is out of scope for this thesis. Open Baton serves as the template for that project and especially the modular architecture and the resulting extensibility will be targeted. Beside that, the whole prototype will be created completely from scratch. The container virtualization will be realized with Docker[3], an open source container platform. Docker has on API where third party apps can communicate with and can control the engine himself. The functional and non-functional constraints could base on YAML Ainť Markup Language (YAML) schemes which are part of the Topology and Orchestration Specification for Cloud Applications (TOSCA) standard for example. These schemes should be extendable and should fit the needs of the constraint functionality.

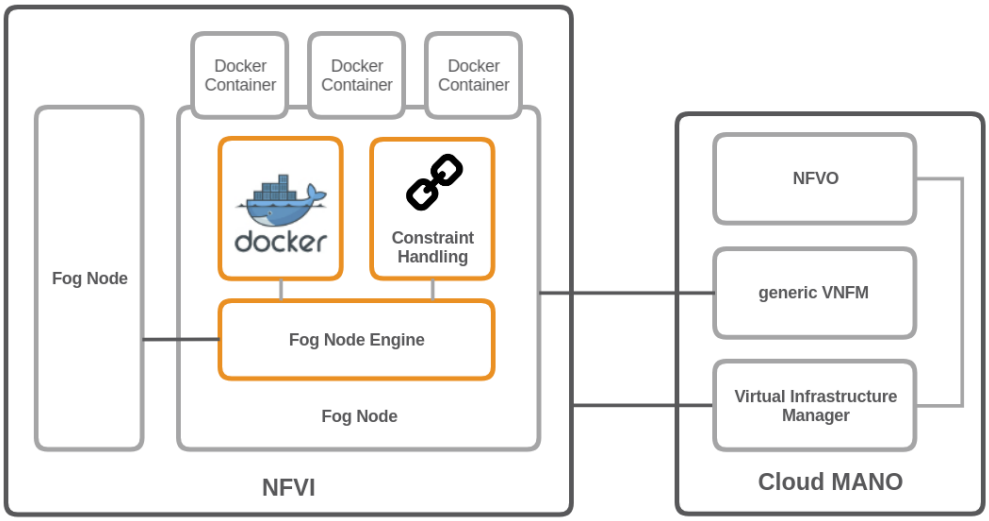

Figure 1: Conceptual architecture.

Figure 1 shows a conceptual architecture design. On the right side is the abstract cloud infrastructure. The Network Function Virtualization Infrastructure (NFVI) on the left side includes the Fog Node These could be for example Open Baton or any other MANO compliant Framework. A single fog node should have the fog node engine, which is the prototype to be developed, as well as a docker engine up and running. The constraint handling is part of the fog node engine and the concrete implementation has to be worked out. The fog node engine can orchestrate the local Docker containers, or if it necessary, it can orchestrate containers to one or multiple other fog nodes. This allows the system to be more flexible and it can achieve an autonomous orchestration level. The autonomous orchestration of containers between fog nodes without an existing connection to the cloud server is desirable, but not part of this thesis. Due to the fact that the prototype will be used in an IoT context, the system will be tested on low power devices, for example on a raspberry pi[4] cluster. It is out of scope to create a system which guaranteed to be executed on arbitrary devices, but theoretically this could be achieved with a system like Open Baton and Docker.

[3] `https://www.docker.com`
[4] `https://www.raspberrypi.org`

1.4 Outline

In chapter 2 an introduction in relevant topics like the IoT with the Industry 4.0, CPSs and Fog Computing as subtopics, virtualization especially container virtualization and orchestration and NFV is given. Followed by a comparison of several established tools available on the market and finally some messaging concepts like Message Queue Telemetry Transport (MQTT) and ZeroMQ. The definition of the requirements as well as features and capabilities of the prototype will be shown in chapter 3. Based on that, the architecture of the system is illustrated in chapter 4. The proof-of-concept implementation will be worked out in chapter 5 and the functionality of the plugin will be demonstrated. Chapter 6 summarizes the results of the work and evaluates the viability in terms of software quality, usability and feature-completeness. Finally the gained learning as well as an outlook for further improvements of the plugin will be argued in chapter 7.

Chapter 2

State of the art

This chapter will give an overview into the background and concepts of this thesis. In the first section the IoT an related subtopics like smart factories and Smart Cities are considered. CPSs, that are important for the development of smart factories are also covered in this section. Virtualization in general is the main topic of the second section. First the area of VMs will be highlighted, followed by Container Virtualization. Both are related to each other and sharing some basic ideas. Container Orchestration as an own subsection show some possibilities of Container Virtualization. The last subsection NFV concludes with an introduction into the virtualization of network node functions to create communication services. Afterwards a brief overview of some major existing tools will be given. The chapter will be finished with an excursion to two important messaging systems.

2.1 Internet of Things

The IoT has been a subject of great media- and economically growth in the recent years. In the year 2008 the number of devices which are connected to the Internet was higher than the human population.[cf. 0, p. 3] Cisco Internet Business Solutions Group predicted that the number will grow up to 50 billion in 2020, this equates to around 6 devices per person.[cf. 0, p. 4] Most of todayś interactions are Human-to-Human (H2H) or Human-to-Machine (H2M) communication. The IoT on the other hand aim for the Machine-to-Machine (M2M) communication. This allows every physical devices to be interconnected and to communicate with each other. These devices are also called "Smart Devices". Creating a network where all physical objects and people are connected via software is one primary goal of the IoT.[cf. 0, p.206][cf. 0, p.2] When objects are able to capture and monitor their environment, a network can perceive external stimuli and respond to them.[cf. 0, p. 40] Therefore a new dimension of information and communication technology will be created, where users have access to everything at any time, everywhere. In addition to smart devices, subcategories are also emerging from the IoT which, in addition to the physical devices, also describe technologies such as protocols and infrastructures. The "Smart Home" has been a prominent topic in media and business for many years. Smart City or Industry 4.0 are also becoming established and are increasingly popular. But the Internet started with the appearance of bar codes and Radio Frequency Identification (RFID) chips.[cf. 0, p. 13] The second step, which is more or less the current situation, sensors, physical devices,

technical devices, data and software are connected to each other.[cf. 0, p. 13] This was achieved, in particular, by cloud computing, which provides the highly efficient memory and computing power that is indispensable for such networks.[cf. 0, p. 206] The next step could be a "Cognitive Internet of Things", which enables easier object and data reuse across application areas, for example through interoperable solutions, high-speed Internet connections and a semantic information distribution.[cf. 0, p. V] Just as the omnipresent information processing in everyday life, also known as "Ubiquitous Computing", which was first mentioned in the "The Computer for the 21st Century"[0] by Marks Weiser, it will take some time until it is ubiquitous.

2.1.1 Industry 4.0 and smart factories

The industry as an changing environment is currently in the state of the so called "fourth industrial revolution". The first industrial revolution was driven by steam powered machines. Mass production and division of labor was the primary improvement of the second industrial revolution, whereas the third revolution was characterized by using electronics and the integration of Information Technology (IT) into manufacturing processes.[cf. 0, p. 1] In the recent years the size, cost and power consumption of chipsets are reduced which made it possible to embed sensors into devices and machines much easier and cheaper.[cf. 0, p. 1] The Industry 4.0 is the fourth step in this evolution and was first mentioned with the German term "Industrie 4.0" at the Hannover Fair in 2011.[cf. 0, p. 1] "Industrie 4.0 is a collective term for technologies and concepts of value chain organization."[cf. 0, p. 11]

Significantly higher productivity, efficiency, and self-managing production processes where everything from machines up to goods can communicate and cooperate with each other directly are the visions of the Industry 4.0.[0, cf.] It also aims for an intelligent connection between different companies and units. Autonomous production and logistics processes creating a real-time lean manufacturing ecosystem that is more efficient and flexible.[0, cf.] "This will facilitate smart value-creation chains that include all of the life-cycle phases of the product from the initial product idea, development, production, use, and maintenance to recycling."[0] At the end, the system can use customer wishes in every step in the process to be flexible and responsive.[0, cf.]

	Cyber-Physical Systems	Internet of Things	Internet of Services	Smart Factory
Interoperability	X	X	X	X
Virtualization	X	-	-	X
Decentralization	X	-	-	X
Real-Time Capability	-	-	-	X
Service Orientation	-	-	X	-
Modularity	-	-	X	-

Table 1: Design principles of each Industry 4.0 component.[cf. 0, p. 11]

Table 1 shows the six design principles which can be from the Industry 4.0 components. They can help companies to identify and implement Industry 4.0 scenarios.[cf. 0, p. 11]

1. *Interoperability* CPS of various manufacturers are connected with each other. Standards will be the key success factor in this area.[cf. 0, p. 11]

2. *Virtualization* CPS are able to monitor physical processes via sensors. The resulting data is linked to virtual plant and simulation models. These models are virtual copies of physical world entities.[cf. 0, p. 11]
3. *Decentralization* CPS are able to make decisions on their own, for example when RFID chips send the necessary working steps to the machine. Only in cases of failure the systems delegate task to a higher level.[cf. 0, p. 11]
4. *Real-Time Capability* Data has to be collected and analyzed in real time and the status of the plant is permanently tracked and analyzed. This enables the CPS to react to a failure of a machine and can reroute the products to another machine.[cf. 0, p. 11]
5. *Service Orientation* CPS are available over the Internet of Services (IoS) and can be offered both internally and across company borders to different participants. The manufacturing process can be composed based on specific customer requirements.[cf. 0, p. 11]
6. *Modularity* The system is able to be adjusted in case of seasonal fluctuations or changed product characteristics, by replacing or expanding individual modules.[cf. 0, p. 11]

Another important aspect of Industry 4.0 is the implementation of process automation with the focused on three distinct aspects. Starting with the vertical integration, which contains the connection and communication of subsystems within the factory enables flexible and adaptable manufacturing systems.[cf. 0, p. 7 ff.] The horizontal integration, as the second aspect, enables technical processes to be integrated in cross-company business processes and to be synchronized in real time through multiple participants to optimize value chain outputs.[cf. 0, p. 7 ff.] Finally end-to-end engineering, planning, and process control for each step in the production process.[0, cf.]

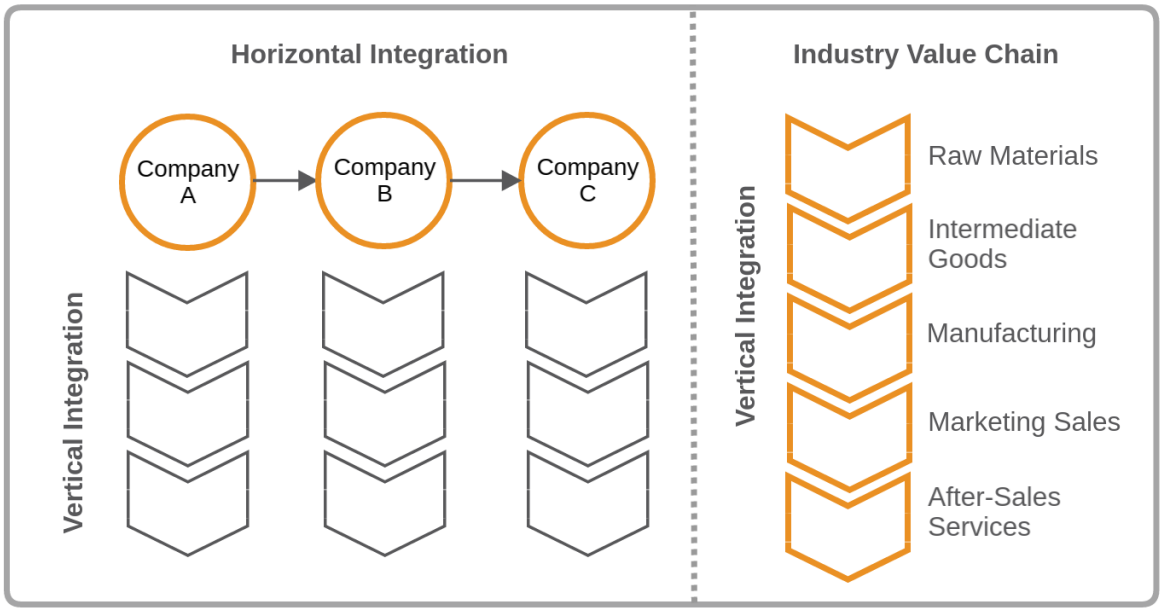

Figure 2: Horizontal vs. Vertical Integration. Adapted from: [0]

Figure 2 illustrates this concept. The left side shows the whole production process over

company boundaries on the horizontal scale, as well as the industry value chain on the vertical scale which is specific for each company. On the right side there is an exemplary industry value chain which starts with the raw materials and ends with the sale of the product to illustrates an more specific example of the vertical integration. From a technical site this means each machine in a factory has exactly to know what they have to do. The underlying system has to be modular and move away from a monolithic centralized system, to a decentralized system which is located locally near the machines himself. The communication path between them have to grow shorter. The machines have to be self organized and should communicate between each other even if the core system is not reachable because of lossy signals or other connection issues. If this can be achieved there will be an highly flexible, individualized and resource friendly mass production, which can be cheaper, faster and can have a much higher fault tolerance.

2.1.2 Cyber Physical Systems

As we already now, in smart factories every physical device is connected to each other. Everything can be captured and monitored in each step of a production process. With CPSs every physical entity has a digital representation in the virtual system.[cf. 0, p. 1363] Before a CS was passive, which means there was no communication between the physical and the virtual world.[cf. 0, p. 1364] While new technologies in the physical world, like new materials, hardware and energy, are developed, the technologies in the virtual worlds are also being improved, for example through the use of new protocols, networking, storage and computing technologies.[cf. 0, p. 1364] This adds more intelligence in such systems, as well as a much more flexible and modular structure. A CPS can organize production automatically and autonomously, which eliminate the need of having a central process control.[0, cf.] Thereby the system can handle lossy signals and short range radio technologies, which are widely used in such a context.[0, cf.] In summary CPSs can help to enable the vision of smart factories in both the horizontal as well as the vertical integration.

2.1.3 Fog Computing

In the beginning of Cloud Computing most of the systems based on a monolithic architecture. Over time the system was broken down to a more distributes multicloud architecture, similar to microservices. With the appearance of Fog Computing the Cloud also moves from centralized data centers to the edge of the underlying network. Main goal is to improve the efficiency, reduce the traffic and the amount of data which is transferred to the cloud and also process, analyze and store data locally, as well as keeping sensitive data inside the network for security reasons.[cf. 0, p. 236][cf. 0, p. 325][cf. 0, p. 4] In contrast to the goals the definition and understanding of Fog Computing differs. One perspective is to that the processing of the data take place on smart devices, e.g. sensors, embedded systems, etc., at the end of the network or in smart router or other gateway devices.[cf. 0, p. 4] Another interpretation is that fog computing appears as an intermediate layer between smart devices and the cloud.[cf. 0, p. 236] Processing the data near devices enables lower latency and real-time applications can take decisions based on analytics running there. That is important because a continuous connection to the cloud can not always be ensured. However fog computing should not be seen as a competitor of cloud computing, it is a perfect ally for use cases where cloud computing alone is not feasible.[cf. 0, p. 325]

2.2 Virtualization

According to the National Institute of Standards and Technology (NIST) the definition of virtualization is: "Virtualization is the simulation of the software and/or hardware upon which other software runs. This simulated environment is called a virtual machine (VM)."[0, p. ES-1]. This means a VM, also referred as guest system, can be executed in a real system, which is referred as host system. A VM has its own Operating System (OS) which is completely isolated from the other VMs and the host system.[cf. 0, p. 2] Basically there are two types of virtualization: Process virtualization where the virtualization software also known as Virtual Machine Monitor (VMM) is executed by the host OS and only an application will be executed inside the guest OS and on the other side there is the the system virtualization where the whole OS as well as the application are running inside the virtualization software. Figure 3 illustrate both concepts. Examples for process virtualization could be the Java Virtual Machine (JVM)[1], the .Net framework[2] or Docker[3], where VMWare[4], Oracle Virtual Box[5], XEN[6] or Microsoft Hyper-V[7] are only some examples for system virtualization. The benefits of all virtualization techniques are the rapid provisioning of resources which could be Random Access Memory (RAM), disk storage, computation power or network bandwidth. Beside that, no human interaction is necessary during the provisioning process. Elasticity which scales a system in a cost-efficient manner in both directions, up and down. Customer as well as the provider profit from such a system. Security based on the isolation of the VMs is another huge benefit. Different processes can not interfere with each other and the data of a single user can not be accessed by other users of the same hardware. A challenge despite all the mentioned benefits is the performance. Running VMs increases the overhead and reduces the overall performance of a system. Therefore the specific use case have to consider these behavior.

[1] https://www.java.com
[2] https://www.microsoft.com/net
[3] https://www.docker.com
[4] http://www.vmware.com
[5] https://www.virtualbox.org
[6] https://www.xenproject.org
[7] https://www.microsoft.com/de-de/cloud-platform/server-virtualization

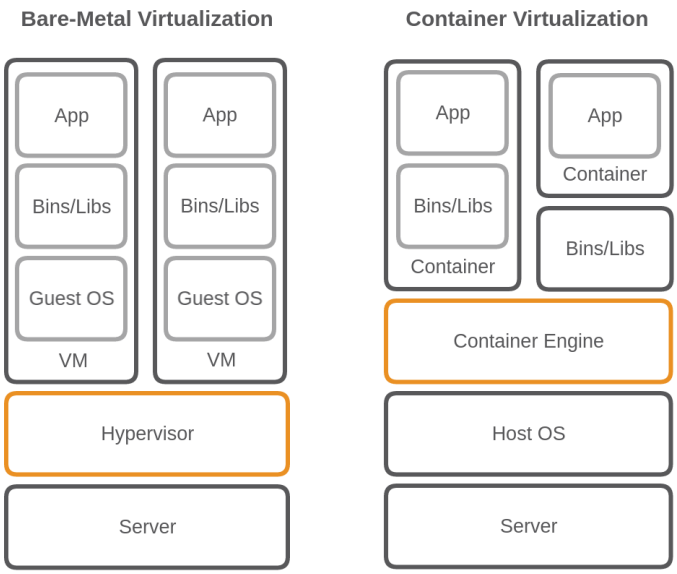

Figure 3: Structure bare-metal virtualization vs. container virtualization. Adapted from: [0, p. 2]

2.2.1 Virtual Machines

VMs are the core virtualization mechanism in cloud computing. There are also two different designs for hardware virtualization. The first and more popular type for cloud computing is the *bare-metal virtualization*. It needs only a basic OS to schedule VMs. The hypervisor runs directly on the hardware of the machine without any host OS in between. This is more efficient, but requires special device drivers to be executed. The other type is the *hosted virtualization*. Unlike the first type the VMM run as a host OS process and the VMs as a process supported by the VMM. No special drivers are needed for these type of virtualization, but by comparison the overhead is much bigger. For both types, the performance limitation remains. Each VM need a full guest OS image in addition to binaries and libraries which are necessary for the application to be executed.[cf. 0, p. 381] If only a single application, which only needs a few binaries and libraries, is needed to be virtualized, VMs are too bloated.

2.2.2 Container Virtualization

Container virtualization which is also known as Operating System-level virtualization, is the second virtualization mechanism. It based on fast and lightweight process virtualization to encapsulate an entire application with its dependencies into a ready-to-deploy virtual container.[cf. 0, p. 72] Such a container can be executed on the host OS which allows an application to run as a sand-boxed user-space instance.[cf. 0, p. 1] All containers share a single OS kernel, so the isolation supposed to be weaker compared to hypervisor based virtualization.[cf. 0, p. 2] Compared to VMs, the number of containers on the same physical host can be much higher, because the overhead of a full OS virtualization is eliminated.[cf. 0, p. 2]

2.2.3 Container Orchestration

Containers by itself helps to develop and deploy applications, but containers release their full potential only when they are used together with an orchestration engine. Before orchestration engines, the deployment of an application or service was realized via CI and deployment tools like Vagrant or Ansible. Deployment scripts or plans was created and be executed every time an application changed or should be scaled up on a new machine. This was less flexible and error-prone. Orchestration engines cover these needs by automatically choosing new machines, deploying containers, handle the lifecycle of them and monitor the system. These flexibility enables a new level of abstraction and automatization of deployment. There are a bunch of orchestration engines out there. For Docker, Kubernetes and Docker Swarm are the most popular at the moment.

2.2.4 Network Function Virtualization

NFV is an architectural framework to provide a methodology for the design, implementation, and deployment of Network Functions (NFs) through software virtualization.[cf. 0, p. 8][0, cf.] "These NFs are referred as Virtual Network Functions (VNFs)."[0, p. 8] It takes into consideration Software Defined Networking (SDN) and preparing for the use of non-proprietary software to hardware integration instead of multiple vendor specific devices for each function, e.g. routers, firewalls, storages, switches, etc.[0, cf.] Now high-performance firewalls and load balancing software for example can run on commodity PC hardware and traffic can be off-loaded onto inexpensive programmable switches.[0, cf.]

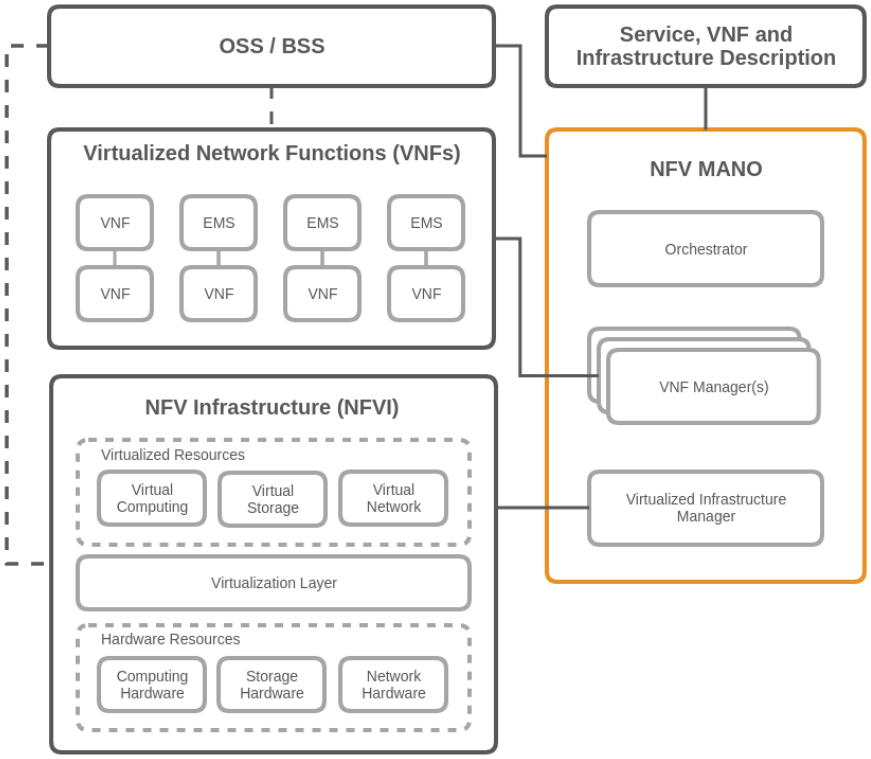

Figure 4: NFV architecture. Adapted from: [0]

Some benefits are speed, agility and cost reduction in deployment as well as execution manner.[0, cf.] Using homogeneous hardware simplifies the process of planning and reduces power, cooling and space needs.[0, cf.] Through virtualization providers can utilize resources more effectively, by allocating only the necessary resources for a specific functionality.[0, cf.] Overall NFV can reduce Operating Expense (OpEx) as well as Capital Expenditure (CapEx) and can decreasing the time necessary to deploy new services to the network.[0, cf.] To achieve NFV the ETSI has defined a framework the Network Function Virtualistion Management And Orchestration (NFV-MANO)[8] and the Organization for the Advancement of Structured Information Standards (OASIS) created TOSCA a NFV specific data model and templates to coordinate and orchestrate the NF into the cloud.

OSS / BSS refers to the Operations Support Systems (OSS) and the Business Support Systems (BSS) of a telecommunication operator.[0, cf.] The OSS is responsible for the underlying soft- and hardware system, for example for network and fault management. The BSS on the other hand is responsible for the business handling, for example customer and product management. Both can be integrated with the NFV-MANO.[0, cf.]

VNFs are the virtualized network elements, for example a virtualized router or virtualized firewall. Even if only a sub-functions or a sub-components of a hardware element is virtualized, it is called VNF.[0, cf.] Multiple sub-functions can act together as one VNF.

Element Management System (EMS) is responsible for the functional management of a single or multiple VNFs.[0, cf.] This includes fault, configuration, accounting, performance and security management.[0, cf.] Furthermore the EMS itself can be a VNF or it can handle a VNF through proprietary interfaces.[0, cf.]

NFVI is the environment where VNFs are executed. This includes physical resources as well as virtual resources and the virtualization layer. The physical resources could be a commodity switch, a server or a storage device. These physical resources can be abstracted into virtual resources through the virtualization layer which is normally a hypervisor. If the virtualization part is missing, the software runs natively on the hardware and the entity is no longer a VNF it is then a Physical Network Function (PNF).[0, cf.]

NFV-MANO consists of three main parts. The Virtual Infrastructure Manager (VIM) is "responsible for controlling and managing the NFVI compute, network and storage resources within one operator's infrastructure domain"[0]. The Virtual Network Function Manager (VNFM) manages one or multiple VNFs. This includes the life cycle management of the VNF instances, such as instantiate, edit or shut down an VNF instance.[0, cf.] In contrast to the EMS, the VNFM handles the virtual part of the VNF, for example instantiate an instance, while the EMS handles the functional part of an VNF, such as issue handling for a VNF. The orchestrator as the third component in the NFV-MANO block manage network services of VNFs. It is responsible for the global resources management, such as computing

[8] `http://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_NFV-MAN001v010101p.pdf`

and networking resources among multiple VIMs.[0, cf.] The orchestrator interacts with the VNFM to perform actions, but not with the VNFs directly.[0, cf.] TOSCA is often used with NFV-MANO frameworks like Cloudify[9] or Open Baton.[0, cf.]

TOSCA is developed by the OASIS to deliver a declarative description of a NFV application topology for network or cloud environments.[0, cf.] In figure 4 it is represented by the *Service, VNF and Infrastructure Description* block. Beside that, it can also be used to define a workflow which should be automated in a virtualized environment.[0, cf.] The TOSCA modeling language can specify nodes, whereby a node can be a network, a subnet or only a server software component, and it also handles relationships between the nodes and also services.[0, cf.] To define schemes, relationships and the configuration of such an infrastructure, it uses YAML files for ease the usage.[0, cf.] TOSCA works pretty well with NFV-MANO components to automate the deployment and management of NFs and services.

2.3 Existing tools

There are several advantages of using frameworks and sophisticated tools, for example they reduces the time and energy in developing any software, they are more secure, well tested and they provides a standardized system through which users can develop applications. They also allow to create a prototype of an application in a short amount of time. To achieve the benefits the user has to spend some time to learn the concepts, functions and how to use a framework.

2.3.1 Linux Containers

When we talk about container virtualization nowadays, Docker have to become one of the most famous tools out there. It based on Linux Containers (LXC)[10] a technology which uses kernel mechanisms like *namespaces* or *cgroups* to isolate processes on a shared OS.[cf. 0, p. 381] Namespaces for example are used to isolate groups of processes whereas cgroups are used to manage and limit resources access just like restricting the memory, disc space or Central Processing Unit (CPU) usage.[cf. 0, p. 381] "The goal of LXC is to create an environment as close as possible to a standard Linux installation but without the need for a separate kernel."[0, p. 72] There are several other container virtualization tools out there like OpenVZ[11] or Linux-VServer[12]. In contrast to them, an advantage of LXC is that it runs on an unmodified Linux kernel. This means that LXC can be executed in most of the popular Linux distributions these days.

2.3.2 Docker

As mentioned before, Docker based on LXC. This allows the Docker Engine to build, deploy and run containers in an easy and customizable way. Similar to VMs, containers are executed

[9] `http://getcloudify.org`
[10] `https://linuxcontainers.org/`
[11] `https://openvz.org/Main_Page`
[12] `http://www.linux-vserver.org`

from images. A mayor benefit of Docker is the fact, that Docker images can be combined like build blocks. Each image can build on top of another. Figure 5 illustrates the concept for the image of the pretty famous Django[13] web framework. In that case, the Django image, which is the resulting image, based on the Python 3.4 image which again based on the Debian Jessy image. All of them are read-only, but Docker adds a writable layer, also known as *container layer*, on top of the images as soon as the container will be created. File system operations such as creating new files, modifying or deleting existing files are written directly to these layer.[0, cf.] The other images dont́ get involved. These chaining mechanism of images allows Docker to ease the use of dependencies and administrative overhead.

Beside that, Docker is split up in several components, such as the Docker Engine, the Docker Registries and Docker Compose to mention only a few. The Docker Engine is a client-server application which can be distinguished by the Docker client, a Representational State Transfer (REST) API and the Docker server. Latter is a daemon process which "creates and manages Docker objects, such as images, containers, networks, and data volumes"[0]. The client is a Command Line Interface (CLI), which interact via a REST API with the daemon.[0, cf.]

Another component, the Docker Registry, is basically a library of Docker images. They can be public or private available, as well as on the same machine like the Docker daemon or an external server.[0, cf.] The most popular one it the official Docker Hub[14]. There is also an Docker Store[15], where customers can buy and sell trusted and enterprise ready containers and plugins. With the Docker client it is pretty easy to *search* for new containers and to *pull* containers from or *push* containers to a specific repository.

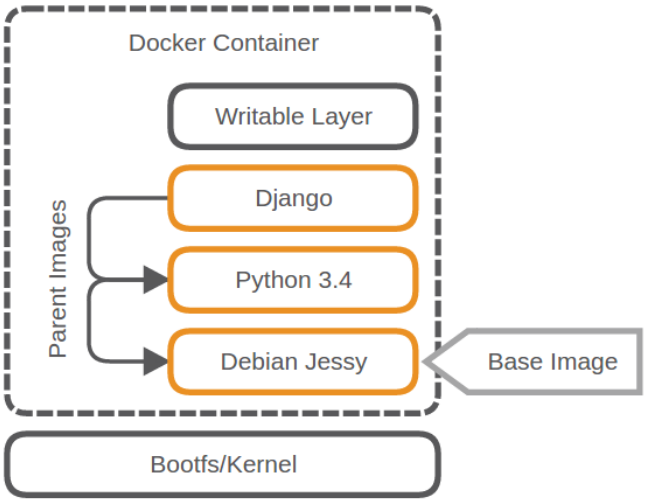

Figure 5: Docker container structure.

With Docker Compose multiple Docker Containers can be executed as a single application. Therefore YAML compose file will be used to configure and combine the services. For example the already mentioned Django image can be executed and linked together with a MongoDB[16]

[13] https://www.djangoproject.com/
[14] https://hub.docker.com
[15] https://store.docker.com
[16] https://www.mongodb.com

images. The main benefit is the ease of configure dependencies between several containers and configuration steps. These concept is similar to deployment tools like Vagrant[17], Ansible[18] or Puppet[19].

A major benefit of Docker is that the execution environment of an application is completely the same on a local machine as on the production environment.[cf. 0, p. 2] There is no need to do things differently when switching from a development environment like a local machine, to a production environment like a server.[cf. 0, p. 2]

2.3.3 Kubernetes

Kubernetes is an open source container cluster manager which was released 2014 by Google. It is "a platform for automating deployment, scaling, and operations"[0, p. 1] of containers. Therefore a cluster of containers can be created and managed. The system can for example schedule on which node a container should be executed, handle node failures, can scale the cluster by adding or removing nodes or enable rolling updates.[0, p. 5 f.]

Figure 6 illustrates the basic architecture of Kubernetes. The user can interact with the Kubernetes system via a CLI, an User Interface (UI) or a third party application, over a REST API to the Kubernetes Master or more specifically the API Server in the master. The master himself controls the one or multiple nodes, monitor the system, schedule resources or pull new images from the repository, to name only a few tasks.

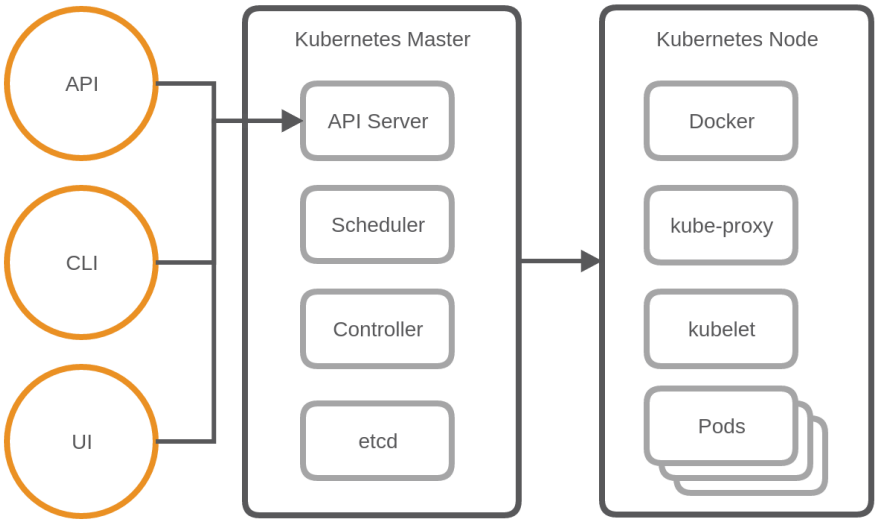

Figure 6: Kubernetes architecture. Adapted from: [0, p. 4]

Each node has a two way communication with the master via a kubelet. In addition each node has the services necessary to run container applications like Docker. Furthermore Kubernetes can combine one or multiple containers into one so called Pod.[cf. 0, p. 7] "Pods are always co-located and co-scheduled, and run in a shared context."[0] One node again can execute multiple Pods. Pods are only temporary grouped containers with a non-stable Internet

[17] https://www.vagrantup.com
[18] https://www.ansible.com
[19] https://puppet.com

Protocol (IP) address. After a Pod is destroyed it can never be resurrected. Pods can also share functionality to other Pods inside a Kubernetes cluster. A logical set of Pods and the access policy of them is called a Kubernetes service. Such a service can abstract multiple Pod replicas and manage them. A frontend which have access to the service do not care about changes in the service. Any change, be it a down scale or an up scale of the system, remains unseen for the frontend. They are exposed through internal or external endpoints to the users or the cluster.[cf. 0, p. 11] Labels can be used to organize and to select subsets of Kubernetes Objects, such as Pods or Services.[0, cf.] They are simply key-value pairs which and should be meaningful and relevant to users, but do not imply semantics to the core system.[0, cf.]

The kube-proxy is a network proxy and load balancer which is accessible from the outside of the system via a Kubernetes service.[cf. 0, p. 7] "Each node and can do simple TCP,UDP stream forwarding or round robin TCP,UDP forwarding across a set of backends."[0] The Replication Controller is one of the mayor controllers in a Kubernetes System. It ensures that a specified number of pod replicas are running and available at any time.[0, cf.] If for example one node disappear because of connection issues, the Replication Controller will start a new one. If the disappeared node is available back again it will kill a node. These functionality increases the stability, the availability and the scalability of the system in an autonomous manner. The last important component in Kubernetes are rolling updates. With rolling updates the system can update one pod at a time, rather than taking down the entire service and update the whole system.[0, cf.] This also increases the stability and availability of the system and eases the managing of container clusters.

2.3.4 Docker Swarm

The basic functionality of Docker Swarm is pretty similar to Kubernetes: It is possible to create, manage and monitor a cluster of multiple machines running Docker on it. Before Docker version 1.12.0, Docker Swarm was an independent tool, which is now integrated in the Docker Engine.[0, cf.] No additional software is necessary to have a bunch of machines work together as a so called swarm. Similar to Kubernetes, Docker Swarm needs a master node called manager and several worker nodes. The manager for example keep track of the nodes and their lifecycle and it can start new instances of an image if one or multiple nodes disappear. Furthermore Docker Swarm has a build in proxy and load balancer, which can redirect requests to the node with the necessary container running on it or redirect requests based on the workload of the machines. Compared to Kubernetes, Docker Swarm is more lightweight, but misses some features like the label functionality or the schema definition of a pod. But as mentioned before, both tools are pretty similar and aim for the same goal.

2.3.5 Open Baton

Open Baton[20] is an open source ETSI NFV compliant MANO Framework[0, cf.]. "It enables virtual Network Services deployments on top of heterogeneous NFV Infrastructures."[0] It works together with OpenStack and provides a plugin mechanism which allows to add additional VIMs.[0, cf.] In Open Baton it is implemented as the VIM as first Point of Presence (PoP) and uses the OpenStack APIs.[0] All the resources in the NFVI are controlled by the VIM, in this case OpenStack.

[20] `https://openbaton.github.io`

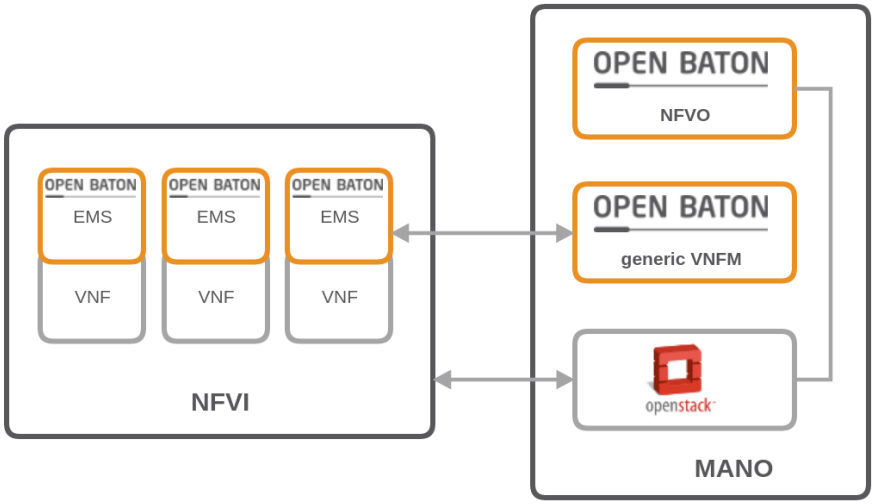

Figure 7: Open Baton abstract architecture. Adapted from: [0]

In the basic configuration Open Baton provides a generic VNFM with generic EMS related to the VNFs, but it can also be replaced with custom components. The VNFM can use a REST API or an Advanced Message Queuing Protocol (AMQP) message queue to communicate with the system. Figure 7 illustrates the abstract architecture of Open Baton together with OpenStack and a generic VNFM. The Network Function Virtualistion Orchestrator (NFVO) is completely designed and implemented as described in the ETSI MANO standard.[0] It communicates with the VIM to orchestrate resources and services and it is implemented as a separate module, so it can be replaced with a custom one if necessary.

A more detailed view of the Open Baton architecture is shown in figure 8. As mentioned before each component communicate over the message queue and can be extended or replaced if necessary. Additional components, such as the Autoscaling Engine (AE) or the Fault Management (FM) system, are provided to manage a network service at runtime.[0] The necessary information are delivered from the monitoring system available at the NFVI level, which can also be extended or replaced with any monitoring system by implementing a custom monitoring driver.[0] The VIM Driver mechanism allows to replace OpenStack with external heterogeneous PoPs, but without the need of modifying the orchestration logic.[0] Beside the generic VNFM, also the Juju[21] VNFM can be used to deploy Juju Charms or Open Baton VNF packages. Open Baton also provides a marketplace[22] for free and open source VNFs, which can directly be loaded into the system.

Furthermore Open Baton comes with a modern and easy to use GUI and user management. The typical workflow of running a NFVI is by starting them through the dashboard. The user input, in this case deploying a VNF, will be submitted as a request to the NFVO. There the orchestrator request the VIM, for example OpenStack, to instantiate the network service. The VIM allocated the resources on the datacenter and starts the VMs based on the provided service description, for example through a TOSCA description. After the machines are finally booted, the EMSs will be installed to communicate with the VNFMs. Open Baton now can send lifecycle events to all the VNFMs responsible for the VNFs which are part of the network

[21] `https://www.ubuntu.com/cloud/juju`
[22] `http://marketplace.openbaton.org`

service. Finally the VNFMs processes the VNFs via the EMS on to the given resources of the NFVI on the datacenter. The services are started and the system is up and running.

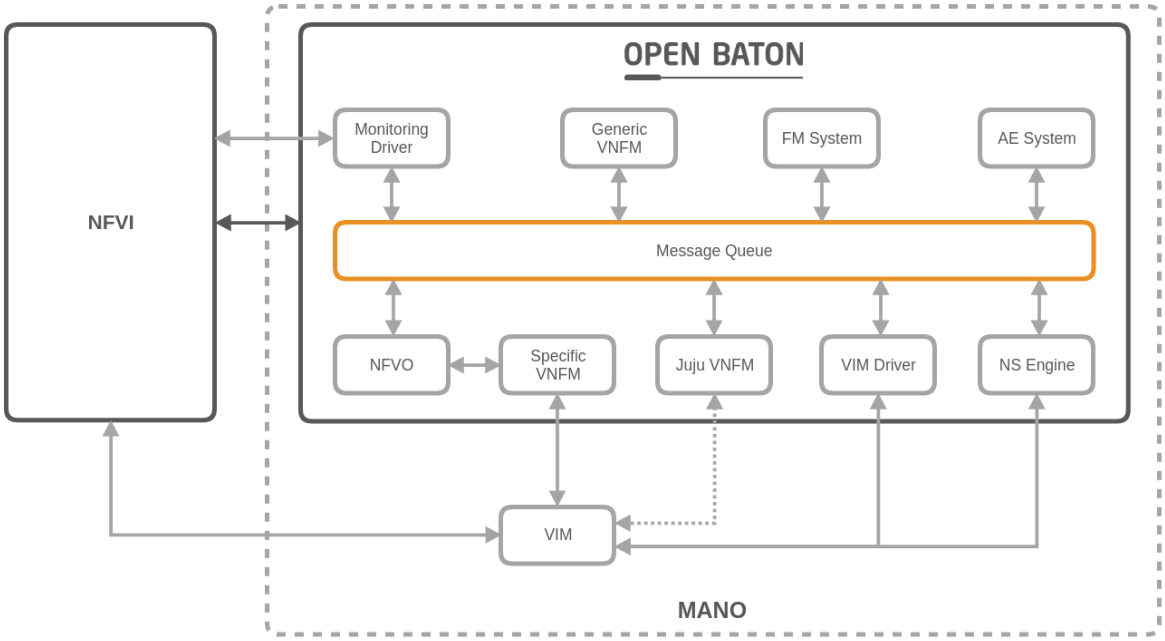

Figure 8: Open Baton detailed architecture. Adapted from: [0]

2.4 Messaging

Messaging protocols are crucial for the IoT area. On the one side bandwidth in an IoT network can be very limited due to a bad infrastructure or connection issues and on the other side a huge amount of data can be generated by all the connected devices. Hundreds or thousands of devices can probably send data at the same time. On top of that several devices like sensors and actuators can send a huge amount of data because a sensor reads data pretty fast but mostly only simple data like the temperature or the humidity or something similar. All conditions ends up in a huge network traffic. Therefore it is quite important the applications in an IoT context use a lightweight protocol which has less data overhead and can delivers data packages very fast. Mostly it is not important to receive each package because for example in several use cases the temperature that is measured via a sensor does not change drastically in a few millisecond. It would be enough to get the data every second and even in this case it would be probably fine to loose a single package. But if there are hundreds of sensors sending the data it could me more useful to reduce the bandwidth and also the latency. For some devices it could be crucial to react to events or issues as fast as possible. Hence the latency should be as low as possible. This also suggests to use a lightweight and fast protocol. In the following two protocols that fits the needs will be discussed.

2.4.1 Message Queue Telemetry Transport

The MQTT protocol formerly known as MQTT-S or MQTT-SN is a lightweight communication protocol developed by Andy Stanford-Clard and Arlen Nipper.[0, cf.] Meanwhile MQTT is an OASIS standard[23], which is often used in an IoT and M2M context.[cf. 0, p. 5] It has a publish/subscribe architecture, which makes it easy to implement and allows thousands of remote clients to be connected to a single server at the same time.[cf. 0, p. 5] The recipient of a message which is called consumer in the MQTT context is completely decoupled from the sender, mostly called producer, via a broker. In general the workflow is that a consumer subscribe to a specific topic at the broker and the producer can send messages with a specific topic to the broker. The producer does not know if there is any consumer subscribed to the topic of a message. The broker is responsible for delivering messages to consumers, by receiving them from the producer and send out copies to the consumers. Figure 9 illustrates this concept.

In contrast to Client/Server protocols such as the HTTP, MQTT is event-oriented, which means that the client does not have to constantly ask the server if there is new data, the broker informs the consumer when there is new data on a topic.[0, cf.] In direct comparison, this concept decrease the traffic, the amount of connections at the server and the delay of the message to be send to the clients.

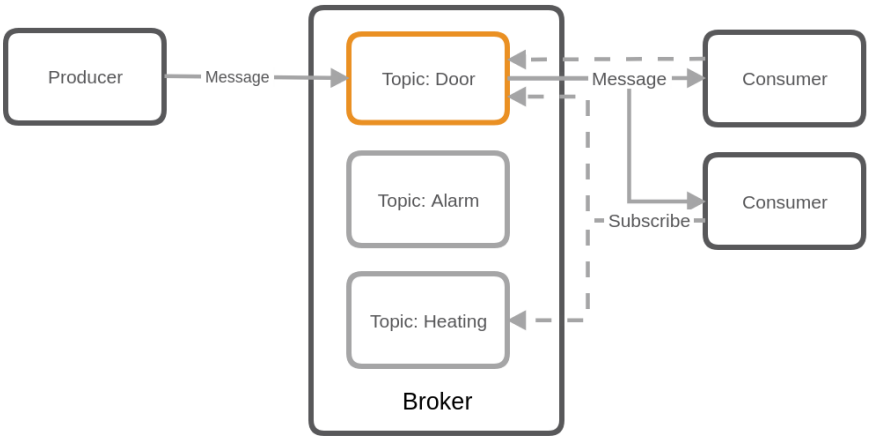

Figure 9: MQTT publish/subscribe architecture. Adapted from: [0]

Beside the publish/subscribe architecture and the lightweight protocol, MQTT has only few but useful features. **Quality of Service (QoS)** define the level of reliability with which messages are delivered.[0, cf.] There are three different levels, where each level differs in reliability and resources usage.[0, cf.] In an IoT context, for instance gathering temperature sensor data from a low power device, most of the time keeping the resources usage low is more important than the reliability of getting every single message.
At *QoS level 0 - at most once*, it is guaranteed that a message can only arrives once, but they can also be lost during transfer. A single message will be send once and the publisher does not check the success of receiving them. This pattern is also called *Fire and Forget*, it is fast and resources friendly.[0, cf.]

[23] `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=mqtt`

At *QoS level 1 - at lest once*, it is guaranteed that a message will be received at the consumer. After the publisher send the message with a specific packet identifier, the consumer will confirm the receipt of the message with a so called pubback packet. The pubback packet also have the same packet identifier included, so that the publisher will know that the message was successfully delivered. If a message get lost during transmission, they will be resend. It is also possible, that the same message appears multiple times at the consumer, depending on the delay in the network.
The most secure but also most resources consuming level is At *QoS level 2 - exactly once*. At this level it is guaranteed, that a message is exactly received once. It is not possible that a message appears multiple times or never. This level has a two-level confirmation process, where at first the consumer confirms the receive of the message and afterwards the producers confirms the receive of the confirmation. Therefor both sides can be sure to not send or receive duplicates and it is guaranteed that a message will be received.

Furthermore the broker has two features to handle connection loses: the *last will testament* and the *message persistence*. The former feature will send a last message from the broker to a consumer if the connection to the publisher will get lost. This could be for instances the information that the connection get lost or something similar. The message persistence will be used if one consumer lose the connection to the broker. In this case the message will be stored and delivered as soon as the consumer reconnects. With the *retrained messages* feature, a consumer which is connected for the first time to the broker will get the last message send for a specific topic. This can be useful if the temperature from a sensor should be displayed, but the value will only be fetched every 30 seconds. This would lead to the behaviour that the initial value on consumer side will in worst case be unknown for 30 seconds. *Persistent sessions* allows a connection be established even if the consumer disappears. In this case all the messages incurred will be stored and delivered if the consumer resume the session. The session can be identified with a unique client identifier.[0, cf.]

Due to the fact that MQTT is a simple protocol with a small footprint and the QoS handshake enables the protocol to be independent from TCP so that it can be used even on devices without a TCP/IP stack like embedded devices such as an Arduino.[0, cf.] There the protocol can be used via a bus or a serial port.[0, cf.] MQTT himself support the protection of the messages via username and password and the communication can be encrypted with SSL or TLS on the transport layer.[0, cf.] The broker can additionally use client certificates to authenticate them or restrict the access via access control lists such as IP filtering.[0, cf.] MQTT is available for most of the common programming languages and platforms.

2.4.2 ZeroMQ

ZeroMQ is an messaging and communication framework to send atomic messages between applications and processes across various transports like Transmission Control Protocol (TCP), in-process, inter-process or multicast.[0, cf.] Every message will be send over sockets via several patterns like publish-subscribe, fan-out or a simple request-reply.[0, cf.] ZeroMQ has an asynchronous I/O model which allows to create high-performance and scalable multicore applications.[0, cf.] To distinguish it from messaging frameworks like AMQP[24] ZeroMQ has no dedicated broker in between. The benefit is that there is no single point of failure, not bottleneck and no need to maintain another component, but ZeroMQ still has the advantages of

[24] `https://www.amqp.org`

such a messaging system.

ZeroMQ supports five different transport types.
In-Process is used for local (in-process or inter-thread) communication transport. This transport passes the messages directly via memory between threads.[0, cf.] These transport type is optimal for creating multithreaded applications without providing access to the outside. This is the fastest transport type available in ZeroMQ.
Inter-Process Communication (IPC) provides also a local communication transport, but passes messages via the OS dependent IPC mechanism, for example UNIX domain sockets. An application can provide a local API for another local application via IPC. Also IPC is much faster than the TCP communication.
TCP is an ubiquitous, reliable, unicast transport to provide an API over a network.[0, cf.]
Pragmatic General Multicast (PGM) is a multicast communication transport using the PGM standard protocol[25] and the datagrams are layered directly on top of IP datagrams.[0, cf.] It is helpful to send a sequence of packets to multiple consumers at the same time. Therefore PGM and also EPGM can only be used with the publish/subscribe pattern.
Encapsulated Pragmatic General Multicast (EPGM) is similar to PGM but with the difference that the PGM datagrams are encapsulated inside UDP datagrams.[0, cf.]

ZeroMQ has several basic patterns and most of them are combinable. To describe them all in detail would exceed the scope of this work, so only the most relevant ones are considered.
Request-Reply is comparable with a Hypertext Transfer Protocol (HTTP) request-response. The client send a message to the server, which does some work and send back a message afterwards. It is represented by the REQ-REP socket pairs in ZeroMQ. Figure 10 illustrates the pattern.

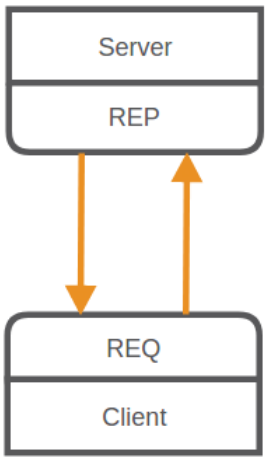

Figure 10: ZeroMQ Request-Reply architecture. Adapted from: [0]

The **Publish-Subscribe** pattern in ZeroMQ is basically comparable with the MQTT publish-subscribe pattern, but without the broker in between. This ends up in the fact that every consumer has to now the publisher and has to connect to them. A direct connection between them will established. Similar to MQTT, the publisher will send out the message to every

[25] `https://tools.ietf.org/html/rfc3208`

subscribed consumer. It is also possible to subscribe to more than one topic. In ZeroMQ this pattern is represented by PUB-SUB socket pairs. Figure 11 shows this pattern.

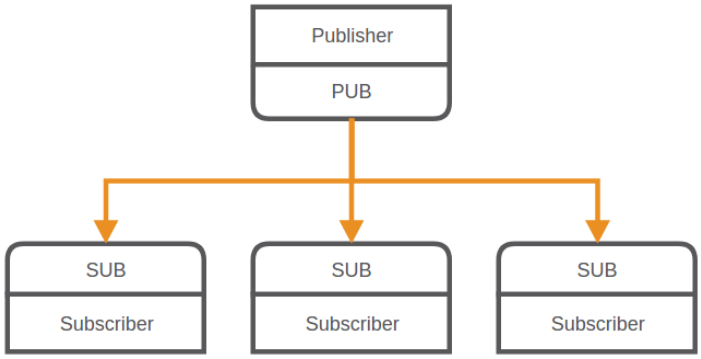

Figure 11: ZeroMQ Publish-Subscribe architecture. Adapted from: [0]

The **Pipeline** pattern is also known as the *Divide and Conquer* pattern. There we have Ventilator which produces multiple tasks that can be done in parallel, a set of worker that can process these tasks and a sink that collects the results from the workers.[0] Benefit of this pattern is, that the workers divide the tasks, this means it will increase the calculation time based on the connected workers. Furthermore the works can pull a new task if they are done. This means a worker has a minimal idle time. Queuing is provided by ZeroMQ. It is also possible to add and remove workers dynamically. This makes an application much more scalable. Finally the sink pull the data from the worker in a so called *fair-queuing*. This means the sink will pull one package from each worker one after another, then he starts from the beginning and will pull again only one package, even if one or multiple worker should have multiple packages retrievable. The whole pattern is shown in figure 12

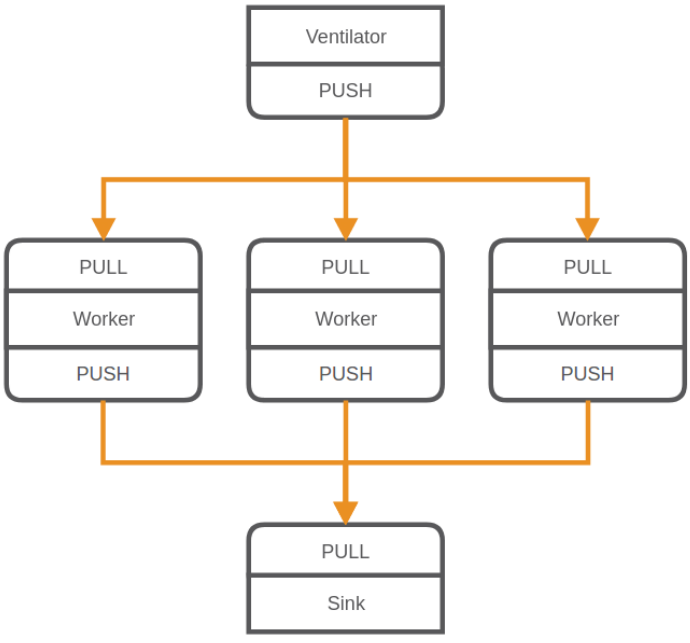

Figure 12: ZeroMQ Pipeline architecture. Adapted from: [0]

Exclusive pair connects tow sockets exclusively, for example two threads in a process.[0]
Combinations of them at least these some of the patterns in ZeroMQ. All of these patterns can be combined to create much more advanced combinations. Figure 13 illustrates such a combination.

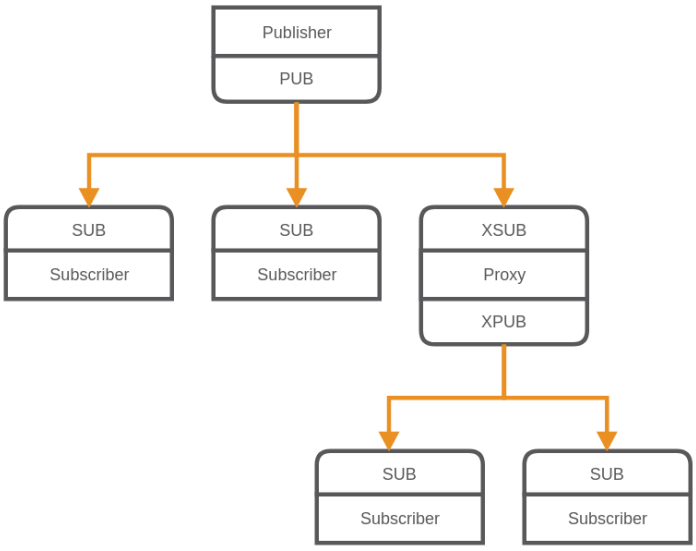

Figure 13: ZeroMQ combination of patterns. Adapted from: [0]

There is a combination between two publish-subscribe patterns and a proxy in between. This could be helpful to create a nested topology, for instances for a controller in a smart home environment. Multiple controllers can be subscribed to a server and multiple nodes can be subscribed to one controller. As mentioned before, there are much more combinations possible. A good overview of many of them can be found in the official guide[26] of ZeroMQ.

By default ZeroMQ has no encryption or authentication machanismn build in. There is a dedicated project called CurveZMQ[27] which enables these functionalities and it is also created by the ZeroMQ maintainers. Since ZeroMQ version 4.x CurveZMQ comes built-in. Finally ZeroMQ has libraries for most of the common programming languages.

[26] `http://zguide.zeromq.org/page:all`
[27] `http://curvezmq.org`

Chapter 3

Requirements Analysis

Based on the fundamentals the requirements for the prototype to be developed will be formulated in this chapter. Thereby relevant aspects for the specific implementation will be considered. The analysis, the creation of requirements and the commitment from all sides to these requirements is an important step to successfully realize a project, not only in the software development area. It does not matter if it is a private project, a company product or a project from a customer. The better a project is defined and everyone is on the same state, the better the final result will be. In these thesis the FOKUS acts as the customer. Each step has to be discussed and approved by a representative of the FOKUS. Due to the fact that the prototype will be created from scratch most of the concepts have to be created in brainstorming meetings. An agile development process will be used to react to rapidly changing requirements. As mentioned before a Kanban like method will be used in this project.

3.1 Functional requirements

As the fundamental requirement, the prototype to be developed has to create, manage and maintain virtualized containers on a fog node. Tools like Open Baton inherently supports OpenStack as the ETSI MANO VIM layer. Most MANO tools like Open Baton uses OpenStack which deploys virtual machines to virtualize the NFs. This is a rock solid solution for a cloud environment. Unfortunately a bare-metal virtualization is most of the times not feasible on small power devices like they are used in the IoT area. Therefore a much more efficient and lightweight solution like container virtualization should be used and handled by an orchestration engine. The desired service like a NFs can be bundled in one or multiple containers and executed afterwards on the expected IoT nodes. Such a bundle of containers should be passed to the node as a build plan or a blueprint of the service. The fog node engine should accept the blueprint and deploy the containers to the desired virtualization layer. Afterwards the lifecycle of the services should be monitored.

The second functional requirement is the implementation of a constraint logic which will be used to filter relevant nodes during the orchestration. A constraint can be a functional and non-functional capability. For example a specific hardware component, like a sensor or a ZigBee dongle, which is necessary to execute the NF or a hardware requirement, like CPU power, RAM or disk space. It could also be a non-functional constraint, for example a specific

software which has to be installed or a protocol which can be used. The engine should be able to manage these constraints by itself and if necessary for all adjacent nodes and should consider them while choosing a suitable nodes for the desired NF. The whole functionality should work similar to the labels in Docker Swarm. There a Docker Swarm node can be have multiple labels, which can be considered when deploying an image. This behavior should be achieved by the fog node engine.

Another important aspect is the lifecycle management of the node, the deployed services and images. In the prototype it has to be elaborated how the lifecycle of the several components can be implemented. An sample implementation of a node management lifecycle is shown in the OpenFog Reference Architecture for Fog Computing[0, p. 52 f.]. Figure 14 describes the five steps of a typical lifecycle based on these architecture.

Figure 14: Node management lifecycle. Adapted from: [0, p. 52]

- The **Commission** is the earliest phase in a lifecycle mostly used to perform action like identification, certificates or calibration of time.[0, p. 52 f.]
- In the **Provision** phase the node will be enrolled to the system so that the node can be discovered and identified and also advertise features and capabilities.[0, p. 52 f.]
- The **Operate** phase is the state of the node when everything operates normal. This includes the reliability, availability and serviceability of the node.[0, p. 53]
- In contrast to that the **Recover** phase performs action if something operates out of norm.[0, p. 53] The node should be able to recover to the normal state.[0, p. 53]
- Finally the **Decomission** phase is used for cleaning up sensitive data on the node and to unregister from other components.[0, p. 53]

In addition to the node lifecycle, an exemplary lifecycle for services and images is shown in the ETSI MANO specification[0, p. 67 ff.] These lifecycle is much more detailed than the node OpenFog node lifecycle and it handles several state from the instantiation of a service, over querying some data, up to the termination of service. Some of the specifications are tightly coupled to NFVs. However these lifecycle specification is a good starting point to elaborate a custom solution.

The last functional component to be developed will be the GUI. Those should only be used for demonstrate the basic functionalities of the prototype. It is not designed to use them in production. Therefore the GUI will not have any security mechanisms like authorization, authentication or user management. It will only has a very basic design and a limited functionality. This includes that existing nodes will be displayed with all the related services. Additionally it can be used to deploy new services and while using the existing endpoints.

Some secondary conditions should also be fulfilled. The whole system should be modular and easy to extend. Modules should be as decoupled as possible and the whole system should be controlled via an API. The centralized cloud environment should be easily replaceable and should not be exclusively bound to Open Baton or any other tool. The same applies to the prototype, it should not be bound to Docker only and should be able to use other virtualization tools. Finally the whole system should be well tested and documented.

3.2 Non-Functional requirements

After the functional conditions for the prototype, also the non-functional requirements will also be addressed. They will be classified into the following categories:

Reliability Due to the fact that the final product will only be a prototype and has only a few development iterations and testers, it will not be production ready. Nevertheless the prototype will be developed with stability and robustness in mind and it will also be well tested and documented.

Performance As a crucial requirement, extra attention will be payed for make the application as lightweight as possible and reducing the dependencies and tool chain as small as possible. The use of powerful but resource consuming tools and libraries like huge databases or message queues will be renounced. Necessary libraries will be used if they are essential, but in general they will be selected with the requirement of being executed on low-power devices.

Usability The prototype should be easy to use, to install and to maintain. An easy to use installation script will be developed and the source code as well as the tool himself will be well documented.

Maintainability Due to the fact that the prototype will be a solid base for further development, it will be created as modular as possible. It should also be well structured by using well known design patterns. Each component should be easy to replace and also easy to maintain. The project should be open source and extensible by everyone. Code tests and code style checks will help to ensure the functionality and extensibility of the application.

Security Also an important part as in nearly every project, also security will be considered. Especially in the IoT security becomes important due to a several bad examples in the last years.Therefore particular scenarios will be considered and recommendations will be addressed. Since this is only a prototype and showing up the functionality will be in focus, some provisions will only be demonstrated, but not implemented.

Correctness As mentioned before the code will be tested and code style checks will help to have a standardized code base.

Flexibility The software will be developed with some well known standards in mind like the MANO specification, standard protocols and also design patterns to make the source code more readable and understandable.

Scalability To prototype will also be developed to be executed at a bunch of nodes at the same time. Primary usage will be in a cluster with several other nodes. This need will be considered during the whole development phase.

3.3 Use-Case-Analysis

This analysis will show up four exemplary use cases for the prototype to be developed. These use cases describes the system in a simplified and abstract manner. They will not refer to any technology but will show up the usage of the prototype from a user perspective.

Service deployment from a cloud orchestrator That will be the most common use case in this consideration. A service for example a NF should be deployed to a single node. Therefore the prototype will be installed on the node himself. A cloud service such as Open Baton will orchestrate the deployment. To do so a so called blueprint which is basically a description of the service, the images and the capabilities that are necessary to execute the image, will be passed over to the prototype via an API. The prototype will receive them and will parse them afterwards. All the provides images will be executed on the same node and at the end the service is up and running. The state of the services can be observed at any step of the process.

Service deployment based on capabilities Also in this case a cloud service like Open Baton will deploy a service to a node cluster. Again one node will get the blueprint and parse them. During the parsing of the blueprint the node does not fulfill one or multiple capabilities that are necessary to execute a specific image. If the node detects such an image, it will search for a node which is able to deploy an image with the given capabilities. For example one image needs a ZigBee dongle to measure data, so the capability *ZigBee* is added to the image in the blueprint. The first node, Node A, has no ZigBee dongle connect and therefore they is not able to fulfill that requirement. Node A will then ask all the other nodes in the cluster if they can satisfy the need. Node B will respond that they is also not able to do so. Node C is able and will respond with a related message. Node A send over the image to be deployed and Node C will execute them. From that point on, Node A is still responsible for the whole service, but Node C is executing the image and will frequently be ask for the state of the image. This process can be repeat for one or multiple images. If each image was deployed, the service is up and running and can again observed by the user.

A new node will appear If a new node will be added to the cluster, all the other nodes should be informed about this circumstance. Therefore the appearing node will be send out a message with some meta information, with for example the IP address of the node, to all the other nodes. Each node in the cluster will get the message and can store the information. Additionally the nodes will also send back a message with their meta information. Again

each node will receive these message, including the newly added node. The new node can also store the information of the other nodes. From now on, all the nodes know each other and can directly communicate with each other. A centralized entity is no longer necessary. The deregistration of a node will be similar to the registration process. The disappearing node will send out a message with meta information to all the other nodes. They will receive them and remove the node from the local storage. This disappeared node will no longer be part of the cluster.

Lifecycle management The lifecycle management is important for the maintainer of the nodes. Each process and each task in the system should be transparent and comprehensible. Therefore the different entities should be expose their lifecycle. The first one is the node lifecycle. This represents the current state of the node. Depending from the state, a node can be able to deploy a service or not. Lets assume the node is currently setting up all the necessary configurations and tools, they will then be in the state of the *configuration phase*. In this state a deployment will not be possible. Similar to that also the services and images will be have a lifecycle. When a service is deployed but does not have started all the related images, it will be in the state of *instantiating* the service. If there occurs an error while starting a container, the image and also the related service will be marked as *Error* and the deployment will fail. All of these states should also be viewable by a centralized instance like the maintainer.

3.4 Delineation from existing solutions

This section is intended to show the features of existing systems and the main differences will be highlighted. As mentioned before, the prototype to be developed should orchestrate virtualized containers on nodes based on functional and non-functional constraints. Therefore the focus of this consideration is the orchestration as well as the constraints.

Kubernetes is especially made for Docker and can orchestrate, scale and manage containers. It is open source, made by Google and one of the most popular orchestration tools out there. It has on huge and pretty active community and is used by several well known companies[0] like ebay[1] and Wikimedia[2]. Due to the fact, that is exclusively made for Docker, it means that the system is not made to easily switch the underlying container engine if needed.

As mentioned in section 2.3.3 Kubernetes supports labels. These are simple key-value pairs provided as JavaScript Object Notation (JSON) objects which can be added by the system administrator to a Kubernetes Object like a pod or a service. The labels are stored on the Kubernetes Master and can be used to filter specific pods or services during the deployment phase. This behavior is pretty close to the one which should be achieved.

Kubernetes is made for the cloud, which means it is not intended to be used on low power devices. There are some trials to do so, but until know it is not used in a productive IoT environment. Furthermore Kubernetes is not ETSI MANO compliant, but provides an easy to use web UI.

[1] `http://www.ebay.com`
[2] `https://www.wikimedia.org`

Docker Swarm is pretty similar to Kubernetes from a functional point of view, which means it has nearly the same pros and cons. It is open source as well, has a quite active community and it is also made exclusively for Docker. Biggest benefit compared to Kubernetes is, that it is build the Docker Engine. No separate installation is necessary and it can be used out of the box.

Also in terms of labels, both platforms are similar. Docker Swarm uses labels in the same way Kubernetes is using them. The user can add them during the initialization phase or edit them during runtime. They are also key-value pairs or alternatively keys only. By default labels can not be predefined in a JSON file and applied to the node afterwards. The placement have to be done by hand via the Docker client or the REST API.

Just as Kubernetes, Docker Swarm is not ETSI MANO compliant and provides no build-in GUI, but there are several third party tools out in the market. Due to the fact that it is a build-in function of Docker, the setup is quite easy and much more lightweight than Kubernetes. This means it will also work on IoT devices by default.

OpenStack is natively supported by Open Baton and will be used by default as the underlying virtualization tool. It only supports VMs and is mainly designed to be used in a cloud environment as well. The installation processes is much more complex then for Docker Swarm or Kubernetes. There are much more dependencies and configurations to be made.

Beside the huge setup effort, it is excellent for NFVs. It has a huge community, it is an established tool and very elaborated. Compared to other tools it is not flexible and lightweight enough for the usage in an IoT context or more specific for the use directly on the nodes himself. There are some efforts to move OpenStack over to the IoT infrastructure[0][0], but also in these attempts it will only be used in the cloud level.

Cloudify is completely compatible to the ETSI MANO standard and can be used as the NFVO, as well as the generic VNFM of this architecture.[0, cf.] It is also able to interact with multiple VIMs, containers, infrastructures and devices and due to the fact that it can be extended with plugins, it is can be used together with several well known tools like OpenStack, Docker or even Kubernetes.[0, cf.] Through these flexibility Cloudify can also be used in an IoT environment if an appropriate VIMs plugin is used. Downside is that Cloudify himself is very limited without a powerful underlying orchestration tool.

By default it is also not possible to orchestrate functionalities based on constraints. To enable this behavior the used plugin has to support such a functionality like Docker Swarm or Kubernetes. Cloudify provides an easy to use GUI where the user can use the whole system, as well as a clean command line tool. By using YAML files to build blueprints based on the TOSCA standard, the creation of such a blueprint is similar to well known workflows like the creation of a Ansible or Vagrant deployment schema. With the help of the Cloudify Composer the creation of a blueprint is getting much easier and also usable for users without any coding experience.

Chapter 4

Concept

This chapter introduces the architectural design of prototype respect to the previously defined requirements. Therefore the used development environment will be analyzed. Followed by the definition of an abstract t architecture of the prototype to be developed. Based on that the different layer of the system will be elaborated as well as some security concerns and the continuous integration environment. The working title of the project will be **Motey**. In the following the prototype will also be referenced as Motey.

4.1 Development environment

The development environment is crucial for both the implementation of the Motey engine, as well as the choice of possible plugins and libraries. It should be easy to used and fast to implement, but it should also consider the knowledge of the FOKUS as well as the developer.

Java is still one of the most important programming languages out there.[0, cf.] Java is platform independent, because Java will not be executed directly in the OS, instead it will be executed in the JVM. The downside is, that especially because of the JVM, Java programs are much more inefficient regarding RAM and CPU usage. Java is object oriented, which makes it easy to extend. There are a lot of well designed and developed established libraries and tools out there, which makes it easy to create rapid prototypes.

Node.js is a relatively young programming environment, which was created 2009 by Ryan Dahl and based on the Google JavaScript engine V8. The V8 engine is written in C++, which makes the language less resources consuming then for example Java. A program will be written in JavaScript, but it is also possible to load C or C++ extension. Node.js is also available for all common OSs. Due to the fact that Node.js has a build-in webserver component, it is easy to create web applications. Libraries can be added via the package manager called npm. JavaScript and also Node.js have a pretty active community, which tends to be as messy as they is increasing. There is enormous amount of libraries and tools out there, where each library has its reason to exist, but it is also a vast of pretty similar designs and use-cases.

Python is one of the most used programming languages with a huge and active community.[0, cf.] It is a dynamic typed programming language with an automatic memory management which uses both, reference counting and garbage collection at the same time.[0, cf.] Python can use several programming paradigms like the object oriented programming or functional programming. Core philosophy is make it simple, make it beautiful, make it explicit and make it readable. Currently two versions are maintained in parallel, 2.7 and the newer 3.6. Due to the fact that a huge code base is still working with Python 2.x, this version is still under development.[0, cf.] Python has a tremendous amount of libraries which can be installed via the Python package manager called pip[1]. Similar to Node.js Python allows to write C extension. Finally there are several Python compilers which compiles Python code to other high-level languages like Java, C or JavaScript.

Go is the new emerging language created by Google and was announced end of 2009. It follows the traditions of C, but also takes added concepts to create a more modern programming language. According to their own statements go is expressive, concise, clean, and efficient.[0, cf.] Furthermore it has a garbage collection and run-time reflection, it is fast, statically typed and a compiled language.[0, cf.] Go is used in production by several tools like Docker, Kubernetes and OpenShift. Downside is that there are not so much libraries out right now and there are no long time experience in the critical mass.

Open Baton, as one of the actively maintained project at the FOKUS, is written in Java and is also has ports to Python and go. Python in general is used for several projects at the FOKUS. Taking also the criteria from section 3.1 into account Python will be used as the programming environments of choice. The GUI will only be a very basic web client, which uses Vue.js[2] as its main framework and some smaller tools like the pretty famous bootstrap framework.

4.2 Architecture of the system

The overall architecture of the system can be separated into two levels. Figure 15 will point up the architecture. The first level is the *centralized fog level*. This could be for example a cloud server or management node in a fog cluster. Ideally this level should be implemented with an MANO compliant framework. Hereinafter Open Baton will referred as the tool of choice for that level. Main function will be the creation of the NFVI as well as the deployment of deployment plans for the NFs. Open Baton will also have an overview of all existing nodes and can manage and maintain them.

[1] `https://pypi.python.org/pypi/pip`
[2] `https://vuejs.org`

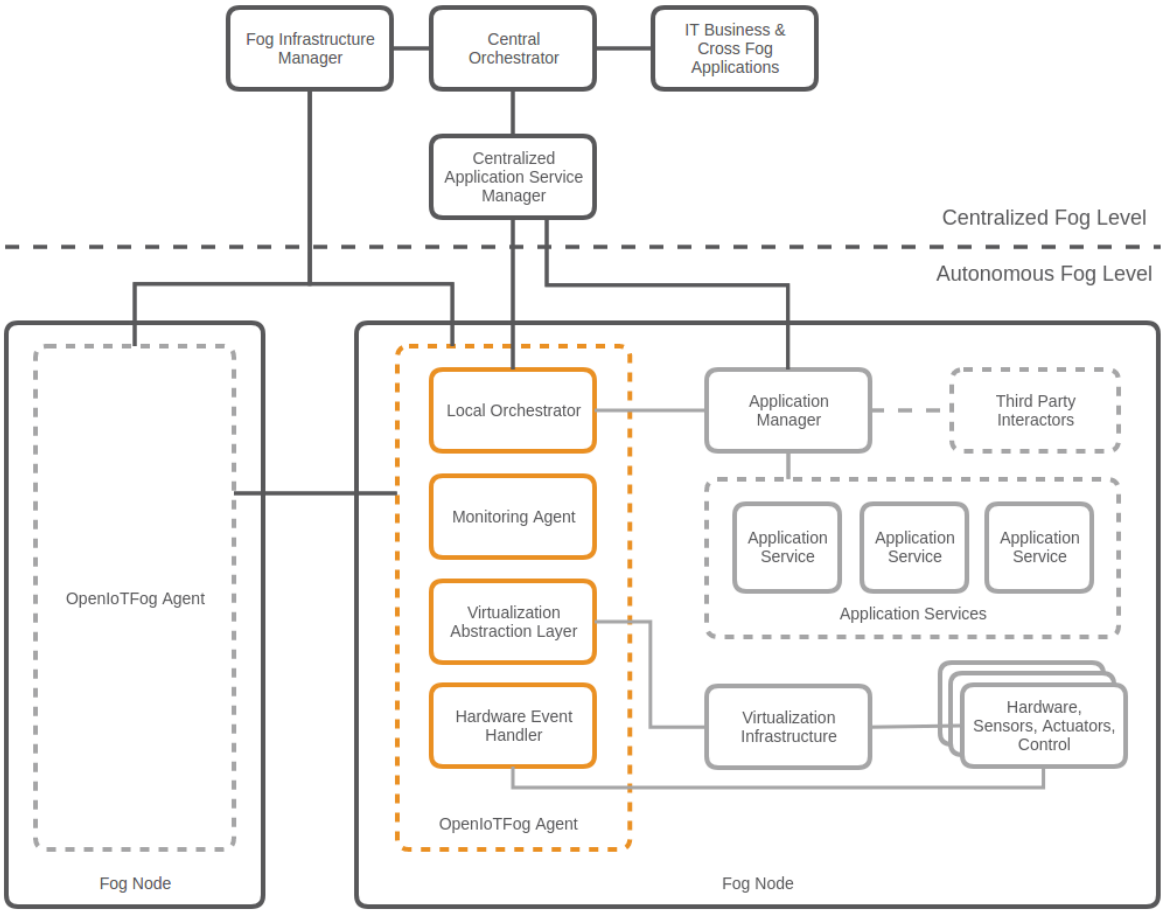

Figure 15: Abstract architecture design

The second level is the *autonomous fog level*. The Motey engine will be located in this level. It includes all existing fog nodes, as well as the Motey engine that is referenced as the *OpenIoTFog Agent* in figure 15. Each node must have a running instance of the OpenIoTFog Agent to be part of the system. Beside Motey a node can have several hardware devices, sensors and actuators connected to them. A virtualization infrastructure such as Docker or XEN, must be installed to be used by Motey. These infrastructure can create the containers and VNFs. Further it is also possible to have additional third party tools installed which can interact with the them as well.

The OpenIoTFog Agent has several external connection points. It has a REST API implemented to get some information about the status of the fog node, as well as endpoints to receive the deployment plans from centralized fog level. A MQTT connection to a broker, which can be running in the centralized fog level as well as on any other fog node, is used for node discovery. Finally there are some ZeroMQ endpoints for capability discovery, image deployment and node-to-node communication. A detailed description will be shown in the following sections.

One of the most important components is the *Local Orchestrator*. It is responsible for the deployment of the containers as well as the inter-node communication. The latter is necessary to let the nodes act in an autonomously was, so that they can react to changing requirements,

even when the centralized fog level disappears. Therefore each node must have knowledge and should be able to interact with each other. Beside that the local orchestrator is tightly coupled to the *Virtualization Abstraction Layer (VAL) Manager*. This is an abstraction layer for each virtualization component in the system, for example Docker or a bare-metal virtualization like XEN. These components should be implemented as plugins, so that it is easy to add or remove virtualization components. Therefore Yapsy[3] is used as the plugin system of choice.

The *Hardware Event Handler* is a communication endpoint for other third party components. For example an external hardware listener could send a message to the event handler to register a new connected device like a ZigBee dongle or a Bluetooth stick. This component should allow multiple third party components to send events to them. Finally the *Monitoring Agent* is used to log all ongoing events and gives the maintainer of the system an overview of the system processes.

As mentioned in the requirements section 3.1 the whole system should be modular and easy to extend. Therefore each layer will be as decoupled as possible. A clean architecture with decorated dependencies and only a single responsibility for each component. The whole code should be independent of any framework, independent of the other components and testable. To realize such an architecture, multiple design patterns will be used. Dependency Injection (DI) decouple the components and increase the testability. The decorator pattern will abstract concrete implementations by a centralized independent layer. The observer pattern can be used to handle streams of data for example from an API endpoint. And finally the singleton pattern can prevent the construction of multiple instances of a single module. All of them made the code base well structured and improve the readability as well as the comprehensibility

4.2.1 Virtualization layer

The virtualization layer is basically an abstraction layer to generalize the different virtualization engines. Therefore a so called VAL manager which will be implemented with the facade design pattern is used to load the plugins and abstract the methods of the plugins. As mentioned before to realize the plugin functionality the Yapsy library will be used. The library offers a way to easily add new plugins to the system and is also designed to be easy to use. It only depends on Python standard libraries and lightweight by design. Each supported virtualization engine needs his own concrete plugin implementation which should be use an interface to have a common ground. The supported default engine is Docker, but could be extended in a future version of Motey.

4.2.2 Communication layer

The communication layer is a pretty important component in the Motey engine. Here three different communication points are necessary to provide the basic functionalities which are needed for the fog node agent.

The first subcomponent is the **node discovery**. The idea behind the node discovery is that each node automatically can register and unregister himself to the cluster. This means in the moment of the startup of a node, they will send out a message, with an information request

[3] `http://yapsy.sourceforge.net/`

about all subscribed nodes. To realize that MQTT will be the tool of choice. The MQTT broker will receive and forward the message to all subscribed nodes and each node will response with an information message to the broker again, which will send the message out again. Therefore each node will be up-to-date at each time. As long as there is no appearance or disappearance of any node, the network will not be stressed. To provide a better flexibility the MQTT broker can be executed in the centralized fog level or even on a node in the autonomous fog level. This allows the system to operate even if the centralized fog level disappears due to network issues or any other communication problems. It is also possible to let all nodes communicate which each other once they shared their information. Smaller connection issues can be covered with this mechanism. As the MQTT broker Mosquitto[4] will be the tool of choice. It is an eclipse[5] project which means it is open source and under continuous development. The project website describes itself as "a lightweight server implementation of the MQTT protocol that is suitable for all situations from full power machines to embedded and low power machines"[0].

The next subcomponent is the **inter- and intra-node communication**. Both kinds of communication will be realized with ZeroMQ. Some of the most important patterns and transport types in ZeroMQ was described in section 2.4.2. For the node-to-node communication the *Request-Reply* pattern via TCP will be used. Typical function calls would be the capability discovery where a node will request another node for their capabilities, the deployment or termination of an image on an external node and the request for an image status. ZeroMQ always requires an IP to establish a connection to another node, therefore we had the node discovery which was described before. This allows us to connect to any other node in the cluster. In addition to the intra-node communication, also an inter-node communication will be implemented. It will be used to add new capabilities to a node. Therefore one or multiple third party applications should be connectable to a single ZeroMQ endpoint via the publish/-subscribe pattern over IPC. Different to a normal publish/subscribe pattern, the endpoint will acts as the subscriber so that multiple publishers can push messages to them. Beside that each exposed socket should be configurable via the configuration file.

The last subcomponent is the **REST API**. It will be mainly used for the communication with the centralized fog level, because most of the orchestration tools out there are using REST for transferring data. In comparison to an implementation with ZeroMQ, this is much bigger overhead in terms of traffic and latency, but to have a better compatibility with other systems, this API will be implemented. As the tool of choice Flask[6] will be used. It is a lightweight and robust Python webserver, which is open source, well documented and under constant development. The REST API himself will follow the Hypermedia As The Engine Of Application State (HATEOAS) constraint with the addition that each endpoint will have a version number in the Uniform Resource Locator (URL) to ensure backwards compatibility if something changes in the implementation.

All the mentioned subcomponents should be controlled by a so called *communication manager* which will be implemented with the facade design pattern. That decouples the communication layer from the other components, the whole system can be maintained much easier and each subcomponent can be easy replaced by any other technology if necessary. This also makes the code more readable and leads up to a cleaner code structure.

[4] `https://mosquitto.org`
[5] `http://www.eclipse.org`
[6] `http://flask.pocoo.org`

4.2.3 Data layer

As mostly the data layer is used to persist necessary data. This includes the deployed services, adjacent nodes and the capabilities of the node. As database engine the lightweight document oriented database TinyDB[7] will be used. It is easy to use and has no execution overhead and is also good to use for small datasets. The whole data layer should be as abstracted as all the other components before. Therefore repositories for each content type will facade the TinyDB methods and allows the underlying library, in this case TinyDB, to be replaced easily and without modifying several classes. The configuration of the databases should be stored in the global config file as well.

4.2.4 Capability Management

The *Capability Management* is used to create, persist, modify and remove the capabilities of the current node. As mentioned before all the capabilities will be stored in the data layer via TinyDB. Furthermore the *Hardware Event Handler* is part of this layer. It enables the system to get new capabilities from third party apps via a ZeroMQ endpoint. As mentioned in the 4.2.2 the endpoint will be implemented as a form of the publish/subscribe pattern and should allow one or multiple publishers to push messages to the handler. Also internal components like the VAL plugins can add new capabilities to the system.

4.2.5 Orchestration layer

As one of the most important layers, the *Orchestration Layer* will be a connector between all the nodes in a cluster and also between multiple components in the Motey engine himself. Primary it is used to handle the business logic of the deployment of new services. If a new service will be send to the communication layer it will be forwarded to the orchestration layer and will be parsed there. Afterwards the service will be validated and finally deployed by the orchestrator. Additionally it will also check the necessary capabilities for each service image and will search for suitable nodes in the cluster if one or multiple capabilities are not fulfilled. Beside that, the orchestrator will also handle the lifecycle of a service and the related images during the deployment phase. Figure 16 show up the lifecycle management of a service from the starting phase to the execution phase.

[7] `http://tinydb.readthedocs.io`

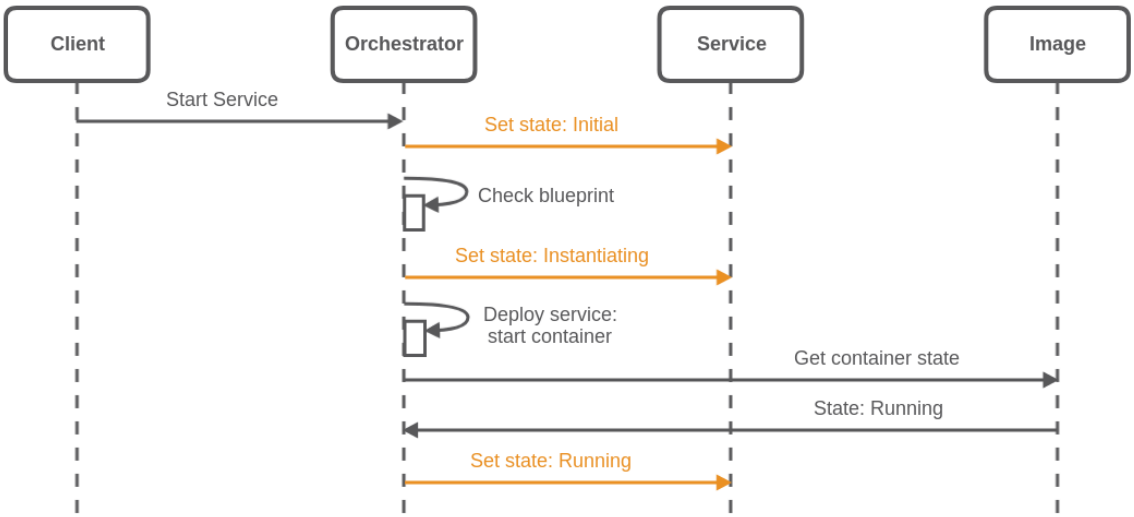

Figure 16: Lifecycle management sequence diagram: starting a service

At first a client, for example a webservices via the REST API, will start a service. Therefore the related service blueprint will be uploaded to the node. The orchestrator will directly start with the lifecycle management by creating an service with an *inital* state. Afterwards the blueprint gets checked by the orchestrator. If it is not valid, the state of the service would be set to *error*. In this case everything went fine and the state changes to *instantiating*. Next the orchestrator will start the images of the service. The state of an image can be requested at every moment during the lifecycle of a service. After the request for the container state was send, the state will be send back, for example *running*. When all images are done loading, the state of the service changes to *running*.

Later on the client probably wants to shut down the service. This time it send the terminate request to the node. Again the orchestrator will get it and set the state of the service to *stopping*. Then it will start to shutdown all the related images. Also in this phase, the state of the images can be requested every time. When all images are terminated, the state of the service changes to *terminated* and the service will no longer be executed on the node. Figure 17 illustrates this behavior.

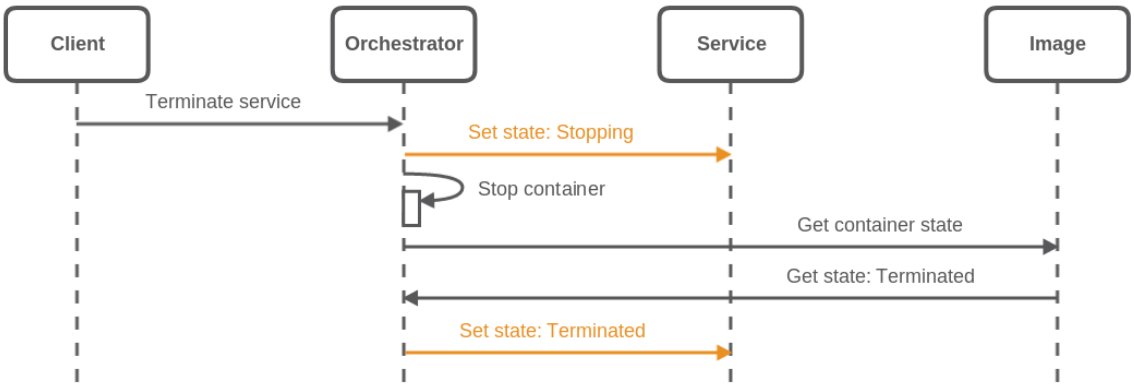

Figure 17: Lifecycle management sequence diagram: stopping a service

The lifecycle management of the images is not handled by the orchestrator himself, it will be detected via the lifecycle management of the virtualization engine like Docker or XEN. Due to the fact that the service is tightly coupled to the images, the state of the images will directly affect the state of the service. If only one image will change its state to *error* the whole service will be marked as *error*. To handle the different states of the different engines, a transformation logic between the state will be implemented.

4.2.6 User interface

As mentioned in section 3.1 the GUI will only be used for demonstration purposes and is not designed for use them in production environments. Figure 18 show the mockup of the web GUI.

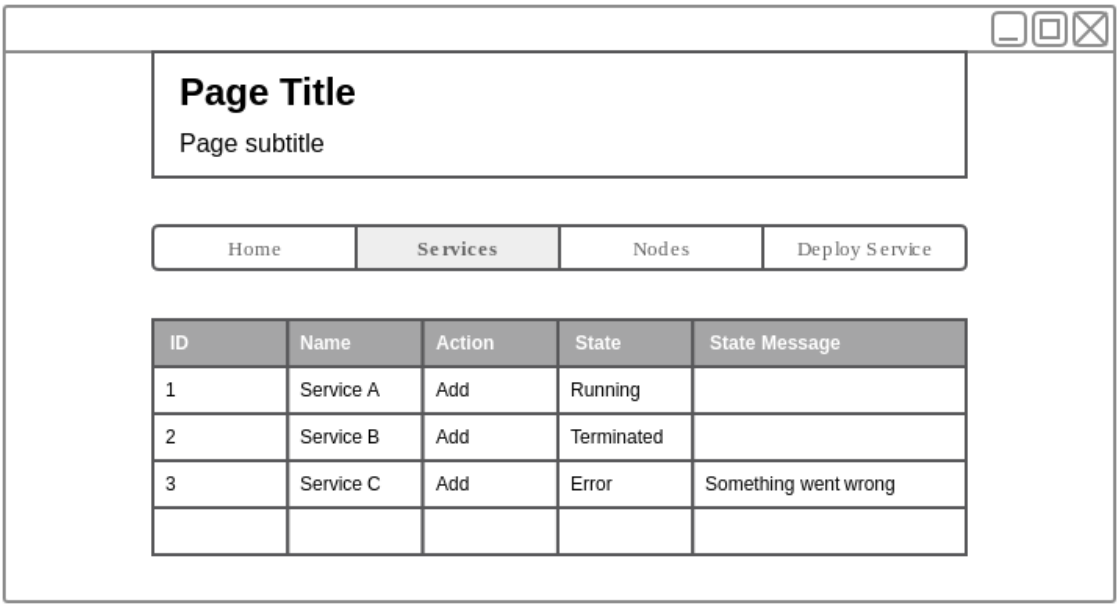

Figure 18: Mockup of the web GUI

The page has only a few components like the header area, a navigation bar and the content area. In the mockup the service tab is shown. This view should list all the deployed services. Similar to this, the *Nodes* tab should list all detected nodes and finally the *deploy service* tab should add the possibility to upload YAML data. The created REST API will be used to get all the necessary information and to deploy new services. A simple web layout based on Bootstrap[8] as the CSS framework and Vue.js[9] as the JavaScript frontend framework will be created. According to Wappalyzer[10] is Bootstrap by far the most used Cascading Style Sheets (CSS) framework out in there with a market share of 65.2% (as of 26. June 2017). It is very elaborated and there are many templates available. Vue.js on the other side is relatively new and the new rising star on the JavaScript frontend framework market. It is much slimmer then tools like Angular.js[11] or react[12] but has also less functionalities. It has a steep learning curve and optimal for small applications and rapid prototypes. Vue.js works natively together with Bootstrap so this will a rock solid solution. All the request will be done by Ajax requests to the API. There will be no authentication or authorization due to the fat that it is only a prototypical implementation for demonstration purposes only.

4.3 Security

Security is a pretty important topic, especially in the IoT context, but also in general in any infrastructure at all. Insecure IoT devices and several botnets like the Mirai botnet[13] or the

[8] `http://getbootstrap.com`
[9] `https://vuejs.org`
[10] `https://wappalyzer.com/categories/web-frameworks`
[11] `https://angular.io`
[12] `https://facebook.github.io/react`
[13] `https://www.corero.com/resources/ddos-attack-types/mirai-botnet-ddos-attack.html`

BrickerBot[14] infected thousands of devices during the last year. Nearly everyday there are new hacks and security vulnerabilities in the news, which show off that there is a lot of space for improvements in this topic. Therefore also this project should have security in mind. Due to the fact that only a prototype should be developed, not every aspect will be covered. But even if there are no implementation for securing a special component, possible attack vectors should be mentioned in this thesis. The *OWASP Internet of Things Project* list the most common vulnerabilities in IoT projects[15]. Some of them are very obvious like *insecure passwords* or *unencrypted services*, but in general the list is a good starting point to improve the security in a project. In case of the prototype the following steps should be implemented:

- secure passwords
- access control for the nodes
- access control for the database
- access control for the communication endpoints
- encryption of the communication protocols

That should be the minimal setup for securing the nodes. To make them production ready even much more improvements should be implemented. Again, just because this is a rapid prototype which should show off the possibilities of such an engine, the security will be neglected and the implementation only annotated.

4.4 Continuous Integration

Continuous integration is nowadays a frequently used technique to automate repeating deployment steps into a self executing pipeline. It starts by running unit tests, code style checks and ends up with deploying the compiled program to a server or a marketplace. Due to the fact that Github[16] will be used to host and maintain the git repository for the Motey project, the pretty famous and seamless integrated Travis CI[17] will be used to implement the continuous integration pipeline. To start with Travis CI only the Github account has to be synced with the platform and a YAML configuration file has to be placed in the root folder of the git repository. Travis CI supports several programming languages and also various third party services, like Docker Hub or Amazon AWS. Every time a new commit will be pushed to the git remote repository, a new build will be started at Travis CI. At first a virtual machine will be started by Travis CI. Afterwards all necessary components like libraries and tools will be installed in the virtual machine. Finally all predefined tasks from the YAML file will be executed. In terms of the Motey project, unit tests will be executed, as well as code style checks and if a version from the master branch will be build, a Docker container will be created and pushed to the related Docker Hub repository. This pipeline guarantees that the project is tested and a coding standard is enforced. Further the manual build of the Docker container including the upload to the Docker Hub will be obsolete.

[14] `https://security.radware.com/ddos-threats-attacks/brickerbot-pdos-back-with-vengeance`
[15] `https://www.owasp.org/index.php/OWASP_Internet_of_Things_Project`
[16] `https://github.com`
[17] `https://travis-ci.org`

Chapter 5

Implementation

This chapter describes the implementation of the Motey engine as well as the deployment and capability logic. Used technologies, libraries and tools, as well as custom components would be presented and the functionality will be demonstrated. Thereby challenges and problems during the development of the plugins will be shown and the solutions will be discussed.

5.1 Environment

To Motey engine is designed to be executed on low-power devices like a Raspberry Pi. The following software is required to execute the whole software stack.

- Ubuntu version 14.10 or higher
- Python 3.5 or newer
- Docker 17.3 or higher

On hardware side Motey will be tested on Raspberry Pis type 2 B or newer. Depending on the amount and type of the executed Docker images, the hardware specifications may can vary.

5.2 Project structure

Beside the main folder with the Motey engine, the project contains several directories with helpful scripts and tools. A brief overview about the project structure will given in this section. A detailed explanation of all files will be omitted, as this would exceed the scope of the thesis.

- **Directory: ./** has all the necessary configuration files for the different services.
  - **File: .dockerignore** is used during the build process of a Docker image. Excludes several folders during the build phase.
  - **File: .editorconfig** contains information for Integrated Development Environments (IDEs) and editors to guarantee a consistent coding style.

  - **File: .gitignore** excludes file to be tracked by the version control system git.
  - **File: .travis.yml** is used by the continuous integration tool Travis CI.
  - **File: AUTHORS.rst** a list of all contributors.
  - **File: CHANGELOG.rst** this document records all notable changes to the Motey engine.
  - **File: LICENSE** the License of the project (Apache License Version 2.0).
  - **File: main.py** can be used to start Motey in debug mode.
  - **File: MANIFEST.in** contains meta information for the Python setup procedure.
  - **File: README.rst** file to show up a short documentation on Github and will act as the starting point of the project.
  - **File: setup.py** will be used to install Motey on a local machine.
- **docs:** contains the files to create and display the documentation resources.
  - **Directory: source** has all the files to auto-generate the documentation files from the source code.
  - **File: Makefile** this file was created by the Sphinx documentation tool. By executing the *Makefile* the related documentation files will be created.
- **motey:** this folder contains the Motey main engine. It is the main Python project. The whole structure will be explained on the next pages in detail.
- **motey-docker-image:** has all the necessary files to create a Docker image.
  - **File: Dockerfile** to build the Docker image. Is analogous to a Makefile but can only be used by the Docker engine.
  - **File: setup.sh** will be executed during the build phase and will install necessary tools and can executed command line instructions.
  - **File: requirements.txt** a list with the Python requirements which are necessary to run the Motey engine and which should be installed during the build phase via pip.
- **motey-rpi-docker-image:** is pretty similar to the motey-docker-image directory but is specifically made for the Raspberry Pi image.
- **performance_test:** contains scripts that are used to perform performance tests for the evalutation chapter.
- **resources:** is a resource folder for the Github documentation. Will only be used by the *README.rst* file in the root folder and the *index.rst* file in the docssource folder.
- **samples:** contains some samples to test the functionality of the Motey engine. Is primarily a playground to test new functions.
- **scripts:** some scripts which will be executed frequently during the development phase.
  - **Folder: config** configuration files which could be used for the Mosquitto MQTT broker Docker image.

  - **File: addcapability.py** can be used to add new capability entries to a running Motey instance.
  - **File: start_test_setup.sh** can be used to start a new local Docker test cluster.
- **tests:** contains all the unit test which are executed by the continuous integration script and the Python setup procedure.
- **webclient:** this folder contains the GUI for the Motey engine. Will also be described on the next pages in detail.

5.3 Used external libraries

This section will show up some of the most important libraries used in the Motey engine. Each library will be introduced briefly and the reason for using it in the project will be shown.

daemonize allows to run a services as a daemon process. It is made exclusively for Unix-like systems. The library will create a pid file after starting the service. In the Motey engine, the file path can be configured via a configuration file. The daemon process can be controlled via a command line interface.

```
1  Motey command line tool.
2
3  Usage:
4    motey start
5    motey stop
6    motey restart
7    motey -h | --help
8    motey --version
9
10  Options:
11    -h, --help       Show this message.
12    --version        Print the version.
```

Listing 5.1: Command line interface documentation for the daemon process

After the Motey engine is installed via the setup script, this command line tool will be available in the terminal.

dependency-injector is a microframework for Dependency Injection (DI) in Python. The DI pattern allows to move the responsibility for creating a dependency from the concrete objects to a factory or a framework which creates the dependency graph. This grants the single responsibility concept for classes and makes the whole code base much easier to unit test, because a dummy object can be passed to the constructor of the class. It is also possible to mocked the object with the help of a mocking library. To realize DI in the Motey a so called *app_module.py* was created which uses the *dependency-injector* framework to create the dependency graph. Several IoC containers are created in that file and will be used by the

framework to generate the glue code. Most of the injected components are instantiated as singleton objects to guarantee that there is only one active instance of that component at a time. The implementation of the singleton design pattern is also provided by the framework. Listing 5.2 demonstrates the implementation of such an IoC container.

```
1  class DIRepositories(containers.DeclarativeContainer):
2      capability_repository = providers.Singleton(CapabilityRepository)
3      nodes_repository = providers.Singleton(NodesRepository)
4      service_repository = providers.Singleton(ServiceRepository)
```

Listing 5.2: Extract of a sample IoC container from the app_module.py

Docker Software Development Kit (SDK) The Docker Python library is a wrapper around the Docker command line tool. Every command that can be executed with this tool can also be executed from any python code. In the initializing phase the library will connect to the Docker Engine API and will perform all the actions through them. This can be realized via an URL to the REST API or via an Unix system socket connection. In the Motey engine the second method will be used, but can be replaced without any limitations. The library is used as a VAL plugin and will be automatically loaded at runtime via the VALManager and the Yapsy plugin system.

Flask is a framework to create web applications. Flask does not provide any templating or database engine, nor does it enforce a specific file structure. Instead it will support extensions to add functionalities like that so that the developer can choose the tools of choice.[0, cf.] Nevertheless Flask is production ready and is used in several big projects like Pinterest[0] or Twilio[0].

Flask can use so called *Blueprint* to configure new routes in the webserver. A Blueprint is basically a Python class that can define methods like *get* or *post* to handle the specific HTTP verbs. This is useful to create valid HATEOAS REST APIs. Each Blueprint will be represented by an URL endpoint.

Listing 5.3 illustrates the implementation of all API endpoints in Motey.

```
1      def configure_url(self):
2        self.webserver.add_url_rule('/v1/capabilities',
   ↪  view_func=Capabilities.as_view('capabilities'))
3        self.webserver.add_url_rule('/v1/nodestatus',
   ↪  view_func=NodeStatus.as_view('nodestatus'))
4        self.webserver.add_url_rule('/v1/service',
   ↪  view_func=Service.as_view('service'))
5        self.webserver.add_url_rule('/v1/nodes',
   ↪  view_func=Nodes.as_view('nodes'))
```

Listing 5.3: Implementation of all Flask API endpoints in Motey

In line 2 a new endpoint will be add via the *add_url_rule* to the Flask webserver. The

first parameter indicates the endpoint URL and the second parameter *view_func* represents the Blueprint class, in this case *Capabilities*. To pass them over, the Blueprint has to be converted to a Flask view by using the *as_view* method. The other endpoints are implemented equivalent.

Logbook is a small logging library that helps to standardize the output of log messages. It helps to address several output methods like the terminal, a file or even emails and Linux desktop notifications. The style of the resulting message can be easily configured and it can be integrated into several other libraries. In addition to that, Logbook has a build-in support for messaging libraries like ZeroMQ, RabbitMQ or Redis. This allows to distribute log messages on heavily distributed systems like a huge node cluster. It was created by Armin Ronacher the creator of Flask and Georg Brandl the creator of Sphinx, both are tools that are used in Motey. Unfortunately there is no build-in support in Flask yet. In the Motey engine Logbook will be extended by a wrapper class to simplify the configuration of the tool. The output folder for the log messages can be configured via the global config file and will be loaded in the constructor of the wrapper class. If the folder path does not exist, it will be created.

paho-mqtt is the python implementation of the Eclipse paho[1] project that is basically the implementation of the MQTT messaging protocols, which was already described in section 2.4.1. The library allows to connect to a MQTT broker like the Mosquitto broker. It also comes with a variety of helper methods to eases the usage. A wrapper class to centralize the usage of the library was created and the configuration as well as some smaller improvements was made in this wrapper class. The whole configuration of the client can be configures via the global config file again. Furthermore the routes are managed in the wrapper and a after connect handler was implemented. It will be used to perform actions after a successfully created connection to the broker was made and all subscriptions to topics are done. This helps to realize the node discovery mechanism described in section 4.2.2.

pyzmq is the third important communication library. It is the official Python binding for ZeroMQ. A detailed description of ZeroMQ can be found in section 2.4.2 and in the great ZeroMQ guide at `http://zguide.zeromq.org/page:all`. This library is also abstracted by wrapper class in Motey. This helps to configure the ZeroMQ server and register all necessary nodes. A detailed explanation of the internals will be discussed in the following section 5.4.4.

Sphinx is a tool to auto-generate a documentation out of the source code documentation. It supports several output formats like Hypertext Markup Language (HTML), LaTeX or ePub and is the de facto standard in Python. The documentation hosting platform Read the Docs[2] completely supports Sphinx documentations. As mentioned before the Makefile in the docs folder will be used to auto-generate the documentation files. The scripts handles also the deployment of the documentation. Therefore the files will be generated, the current branch will be switched to *gh-pages*, which is be used to display the Github page at `https://neoklosch.github.io/Motey/` and a new commit will be pushed with an auto-generated commit message. Finally the branch will be switched back again. Read the Docs has an

[1] `http://www.eclipse.org/paho`
[2] `http://readthedocs.org`

active webhook that builds the builds the current documentation and display them at `http://motey.readthedocs.io`. The documentation is also used as the official Github page of the project at `https://neoklosch.github.io/Motey`.

TinyDB is a wrapper to implement a lightweight document oriented database. It stores the data into single JSON files. The location can be configured via the global config file. TinyDB only supports very basic functionalities. For example it does not support indexes or relationships and it is not optimized concerning performance. But it is easy to use, has no execution overhead and it performs very well on smaller datasets. The main purpose of the library is to be used for small apps where database server like MySQL[3] or MongoDB[4] will be a huge overhead. Furthermore TinyDB has several extension to add more functionalities like indexing or caching. It also allows to easily extend the library with custom middlewares and extensions. In the Motey engine, TinyDB is used in every *repository* to decorate the usage of the library. Thereby the used library can easily replaced by a different one, without refactoring several class in the project. A detailed description of the implementation will be shown in 5.4.2.

Yapsy is a plugin system that was designed to make an application easily extensible and should also be easy to use. Several plugin systems are too complicated for a basic usage or have a huge dependency overhead. Yapsy claims to be different, because it is written in pure Python and can be used with only a few lines of code. In the Motey engine all VAL plugins will be loaded via Yapsy. An extract of the VALManager with the method to register the plugins, is shown in listing 5.4.

```
1   def register_plugins(self):
2     self.plugin_manager.setPluginPlaces(
3       directories_list=[absolute_file_path("motey/val/plugins")]
4     )
5     self.plugin_manager.collectPlugins()
6     for plugin in self.plugin_manager.getAllPlugins():
7         plugin.plugin_object.activate()
```

Listing 5.4: Extract of the VALManager with the method to register plugins

In Motey there is a specific folder where all images has to be located (line 2). This could be extended in the future if necessary. Afterwards all the valid plugins will be loaded and activated (line 3). Finally for all activated plugins the *activate* method will be executed, which is a custom implementation to call some functions after activating a plugin (line 4 and 5). All plugins can be used via the *self.plugin_ manager*.

[3] `https://www.mysql.com`
[4] `https://www.mongodb.com`

5.4 Important Implementation Aspects

This section will introduce to the most important aspects of the implementation of the Motey engine. At first a short overview of the whole class structure will be shown, followed by a detailed explanation of the major components. Finally the created GUI as well as the CI pipeline will be explained.

5.4.1 Motey engine

The main component in the Motey engine is the *Core* class. It will start all the necessary components to run the engine. The core can be executed in debug mode or as a daemon. Latter creates a pid file which can be configured via the *config.ini* file. The dependencies of the most important components is shown in figure 19.

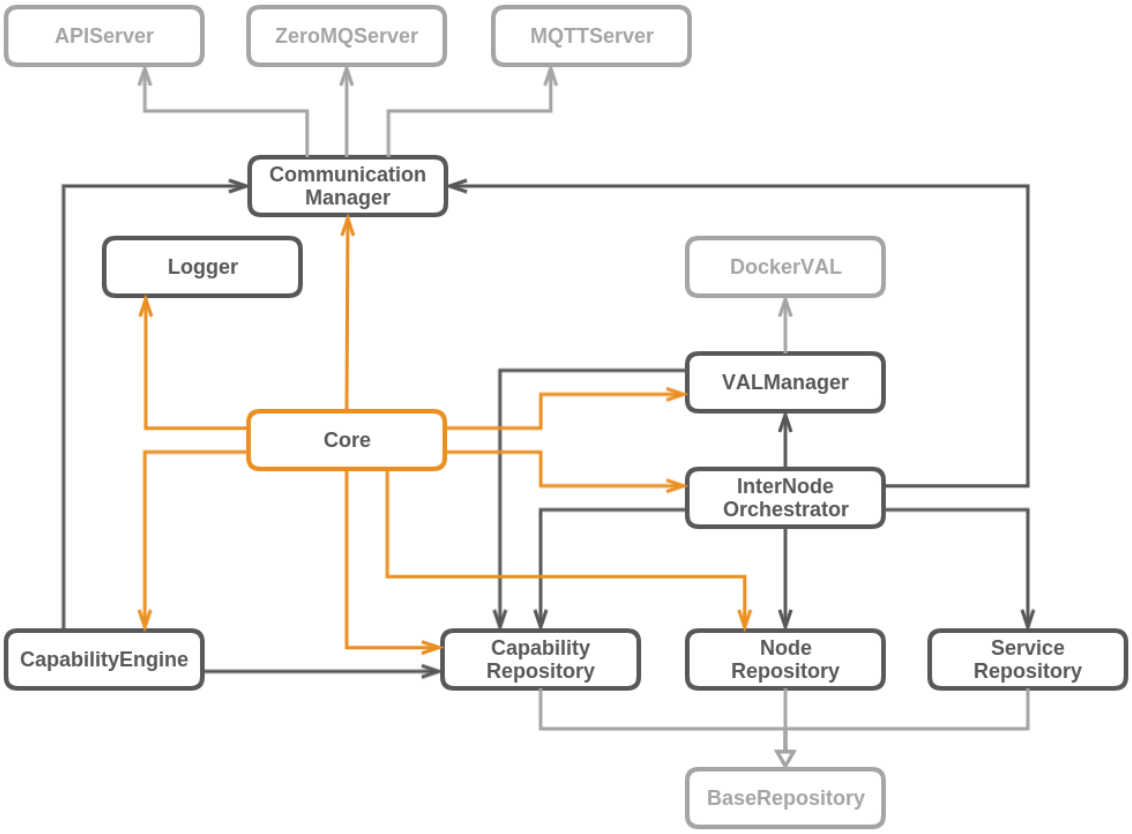

Figure 19: Motey class diagram

This class diagram is simplified in a way that not all components and not all dependency connections are shown. But it is pretty helpful to get an basic understand of the interconnection of the different layers. A good example of the decorator pattern can be found in the top left corner of the diagram. The *CommunicationManager* acts as a decorator for all the connection endpoints. Therefore it is easy to add a new endpoint or replace an existing one, only the *CommunicationManager* has to be modified instead of all the classes which uses the

communication layer. Another example for that behavior is the *VALManager*. Also this class acts as a decorator and manages all virtualization plugins, in this case the *DockerVAL*. The repositories are a special case of the decorator pattern, because they covers the usage of the *TinyDB* library, but they are also a centralized place for using them instead of all over in the engine. The *CapabilityEngine* and the *InterNodeOrchestrator* are the only component that uses the *CommunicationManger*. The *Core* also imports that class but only to start them. Beside the *Core* the *InterNodeOrchestrator* is the central place in the app. Each event will be executed in the layer. Therefore most of the other components will be come together in this class. In the following the separate layers are described in detail.

5.4.2 Data layer

As mentioned before TinyDB is used as the database engine of choice. To abstract TinyDB from the rest of the source code, the repository pattern will be used. Each content type has its own database and also a related repository. All of them are inheriting from the *BaseRepository*. This repository is used to implement some default methods and will also create the database path in the constructor if it is necessary. Beside that all of them are pretty similar implementation-wise. They only differ in using different models and have some specific methods to be executed. These models are used to represent the related JSON objects. Therefore each of them have a static transform method to convert JSON objects to the related model. The models folder also contains a file called *schemas.py*. This file contains the validation schemes for the different YAML and JSON objects that could be received by the nodes. A library called *jsonschema* is used to validate the objects based on such a schema. An example for a schema is shown in listing 5.5.

```
1  capability_json_schema = {
2      "type": "array",
3      "items": {
4          "type": "object",
5          "properties": {
6              "capability": {
7                  "type": "string"
8              },
9              "capability_type": {
10                 "type": "string"
11             }
12         },
13         "required": ["capability", "capability_type"]
14     }
15 }
```

Listing 5.5: Capability JSON validation schema

The *jsonschema* library allows to validate the type of an entry (line 2, 7 and 10) and also if the fields are required or optional (line 13). This guarantees that only valid objects will be processed.

Another data layer component is the configuration reader. This is implemented with the default Python *configparser*. The parsed configuration is only kept in memory. The configuration file for the whole project is stored in the *motey/configuration* folder and is called config.ini. Listing 5.6 show the sample content of that file.

```
1  [GENERAL]
2  app_name = Motey
3  pid = /var/run/motey.pid
4
5  [LOGGER]
6  name = Motey
7  log_path = /var/log/motey/
8  file_name = application.log
9
10 [WEBSERVER]
11 ip = 0.0.0.0
12 port = 5023
13
14 [MQTT]
15 ip = 172.18.0.3
16 port = 1883
17 keepalive = 60
18 username = neoklosch
19 password = neoklosch
20
21 [DATABASE]
22 path = /opt/Motey/motey/databases
23
24 [ZEROMQ]
25 capability_engine = 5090
26 capabilities_replier = 5091
27 deploy_image_replier = 5092
28 image_status_replier = 5093
29 image_terminate_replier = 5094
```

Listing 5.6: Example of the config.ini file

Different sections can be separated by squared brackets like *[GENERAL]* like on line 1. The entries are simple key-value pairs. Line 3 for example set the path to the used pid file. The usage of the configreader is shown in listing 5.7.

```
1  from motey.configuration.configreader import config
2
3  daemon = Daemonize(
4      app=config['GENERAL']['app_name'],
5      pid=config['GENERAL']['pid'],
6      action=run_main_component
7  )
```

Listing 5.7: Example of the usage of the configreader

The configuration object must be imported from the configreader module (see line 1) and can be used directly afterwards (line 4 and 5). Due to the implementation of the import logic in Python, the containing script will be executed only once, regardless how many files import that module.

5.4.3 Orchestration layer

The so called InterNodeOrchestrator contains the main business logic of the application. It is the connector between the communication layer or more specific between all inter-node and client communication endpoints and the VAL. The orchestrator uses the observer pattern to interact with the communication layer. At the startup of the orchestrator it will subscribe to an observer for example for receiving a new service. If a service event occurs, it will start the *instantiate_service* method. This will be executed in a separate thread. At first the service will be stored in the data layer. The lifecycle will be changed to the *instantiating* state. Afterwards the capabilities of each contained image has to be checked. This is necessary to identify the node that is handling the image. There are three different possibilities.

1. there are no capabilities located in the image, therefore the current node starts the image.
2. there are capabilities and the current node is able to fulfill all of them. The same node is handling the image.
3. there are capabilities but the current node is not able to fulfill one or multiple of them. Another node has to be search to manage the image.

The capabilities of the current node are fetched via the *CapabilityRepository*. In each case the an image will always get an IP address of the node that is handling them, even if it is the current node. Later on the communication layer will deploy the image via an ZeroMQ connection. This unifies the deployment process. If the current node is not able to fulfill all required capabilities another node in the cluster will be searched to run the image. Therefore the *find_node* method is going to be used. This method will at first fetch all the known nodes from the *NodeRepository*.
Afterwards it will send out a *capabilities request* again via the communication layer. That is a ZeroMQ request-reply call between two nodes. The targeted node sends back their capabilities. They will be passed back to the orchestrator which compares them with the capabilities of the image. If all of the are fulfilled then, this node becomes responsible for that image. This means the IP address of the node will be stored in the image. If the node is not able to accomplish them, the next node is requested. When non of the nodes suits, the state of the deployment will be marked as *error* and cancelled at the same time. Otherwise the deployment phase starts. Each image will be passed via ZeroMQ call to the related IP address to a deployment endpoint, which will then call the *VALManager*. The latter will then

instantiate the image via the related VAL plugin. When all images finally started the state of the service will change to *running* and the deployment successfully finished.
The state of the service is also handled by the orchestrator. It depends highly on the state of the images. If at least only a single image is in a *error* state, the whole service will be marked as erroneous. This behaviour is similar for the other states. Based on the MANO service lifecycle the following states was created:

- **INITIAL** - The service is created, but no other action was performed so far.
- **INSTANTIATING** - The deployment phase was started, but is not done yet.
- **RUNNING** - All images are deployed and the service is running. Everything works normal.
- **STOPPING** - The termination phase was executed, but is not done yet.
- **TERMINATED** - All images are terminated. The service no longer exist.
- **ERROR** - An error occurred in one of the other states. A detailed description is stored in the service.

All of them are located in a model object called *ServiceState*.
If a service state request is performed by the client, the request will then be received in the communication layer and passed over to the orchestrator. Afterwards The *get_service_status* method is executed which maps the states of the images to the service state. Listing 5.8 shows this method.

```
1   def get_service_status(self, service):
2       image_status_list = []
3       for image in service.images:
4           image_status =
    ↪   self.communication_manager.request_image_status(image)
5           image_status_list.append(image_status)
6
7       if Image.ImageState.ERROR in image_status_list:
8           service.state = Service.ServiceState.ERROR
9           self.terminate_service(service=service)
10      elif Image.ImageState.TERMINATED in image_status_list:
11          service.state = Service.ServiceState.TERMINATED
12          self.terminate_service(service=service)
13      elif Image.ImageState.STOPPING in image_status_list:
14          service.state = Service.ServiceState.STOPPING
15          self.terminate_service(service=service)
16      elif Image.ImageState.INSTANTIATING in image_status_list:
17          service.state = Service.ServiceState.INSTANTIATING
18      elif Image.ImageState.INITIAL in image_status_list:
19          service.state = Service.ServiceState.INITIAL
20      elif len(image_status_list) > 0 and image_status_list[1:] ==
    ↪   image_status_list[:-1] and \
21              image_status_list[0] == Image.ImageState.RUNNING:
22          service.state = Service.ServiceState.RUNNING
23      else:
24          service.state = Service.ServiceState.ERROR
25
26      self.service_repository.update(dict(service))
27      return service.state
```

Listing 5.8: The mapping of the service lifecycle state.

Important in this method is that each status will again be requested via an ZeroMQ endpoint. Also in this case a request-reply call between the nodes will be executed. Line 4 show this request. As all states are received and stored the mapping starts. Line 10 up to 12 show the mapping of the *terminated* state. At first the list with all image states will be searched for this particular state. If it is in the list, also the service will be marked as terminated. To make sure that all the related service images are terminated, not only a single one, all the other images will be terminated as well. Line 12 shows the termination of the service. This behavior is also implemented for the *error* and *stopping* state. All the other states are mapped directly to the service. Finally the new state will be stored in the *ServiceRepository* and the state returned (line 26 and 27).
The termination of a service is pretty similar to the creation beside the fact that there is no capability comparison and node retrieval. The image termination command will directly passed over to the related node. Also this method will be executed in a separate thread to not block the main thread.

5.4.4 Communication layer

As mentioned in section 4.2.2 the communication layer is splitted up into three different components as is decorated by an extra layer called *CommunicationManager*. The important implementation details of all of them will be described in this subsection. All the necessary communication components are located in the *motey/communication* folder.

APIServer This server is responsible for the REST API. Therefore the Flask server will be instantiated and configured in the constructor of the wrapper class. The server will be executed in a separate thread due to the nature of a webserver to block the main thread because the it will run endless to receive all incoming requests. If it would not be implemented with a thread only the Flask server would be started and the following code would be blocked. Furthermore Flask has to be configured to accept cross-site requests, by disabling the *same origin policy* with a *Cross-Origin Resource Sharing (CORS)* library. This behavior is a development only feature and it is strictly recommend to deactivate it in production mode. By deactivating it the server is vulnerable for Cross-Site Request Forgery (CSRF) and clickjacking attacks. In the development phase cross-site requests should be allowed to make it easier to communicate between a web client and the REST API.
In addition all the configured API will be initialized. As mentioned before these routes can be implemented as Flask Blueprints. All routes are located in the *motey/communication/api_routes* directory.
There are four different routes:

- The **Capabilities** Blueprint which is used to send the capabilities of the node, add new capabilities or to remove them. The HTTP verb *GET* is used to deliver all existing capabilities. If a request is received, the *CapabilityRepository* will be used to fetch all capabilities and then they will be converted to a JSON string afterwards. The HTTP verb *PUT* will add new entries to the repository. After the JSON request will be received, the content will be parsed and validated with the corresponding JSON schema. If it is not valid or the content type of the request is not *application/json* a HTTP status code 400 is returned. Otherwise the capabilities will be added via the *CapabilityRepository* to the database. This will end up in the HTTP status code 201. The same logic will be used for the *DELETE* request.
- The second endpoint is implemented as the **Nodes** Blueprint. The only functionality is to respond with all stored nodes as JSON. This is used for testing purposes and could also be helpful for maintaining the cluster.
- To get some information about the node health status the **NodeStatus** Blueprint was created. It respond with some hardware information like the current CPU or memory usage. This is also useful for maintaining the nodes.
- The last Blueprint implementation is the **Service** endpoint. It is used to get a JSON list with all stored services via the HTTP *GET* verb and also to deploy and remove services. Therefore the *POST* or *DELETE* verb is used. Both implementations are pretty similar. Also in this case the provided YAML file will be validated. If it is valid the parsed service will be handed over to the orchestration layer and a 201 will be returned to the client. If something went wrong a 400 will be returned.

MQTTServer The purpose of using MQTT in the Motey engine is the node discovery. Section 4.2.2 describes the basic idea of it. To implement the logic the node must have knowledge about the IP and port of the MQTT broker as well as the authentication credentials

if they are required. They can be configured via the global configuration file. As all the other classes, the *MQTTServer* is a wrapper class for the Python MQTT library. In the constructor the routes will be defined as well as some callbacks and the client will be configured. The *start* method connects the client to the broker and execute the request loop. That is pretty similar to the implementation of the *APIServer*. Therefore the MQTT client has to be executed in a separate thread too.

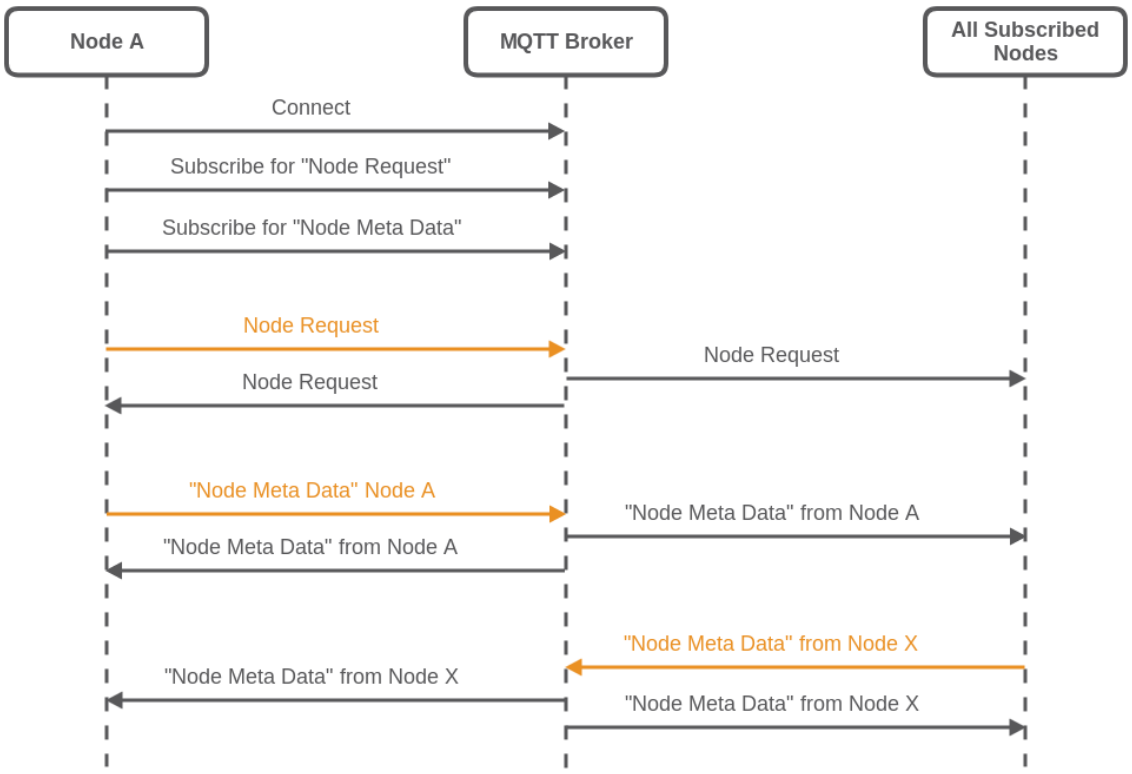

Figure 20: Node discovery sequence diagram

Figure 20 shows how the node discovery will be implemented. Important aspect is that the sender node is also part of the subscribed nodes. This means after the *connection* to the broker is established and the client is subscribed to the topics, each *Node Request* will also be received by the sender himself. Benefit out of it, is that beside the fact that a new registered node gets all meta information from all other nodes, also these nodes get the meta information of the new node. In this way each node has knowledge of all the other nodes and can keep track of them. That is also the reason why the *"Node Meta Data" from Node A* and *"Node Meta Data" from Node X* are duplicated in figure 20. Beside that the procedure is straight forward: At first one node send out a *Node Request*. Each subscribed node will get it and send out a response with the *Node Meta Data* to all the other nodes via the broker. The *after_connect* handler will be used to send out the *Node Request* after a new nodes is successfully subscribed to the broker. Before a node disconnects, a *Remove Node* request will be send out, that will inform each node that a specific node will be disappear. All the nodes react to this by removing the meta data about this node from the database.

ZeroMQServer Section 2.4.2 and 4.2.2 gave a detailed overview about ZeroMQ and how does it work. Now the concrete implementation will be discussed. The wrapper class *ZeroMQServer* binds multiple sockets to the related ports. There are four sockets binded for the direct node-to-node communication via TCP and one port to connect third party applications to the Motey engine. The latter is used to add or remove capabilities on a node. ZeroMQ provides an IPC protocol for such a use case. The ZeroMQServer will bind a socket with the Publish-Subscribe pattern that allows multiple publisher to connect to the endpoint. Two important aspects have to be considered by using the Publish-Subscribe pattern in ZeroMQ. As in MQTT each subscription has to be done to a topic. Listing 5.9 show the subscription for the *capabilityevent* at line 3. This also means that it is possible to send multiple different events to a single socket endpoint. In Motey this feature is not used yet, but nevertheless it is necessary to subscribe to a topic because of the implementation specification of ZeroMQ. Without that, the subscriber will receive nothing.

```
1   def start(self):
2     self.capabilities_subscriber.bind('ipc://*:%s' %
    ↪ config['ZEROMQ']['capability_engine'])
3     self.capabilities_subscriber.setsockopt_string(zmq.SUBSCRIBE,
    ↪ 'capabilityevent')
4     # [...]
5
6   def __run_capabilities_subscriber_thread(self):
7     while not self.stopped:
8         result = self.capabilities_subscriber.recv_string()
9         topic, output = result.split('#', 1)
10        self.capability_event_stream.on_next(output)
```

Listing 5.9: Example of the usage of the configreader

The second important aspect is at line 9. An incoming message has always been parsed for a delimiter. The reason for that is, that the topic, in this case *capabilityevent* have to be prepend to the message followed by the delimiter. Listing 5.10 show such a message.

```
1   capabilityevent#{'capability': 'zigbee', 'capability_type': 'hardware'}
```

Listing 5.10: Example ZeroMQ capability event message

Therefore the topics and the delimiter has to be removed before the message can be parsed. The other four sockets are implemented with the Request-Reply pattern and are used to:

- reply to a capability request. This is used by the *InterNodeOrchestrator* to get the capabilities of other nodes. The result is send as a JSON string. If there are no capabilities an empty JSON array will be send.
- deploy images is also used by the *InterNodeOrchestrator* to deploy a single image to another node. The image model will be transformed to a JSON object and send over as a string and vice versa and passed over to the *VALManager* if received.
- return the status of an image. This is also used by the *InterNodeOrchestrator* when the state of a service should be fetched. To current state of an image will be detected by the virtualization engine. Afterwards it will transformed to a unified image state. Finally

the InterNodeOrchestrator maps the state of all images to the service state.
- terminate an existing image instance. Only the image id has to be send to perform that action. The *VALManager* will handle the termination of the instance.

Important aspect while using the Request-Reply pattern, there can only be one connection established at the same time. In addition a reply must send by the consumer after a request was receive. As long as there was no message send back, the connection will be blocked and the consumer can not receive any new messages.
As discussed by the other servers, also this one have to be executed in threads. Difference here is that each socket has its own thread. This is necessary because each connection waits for a request and has its own request loop. Therefore each loop would block the main thread.

CommunicationManager Finally the *CommunicationManager* is used to decorate the servers from the rest of the source code. This is helpful for decoupling the components as well as replace or add a new communication component. Beside that the manager simply forward the methods to the specific server. It is also used as a central place to start and stop all servers.

5.4.5 Capability management

Similar to the InterNodeOrchestrator the *CapabilityEngine* is used as a connector between the communication layer and the in this case CapabilityRepository. Therefore the main components of the capability management are the CapabilityEngine and the *ZeroMQServer*. The latter was extensively described in the previous section and is mainly used to receive new capabilities or to remove them after a request via one of the two ZeroMQ endpoints. The CapabilityEngine on the other side has two subscriptions to the observer located in the ZeroMQServer. These subscriptions reacts to add and remove requests of any external application. Due to the fact that the endpoints are implemented via the ZeroMQ IPC protocol only application that are located on the node can interact with the CapabilityEngine. After a new add service request was received the JSON data will be parsed, validated and transformed to the capability model and afterwards stored via the *CapabilityRepository* to the database. The Removal of an entry is pretty similar. The received JSON data will be again validate with the *capability_json_schema* from the schemes model.

5.4.6 User interface

The whole Motey GUI is located in a separate folder called *webclient*. Due to the fact that the GUI is only for demonstration purposes, it should not be part of the main engine. Vue.js the used web framework is pretty lightweight, therefore only two files are necessary to execute the client. The *index.html* file contains the skeleton of the page. In addition to that the *main.js* file that is located in the *js* folder contains the business logic of the page. Figure 21 shows the final GUI.

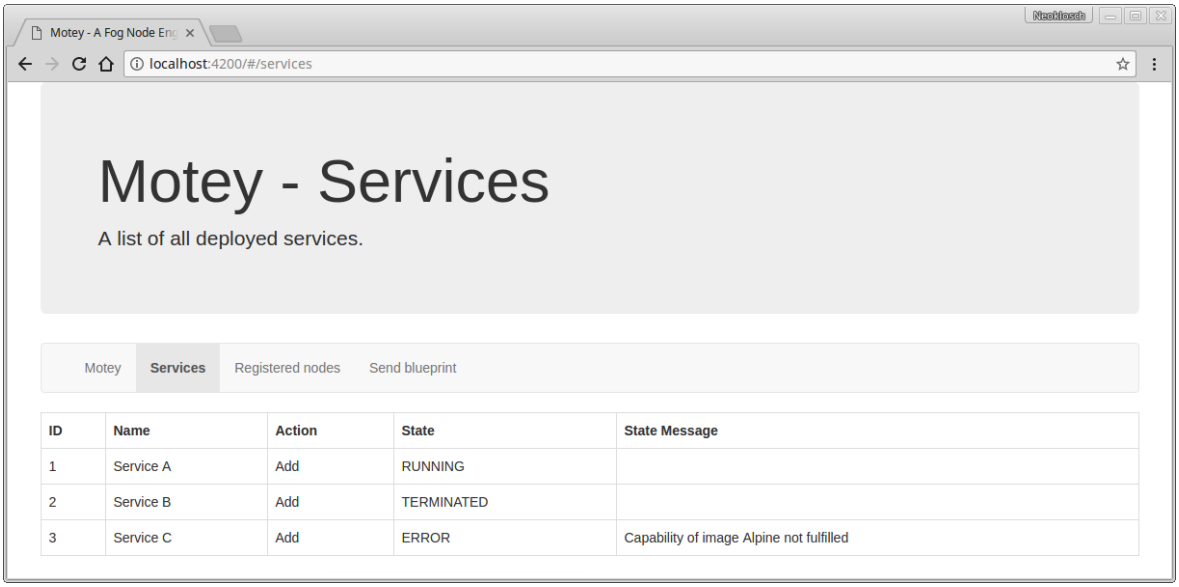

Figure 21: Screenshot of the Motey GUI

The header is fixed and only contains the name of the engine and a short description of the page content. Below them the navigation bar located. As in the mockup four different pages are available. The first called *Motey* has a very short introduction into the web client. The second and also the selected one in the screenshot shows a list with all deployed services. *Registered nodes* will show a list with all discovered nodes, again in a table. Finally the *Send blueprint* contains a textarea to put in the YAML service description and a button to send the data.
The *index.html* file contains also the files for the bootstrap and Vue.js libraries. They are loaded from a Content Delivery Network (CDN) provider to speed up the loading time. This is possible because if the user already visited a page that uses the same CDN provider and the same libraries, they will be cached in the browser internal cache. If the user then visits another site that try to load the files, they will be taken from the cache instead of requested again. This increases the page loading speed as well as reduces the traffic. Another benefit is that the files does not have to be stored on the server.
Beyond that the *main.js* file initializes the routes to handle the navigation bar redirects. Each route has an own controller similar to the Blueprint concept of the Flask server. The *NodesListing* and *ServiceListing* controllers mainly fetches the JSON data from the REST API and map them to the related template in the HTML file. The *BlueprintTemplate* controller get the data from the textarea and send them over to the API. Finally the *Content-Type* of the request will be set to *application/x-yaml* as specified in the server endpoint. As defined before, there is no user authentication or other access control mechanisms due to the fact that this GUI is exclusively made for testing purposes.

5.4.7 Deployment and Continuous Integration

Travis CI as the tool of choice for the CI uses a *.travis.yml* configuration file. The content of the file is shown in listing 5.11.

```
1  sudo: true
2  services: docker
3  language: python
4  os: linux
5  cache: pip
6  python:
7    - "3.6"
8
9  install: "pip install -r motey-docker-image/requirements.txt"
10
11 script:
12   - pycodestyle --ignore=E241,E501 motey/
13   - pycodestyle --ignore=E241,E501 samples/
14   - python3 -m unittest tests/capabilityengine/test_*
   ↪ tests/communication/test_* tests/models/test_*
   ↪ tests/orchestrator/test_* tests/repositories/test_* tests/utils/test_*
   ↪ tests/val/test_*
15
16 after_success:
17   - if [[ "$TRAVIS_BRANCH" = "master" ]]; then
18     docker build -t neoklosch/motey motey-docker-image;
19     docker login -u="$DOCKER_USERNAME" -p="$DOCKER_PASSWORD";
20     docker push neoklosch/motey;
21     fi
```

Listing 5.11: Travis CI configuration file

There are some important parts in it. At first the script need root privileges to build and upload Docker images. Line 1 add this privilege and in line 2 the Docker service is requested. The programming language the application to build is written in, is declared in line 3, in this case python. Line 6 and 7 defines the python version to use. There could be multiple, but due to the fact that the script should only be build one Docker image and should only upload them once, only one python version can be used. Otherwise multiple python version would be executed and after each successfully build version an container would be build and uploaded. Line 9 defines the installation script. This will executed after the virtual environment are created by Travis CI. In this case the python requirements will be installed. Line 11 up to 14 are used to test the source code. A code style check is performed in line 12 and 13. It will check to code against the Python Enhancement Proposal (PEP) 8 standard[5], which is be used by several other libraries like the pretty famous Django project[6]. After that all the related unit tests will be executed. If there is an error during the execution of the *script* block, the whole build process will be stopped and marked as error. The current state can be view in Travis CI at `https://travis-ci.org/Neoklosch/Motey` and also in the readme of the project at `https://github.com/Neoklosch/Motey` or `http://motey.readthedocs.io/en/latest`.
The *after_success* block is used to create and upload the Docker image. Beside that the install routine and also the CI commands are similar. The image will only be executed if it is a master branch build (see line 17). This is reasonable because a development branch build

[5] `https://www.python.org/dev/peps/pep-0008`
[6] `https://docs.djangoproject.com/en/dev/internals/contributing/writing-code/coding-style`

can be unstable and should not be used to create an official Docker image. Line 18 will build the image and line 19 uploads it. Two environment variables are used here, because to upload an image, the username and the password of the Docker Hub account has to be passed. Due to the fact that this file is under public version control, this sensible information should not be commited. Therefore preconfigured login credentials are used. Finally the Docker image will be pushed in line 20.
In the root folder of the Motey project there are two docker image directories. The first one can be used to create and upload a normal Docker image and it is called *motey-docker-image*. That folder is also used in the *after_success* block. The second folder can be used to build and upload a specific Raspberry Pi Docker image. Both scripts are slightly different, because a Raspberry Pi is using a CPU with an ARM architecture. Therefore Docker images that should support ARM CPUs must have specific dependencies and has to be build differently. To build an image, a so called *Dockerfile* has to be created. The Dockerfile for the Raspberry Pi image is different to the one for the nomal image, because it load a Raspbian base image instead of an Ubuntu image like the normal file will do. Also the setup script is slightly different, because Raspbian sometimes need other dependencies or has different preinstalled tools. Listing 5.12 shows the Dockerfile for the normal Motey build.

```
1  FROM ubuntu:xenial
2
3  MAINTAINER Neoklosch version: 0.0.1
4
5  ADD ./setup.sh /tmp/setup.sh
6  ADD ./requirements.txt /tmp/requirements.txt
7  RUN /bin/bash /tmp/setup.sh
8
9  EXPOSE 5023 5091 5092 5094 5094 1883
10
11 CMD ["motey", "start"]
```

Listing 5.12: Dockerfile to create the Motey Docker image

The *FROM* command at line 1 defines the base image. In this case it is an *Ubuntu* image in version 16.04 also called Xenial Xerus, which has a long-time support and is a pretty stable version. The *ADD* command is used to add external files to the Docker container to be executed later or make them available in the image. The script adds a *setup.sh* file that is executed in line 7. This file uses the *requirements.txt* file, that is also added. Additional the ports in line 9 are exposed, which means they are provided to the Docker engine as used by the container. There is no automatic mapping between the host and the guest system ports, but they can be mapped much faster due to the *EXPOSE* command. Finally the *CMD* command will execute a specific shell command, in this case the *motey start* command.
This command is available after Motey was installed by the *setup.py* file in the root folder. To install Motey, only two build steps has to be executed, that are shown in listing 5.13.

```
1  python3 setup.py build
2  python3 setup.py install
```

Listing 5.13: Motey setup procedure

When the installation routine is done, the *motey* command becomes available globally. The usage of the command line tool was shown in 5.3. Beside the installation via the setup script, Motey can also be added as a Docker container named *neoklosch/motey*. This method is preferred to make the first steps with Motey, because it is more isolated from the host system, easier to use, to remove and to update.

Chapter 6

Evaluation

In this chapter the implementation of the plugins, based on the previously defined requirements and concepts, are evaluated. Afterwards a demonstration of the system will be shown.

6.1 Test Environment

Motey and the performance scripts in section 6.2 was tested on three different devices.

1. **Raspberry Pi 2 Model B Version 1.1** hereinafter called *NeoPi*, that has a ARM Cortex-A7 CPU with four cores each with 900 MHz and a 32-bit architecture. It has 1024 MB RAM and a 10/100-MBit-Ethernet port. The device is running a Raspbian 8 (jessie) with Linux Kernel version 4.9.
2. **Raspberry Pi 3 Model B Version 1.2** hereinafter called *BuffPi*, is the currently the newest Raspberry Pi on the market. It has a ARM Cortex-A53 CPU also with four cores, but each has 1200 MHz and 64-bit architecture. It also has 1024 MB RAM and a 10/100-MBit-Ethernet port. This devices is also runngin Raspbian 8 (jessie) with Linux Kernel version 4.9.
3. **Acer Aspire V5-573G** hereinafter called *Laptop*, with an Intel Core i7-4500U CPU with two cores each 1.80 GHz and a 64-bit architecture as well. It has 8 GB RAM and a 802.11n WiFi connection. It is running a Linux Mint 18.1 64-bit OS with a Linux Kernel version 4.4.

Both the *NeoPi* and the *BuffPi* are connected via ethernet to a router. The *Laptop* is connect via 802.11n WiFi to the router.

6.2 Performance Evaluation

6.3 Scalability

6.4 Code Verification

Code verification is a quite important technique in every development project. There are several possibilities to check and verify the integrity of the created source code. Two of them are used in this project. The style check is used to ensure the compliance with the code style guidelines and the unit tests are used to verify the correctness of the code and the outcome of the functions itself. As the guidelines the pretty famous PEP 8 style guide[1] is used. PEP 8 is used in the Python standard library code and is well established in the community. A standardized code style is recommended in team projects as well as in open source projects. But also in private project a commitment to a specific style can be helpful to make the project easier to maintain. The code in general becomes more readable, more understandable and the amount of errors can be decreased. Due to the fact that overall code is read more often than it is written, other people will be satisfied by having a common understanding of the "grammar" of the code to be used. As the tool of choice the library *pycodestyle* will be used. The style check is part of the CI pipeline and the concrete implementation was shown in listing 5.11. If there is any style check violation, the CI build process will fail and the new Docker image will not be uploaded.

[1] `https://www.python.org/dev/peps/pep-0008`

```
$ pycodestyle --ignore=E241,E501 motey/
motey/di/app_module.py:48:38: E128 continuation line under-indented for
↪ visual indent
motey/di/app_module.py:49:38: E128 continuation line under-indented for
↪ visual indent
motey/di/app_module.py:50:38: E128 continuation line under-indented for
↪ visual indent
motey/di/app_module.py:51:38: E128 continuation line under-indented for
↪ visual indent
motey/di/app_module.py:52:38: E128 continuation line under-indented for
↪ visual indent
motey/di/app_module.py:53:38: E128 continuation line under-indented for
↪ visual indent
motey/di/app_module.py:54:38: E128 continuation line under-indented for
↪ visual indent
motey/configuration/configreader.py:6:66: E703 statement ends with a
↪ semicolon

The command "pycodestyle --ignore=E241,E501 motey/" exited with 1.

$ pycodestyle --ignore=E241,E501 samples/

The command "pycodestyle --ignore=E241,E501 samples/" exited with 0.
```

Listing 6.1: Sample output of the style check validation from the Travis CI build process number 148[0]

Listing 6.1 shows an example output of the Travis CI build process. The first style check call fails due to multiple validations, but the second one exited successfully.

The second verification is are the unit tests. A unit test is as the name implies a software test method which tests each component of an application. Each component should be tested as isolated as possible by invoking only one or a couple of methods from a unit and the result should be verified automatically and compared with an expected result.[cf. 0, p. 320] All the used objects in a class should be as independent from each other as possible. To ensure this, the objects have to be mocked by a mocking framework. The python unit test framework has a build-in mocking framework since version 3. A mock is a fake objects that acts as a dummy for the class to be tested. To decouple the dependencies and to increase the testability and more specific to make used object easier mockable, the DI pattern is used. Each injected object can be easily mocked outside of the class. Due to the fact that Python allows to modify each class member at any time, DI is not really necessary, but it helps to make it clearer to understand and the code more readable and maintainable.

The advantages of unit tests in general are that problems can be easier localized and much faster detected. If an error occures during unit testing a former well working code, it is obvious that the last code changes are the reason for the error. This benefit is also helpful to readuce the fear of refactoring or extend code parts. Even pretty old parts of an application can be modified without the risk of any major problems. When the unit test are part of the CI, each

build will be checked. This reduces the risk of the deployment of faulty code. And finally unit tests can be a good starting point for new team members, because it helps to understand the functionality of a class. An unit test can acts as some kind of documentation. The downsides are that unit tests are coded. This means also unit tests can have errors. Furthermore poorly written unit tests or tests that a written unpleasured can end up in a wrong feeling of safeness. Some error case could not be catched and the submitted code is still faulty. Additionally the test setup is not realistic. Integration test are more accurate in terms of the human behavior. Also the dependencies between the component and the interaction between them can be tested much better with integration test. The most critical part of unit test in the economy is that the take time. For each written component, the realted tests take a significant time to be developed. In a real life project this could be unacceptable even if it is important to have them.

```
1   class TestServiceRepository(unittest.TestCase):
2     @classmethod
3     def setUp(self):
4         self.text_service_id = uuid.uuid4().hex
5         self.test_service = {'id': self.text_service_id, 'service_name':
    ↪ 'test service name', 'images': ['test image']}
6         service_repository.config = {'DATABASE': {'path':
    ↪ '/tmp/testpath'}}
7         service_repository.BaseRepository =
    ↪ mock.Mock(service_repository.BaseRepository)
8         service_repository.TinyDB = mock.Mock(TinyDB)
9         service_repository.Query = mock.Mock(Query)
10        self.test_service_repository =
    ↪ service_repository.ServiceRepository()
11
12    def test_has_entry(self):
13        self.test_service_repository.db.search =
    ↪ mock.MagicMock(return_value=[1, 2])
14
15        result =
    ↪ self.test_service_repository.has(service_id=self.test_service['id'])
16
17        self.assertTrue(self.test_service_repository.db.search.called)
18        self.assertTrue(result)
```

Listing 6.2: Extract from the Motey unit test of the ServiceRepository

describe the listing

Unit testing in general can be realized in two different ways. The first possibility is the by writing the tests after the code is done. finalize this paragraph

The second possibility is the Test Drivin Development (TDD), that is related to the test-first programming concept of the extreme programming development. In this development methodology the tests are written before the implementation of a class is created. This

helps to plan the architecture of a class and catch edge cases before the implementation is done. TDD is an iterative process where at first the test is written, then the test will be executed and must fail, afterwards the code is written and the tests should be executed again, but this time successfully. Finally the process can be repeated for example if the code has to be refactored or extended. Especially in an extrem programming environment TDD in combination with pair programming and code reviews are reasonable.

Motey was created with the "normal" unit testing approach, due to the fact that it was a one man project with rapidly changing requirements and an explorative approach to create the prototype. Nevertheless both methods ends up in a well tested code base and an assured code stability.

6.5 Conclusion

Chapter 7

Conclusion

7.1 Summary

7.2 Dissemination

7.3 Impact

7.4 Outlook

Acronyms

AE	Autoscaling Engine
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
BSS	Business Support Systems
CapEx	Capital Expenditure
CDN	Content Delivery Network
CI	Continuous Integration
CLI	Command Line Interface
CORS	Cross-Origin Resource Sharing
CPU	Central Processing Unit
CPS	Cyber-Physical System
CSRF	Cross-Site Request Forgery
CS	Cyber System
CSS	Cascading Style Sheets
DI	Dependency Injection
EPGM	Encapsulated Pragmatic General Multicast
EMS	Element Management System
ETSI	European Telecommunications Standards Institute
FM	Fault Management
FOKUS	Fraunhofer-Institut für Offene Kommunikationssysteme
GUI	Graphical User Interface
H2H	Human-to-Human
H2M	Human-to-Machine
HATEOAS	Hypermedia As The Engine Of Application State
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IERC	European Research Cluster on the Internet of Things
IoC	Inversion of Control
IoS	Internet of Services
IoT	Internet of Things
IP	Internet Protocol
IPC	Inter-Process Communication
IT	Information Technology
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LXC	Linux Containers
M2M	Machine-to-Machine
MANO	Management And Orchestration
MQTT	Message Queue Telemetry Transport
NF	Network Function
NFV	Network Function Virtualistion
NFVI	Network Function Virtualization Infrastructure
NFV-MANO	Network Function Virtualistion Management And Orchestration
NFVO	Network Function Virtualistion Orchestrator
NIST	National Institute of Standards and Technology

OASIS	Organization for the Advancement of Structured Information Standards
OpEx	Operating Expense
OS	Operating System
OSS	Operations Support Systems
PEP	Python Enhancement Proposal
PGM	Pragmatic General Multicast
PNF	Physical Network Function
PoP	Point of Presence
QoS	Quality of Service
RAM	Random Access Memory
REST	Representational State Transfer
RFID	Radio Frequency Identification
SDK	Software Development Kit
SDN	Software Defined Networking
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TDD	Test Drivin Development
TLS	Transport Layer Security
TOSCA	Topology and Orchestration Specification for Cloud Applications
UI	User Interface
URL	Uniform Resource Locator
VAL	Virtualization Abstraction Layer
VIM	Virtual Infrastructure Manager
VM	Virtual Machine
VMM	Virtual Machine Monitor
VNF	Virtual Network Function
VNFM	Virtual Network Function Manager
YAML	YAML Ainť Markup Language

Glossary

Algorithmus a
Chiffrierung a
Dechiffrierung a

Bibliography

[0] A. Bosche, D. Crawford, D. Jackson, M. Schallehn, and P. Smith. *How Providers Can Succeed in the Internet of Things*. Accessed: 2017-02-20. Aug. 2016. URL: http://bain.com/publications/articles/how-providers-can-succeed-in-the-internet-of-things.aspx (cit. on p. 1).

[0] E. A. Lee. "Cyber Physical Systems: Design Challenges". In: *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. May 2008, pp. 363–369. DOI: 10.1109/ISORC.2008.25 (cit. on p. 1).

[0] R. Poovendran. "Cyber Physical Systems: Close Encounters Between Two Parallel Worlds". In: *Proceedings of the IEEE* 98.8 (Aug. 2010), pp. 1363–1366. ISSN: 0018-9219. DOI: 10.1109/JPROC.2010.2050377 (cit. on pp. 1, 8).

[0] C. Pahl and B. Lee. "Containers and Clusters for Edge Cloud Architectures – A Technology Review". In: *2015 3rd International Conference on Future Internet of Things and Cloud*. Aug. 2015, pp. 379–386. DOI: 10.1109/FiCloud.2015.35 (cit. on pp. 1, 10, 13).

[0] M. S. D. Brito, S. Hoque, R. Steinke, and A. Willner. "Towards Programmable Fog Nodes in Smart Factories". In: *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W)*. Sept. 2016, pp. 236–241. DOI: 10.1109/FAS-W.2016.57 (cit. on pp. 1, 6, 8).

[0] M. Yannuzzi, R. Milito, R. Serral-Gracià, D. Montero, and M. Nemirovsky. "Key ingredients in an IoT recipe: Fog Computing, Cloud computing, and more Fog Computing". In: *2014 IEEE 19th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*. Dec. 2014, pp. 325–329. DOI: 10.1109/CAMAD.2014.7033259 (cit. on pp. 2, 8).

[0] C. Pahl, S. Helmer, L. Miori, J. Sanin, and B. Lee. "A Container-Based Edge Cloud PaaS Architecture Based on Raspberry Pi Clusters". In: *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*. Aug. 2016, pp. 117–124. DOI: 10.1109/W-FiCloud.2016.36 (cit. on p. 2).

[0] D. Evans. *The Internet of Things: How the Next Evolution of the Internet Is Changing Everything*. Tech. rep. Accessed: 2017-02-12. Cisco Internet Business Solutions Group (IBSG), Apr. 2011. URL: http://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf (cit. on p. 5).

[0] J. Rui and S. Danpeng. "Architecture Design of the Internet of Things Based on Cloud Computing". In: *2015 Seventh International Conference on Measuring Technology and Mechatronics Automation*. June 2015, pp. 206–209. DOI: 10.1109/ICMTMA.2015.57 (cit. on pp. 5, 6).

[0] T. Kramp, R. van Kranenburg, and S. Lange. "Introduction to the Internet of Things". In: *Enabling Things to Talk*. Vol. 1. Springer-Verlag Berlin Heidelberg, 2013, pp. 1–10. ISBN: 978-3-642-40402-3. DOI: 10.1007/978-3-642-40403-0 (cit. on pp. 5, 6).

[0] *ITU Internet Reports: The Internet of Things*. Tech. rep. Accessed: 2017-02-12. International Telecommunication Union, Nov. 2005. URL: https://www.itu.int/net/wsis/tunis/newsroom/stats/The-Internet-of-Things-2005.pdf (cit. on p. 5).

[0] M. Weiser. *The Computer for the 21st Century*. Accessed: 2017-02-12. Sept. 1991. URL: http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html (cit. on p. 6).

[0] M. Lom, O. Pribyl, and M. Svitek. "Industry 4.0 as a part of smart cities". In: *2016 Smart Cities Symposium Prague (SCSP)*. May 2016, pp. 1–6. DOI: 10.1109/SCSP.2016.7501015 (cit. on pp. 6, 8).

[0] M. Hermann, T. Pentek, and B. Otto. *Design Principles for Industrie 4.0 Scenarios: A Literature Review*. Tech. rep. Accessed: 2017-02-12. Technische Universität Dortmund - Fakultät Maschinenbau, Jan. 2015. URL: http://www.thiagobranquinho.com/wp-content/uploads/2016/11/Design-Principles-for-Industrie-4_0-Scenarios.pdf (cit. on pp. 6, 7).

[0] B. Lydon. "Industry 4.0: Intelligent and flexible production". In: *InTech Magazine* (May 2016). Accessed: 2017-02-13. URL: https://www.isa.org/intech/20160601/ (cit. on pp. 6, 7).

[0] *Dienstleistungspotenziale im Rahmen von Industrie 4.0*. Tech. rep. Accessed: 2017-02-12. vbw Vereinigung der Bayerischen Wirtschaft e. V., Mar. 2014. URL: http://www.forschungsnetzwerk.at/downloadpub/dienstleistungspotenziale-industrie-4.0-mar-2014.pdf (cit. on p. 7).

[0] O. Jurevicius. *Vertical Integration*. Accessed: 2017-02-13. Apr. 2013. URL: https://www.strategicmanagementinsight.com/topics/vertical-integration.html (cit. on p. 7).

[0] K. Scarfone, M. Souppaya, and P. Hoffman. *Guide to Security for Full Virtualization Technologies*. Tech. rep. Accessed: 2017-02-19. National Institute of Standards and Technology, Jan. 2011. URL: http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-125.pdf (cit. on p. 9).

[0] A. Celesti, D. Mulfari, M. Fazio, M. Villari, and A. Puliafito. "Exploring Container Virtualization in IoT Clouds". In: *2016 IEEE International Conference on Smart Computing (SMARTCOMP)*. May 2016, pp. 1–6. DOI: 10.1109/SMARTCOMP.2016.7501691 (cit. on pp. 9, 10).

[0] S. Gallagher. *Mastering Docker*. Packt Publishing Ltd., Dez 2015. ISBN: 978-1-78528-703-9 (cit. on pp. 10, 15).

[0] A. Tosatto, P. Ruiu, and A. Attanasio. "Container-Based Orchestration in Cloud: State of the Art and Challenges". In: *2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems*. July 2015, pp. 70–75. DOI: 10.1109/CISIS.2015.35 (cit. on pp. 10, 13).

[0] J. Anderson, H. Hu, U. Agarwal, C. Lowery, H. Li, and A. Apon. "Performance considerations of network functions virtualization using containers". In: *2016 International Conference on Computing, Networking and Communications (ICNC)*. Feb. 2016, pp. 1–7. DOI: 10.1109/ICCNC.2016.7440668 (cit. on p. 10).

[0] *Network Function Virtualistion (NFV); Architectural Framework*. Tech. rep. Accessed: 2017-06-24. ETSI - European Telecommunications Standards Institute, Oct. 2013. URL: http://www.etsi.org/deliver/etsi_gs/nfv/001_099/002/01.01.01_60/gs_nfv002v010101p.pdf (cit. on p. 11).

[0] L. Rivenes. *What is Network Function Virtualization (NFV)?* Accessed: 2017-03-03. Sept. 2014. URL: https://datapath.io/resources/blog/network-function-virtualization-nfv (cit. on p. 11).

[0] S. Noble. *Network Function Virtualization or NFV Explained*. Accessed: 2017-03-03. Apr. 2015. URL: http://wikibon.com/network-function-virtualization-or-nfv-explained (cit. on pp. 11, 12).
[0] *NFV*. Accessed: 2017-03-19. URL: https://sdn-wiki.fokus.fraunhofer.de/doku.php?id=nfv (cit. on p. 11).
[0] F. Kahn. *Kubernetes User Case Studies*. Accessed: 2017-03-19. Mar. 2015. URL: http://www.telecomlighthouse.com/a-cheat-sheet-for-understanding-nfv-architecture (cit. on pp. 12, 13).
[0] *Why is TOSCA Relevant to NFV? Explanation*. Accessed: 2017-03-19. URL: https://www.sdxcentral.com/nfv/definitions/tosca-nfv-explanation (cit. on pp. 12, 13).
[0] *Understand images, containers, and storage drivers - Docker*. https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/. Accessed: 2017-02-24 (cit. on p. 14).
[0] *Docker Overview - Docker*. https://docs.docker.com/engine/understanding-docker/. Accessed: 2017-02-24 (cit. on p. 14).
[0] B. Grant. *Kubernetes: a platform for automating deployment, scaling, and operations*. Accessed: 2017-02-27. Nov. 2015. URL: https://www.slideshare.net/BrianGrant11/wso2con-us-2015-kubernetes-a-platform-for-automating-deployment-scaling-and-operations (cit. on p. 15).
[0] J. MSV. *Kubernetes architecture*. Accessed: 2017-03-03. Oct. 2016. URL: https://www.slideshare.net/janakiramm/kubernetes-architecture (cit. on pp. 15, 16).
[0] E. Mulyana. *Kubernetes Basics*. Accessed: 2017-03-03. May 2016. URL: https://www.slideshare.net/e2m/kubernetes-basics (cit. on pp. 15, 16).
[0] *Pods - Kubernetes*. Accessed: 2017-03-03. Dec. 2016. URL: https://kubernetes.io/docs/user-guide/pods (cit. on p. 15).
[0] *Labels and Selectors - Kubernetes*. Accessed: 2017-03-03. Dec. 2016. URL: https://kubernetes.io/docs/user-guide/labels (cit. on p. 16).
[0] *kube-proxy - Kubernetes*. Accessed: 2017-03-03. Dec. 2016. URL: https://kubernetes.io/docs/admin/kube-proxy (cit. on p. 16).
[0] *Replication Controller - Kubernetes*. Accessed: 2017-03-03. Dec. 2016. URL: https://kubernetes.io/docs/user-guide/replication-controller (cit. on p. 16).
[0] *Rolling Updates - Kubernetes*. Accessed: 2017-03-03. Dec. 2016. URL: https://kubernetes.io/docs/user-guide/rolling-updates (cit. on p. 16).
[0] *Docker Swarm Documentation*. https://docs.docker.com/engine/swarm. Accessed: 2017-03-18 (cit. on p. 16).
[0] *OpenBaton Documentation*. http://openbaton.github.io/documentation. Accessed: 2017-03-18 (cit. on pp. 16–18).
[0] *FAQ - Frequently Asked Questions | MQTT*. Accessed: 2017-06-03. URL: http://mqtt.org/faq (cit. on p. 19).
[0] V. Lampkin, W. Leong, L. Olivera, S. Rawat, N. Subrahmanyam, R. Xiang, G. Kallas, N. Krishna, S. Fassmann, M. Keen, et al. *Building Smarter Planet Solutions with MQTT and IBM WebSphere MQ Telemetry*. IBM redbooks. IBM Redbooks, 2012. ISBN: 9780738437088. URL: https://books.google.de/books?id=F_HHAgAAQBAJ (cit. on p. 19).
[0] T. Bayer. *MQTT, Einführung in das Protokoll für M2M und IoT*. Accessed: 2017-06-03. Mar. 2016. URL: https://www.predic8.de/mqtt.htm (cit. on pp. 19, 20).
[0] *ØMQ - The Guide*. Accessed: 2017-06-06. URL: http://zguide.zeromq.org/page:all (cit. on pp. 20–23).

[0] *zmq_inproc(7) - 0MQ Api.* Accessed: 2017-06-06. URL: http://api.zeromq.org/3-2:zmq-inproc (cit. on p. 21).
[0] *zmq_tcp(7) - 0MQ Api.* Accessed: 2017-06-06. URL: http://api.zeromq.org/3-2:zmq-tcp (cit. on p. 21).
[0] *zmq_pgm(7) - 0MQ Api.* Accessed: 2017-06-06. URL: http://api.zeromq.org/3-2:zmq-pgm (cit. on p. 21).
[0] *OpenFog Reference Architecture for Fog Computing.* Tech. rep. Accessed: 2017-06-24. OpenFog Consortium Architecture Working Group, Feb. 2017. URL: https://www.openfogconsortium.org/wp-content/uploads/OpenFog_Reference_Architecture_2_09_17-FINAL.pdf (cit. on p. 25).
[0] *Network Functions Virtualisation (NFV); Management and Orchestration.* Tech. rep. Accessed: 2017-06-24. ETSI - European Telecommunications Standards Institute, Dec. 2014. URL: http://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_NFV-MAN001v010101p.pdf (cit. on p. 25).
[0] T. K. Authors. *A Cheat Sheet for Understanding "NFV Architecture".* Accessed: 2017-04-10. URL: https://kubernetes.io/case-studies (cit. on p. 28).
[0] *Cloud Platform for IoT: Designing and Evaluating Large-Scale Data Collecting and Storing Platform | OpenStack Summit Videos.* Accessed: 2017-06-25. URL: https://www.openstack.org/videos/video/cloud-platform-for-iot-designing-and-evaluating-large-scale-data-collecting-and-storing-platform (cit. on p. 29).
[0] *OpenStack and Kubernetes join forces for an Internet of Things platform.* Accessed: 2017-06-25. URL: http://superuser.openstack.org/articles/openstack-and-kubernetes-join-forces-for-an-internet-of-things-platform/ (cit. on p. 29).
[0] G. Technologies. *Orchestration-First, Model-Driven NFV Cloud Management.* Accessed: 2017-04-14. URL: http://getcloudify.org/network-function-virtualization-vnf-nfv-orchestration-sdn-platform.html (cit. on p. 29).
[0] *Most Used Programming Languages 2017: The Trendiest & Most Sought After Coding Languages.* Accessed: 2017-06-16. URL: https://stackify.com/trendiest-programming-languages-hottest-sought-programming-languages-2017 (cit. on pp. 30, 31).
[0] *Python Garbage Collection - Digi Developer.* Accessed: 2017-06-17. URL: https://www.digi.com/wiki/developer/index.php/Python_Garbage_Collection (cit. on p. 31).
[0] B. Peterson. *Python 2.7 Release Schedule.* Accessed: 2017-06-17. June 2016. URL: http://legacy.python.org/dev/peps/pep-0373 (cit. on p. 31).
[0] *Documentation - The Go Programming Language.* Accessed: 2017-06-16. URL: https://golang.org/doc (cit. on p. 31).
[0] *Eclipse Mosquitto.* Accessed: 2017-06-15. URL: https://projects.eclipse.org/projects/technology.mosquitto (cit. on p. 34).
[0] *Foreword - Flask Documentation.* Accessed: 2017-06-18. URL: http://flask.pocoo.org/docs/0.12/foreword (cit. on p. 43).
[0] *Steve Cohen's answer to What challenges has Pinterest encountered with Flask? - Quora.* Accessed: 2017-06-18. URL: https://www.quora.com/What-challenges-has-Pinterest-encountered-with-Flask/answer/Steve-Cohen (cit. on p. 43).
[0] *Introducing Flask-RESTful.* Accessed: 2017-06-18. URL: https://www.twilio.com/engineering/2012/10/18/open-sourcing-flask-restful (cit. on p. 43).
[0] *Travis CI Motey build 148.* Accessed: 2017-07-03. URL: https://travis-ci.org/Neoklosch/Motey/builds/242813268 (cit. on pp. ix, 62).

[0] M. Olan. “Unit Testing: Test Early, Test Often”. In: *J. Comput. Sci. Coll.* 19.2 (Dec. 2003), pp. 319–328. ISSN: 1937-4771. URL: http://dl.acm.org/citation.cfm?id=948785.948830 (cit. on p. 62).