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 Santos de Brito

"Obstacles don’t have to stop you.
If you run into a wall, don’t turn around and give up.
Figure out how to climb it, go through it, or work around it."

- Michael Jordan -

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 31, 2017

____
Markus Paeschke

Acknowledgments

Firstly, I would like to express my sincere gratitude supervisor Mathias Santos de Brito for the continuous support and accompaniment during my study and related research. His guidance helped me in all the time of research and writing of this thesis.

I would also like to acknowledge Dr.-Ing. Alexander Willner as the second reader of this thesis, for his insightful comments and encouragement.

I would also like to thank the FOKUS, which has made it possible for me to start this thesis.

I would like to thank my parents, who have supported me my whole life in everything I've ever did. I will be grateful forever for your love.

A special thank go out to Nora Mudrack, Daniel Ghioreanu and Andreas Buff, who helped me not only in the Master thesis, but also through my entire studies and have given me the unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Finally, I would like to thank Christian Sommer and Birgit Mudrack for the proofreading and the helpful advice in the preparation of this work.

Berlin, July 31, 2017

Abstract

The Internet of Things (IoT) is one of the biggest topics in the recent years. In order to limit the vast area of IoT, more and more standards are defined and subtopics established. The Industry 4.0, that includes smart factories, is one of them, with the goal to improve the value chain in a factory, enable process automation and adding intelligent connections between different companies and units. The fundamental architecture that is necessary in this context is a highly distributed architecture, that can have thousands of nodes with multiple sensors, machines or smart components connected to them.

The concept of Fog Computing brings the cloud and the related functionalities as close to the underlying network as possible, to create smaller independent fog clouds. Virtual Machines are a common approach in cloud systems, but due to the fact that in an IoT context mostly low power devices are used, these technology is not feasible in any case. In addition, in some scenarios these devices have to interact in a highly heterogeneous and hybrid environment. Lossy signals and short range radio technologies are widely used and nodes can appear and disappear frequently. The latency has to be kept low to enable real-time applications, the traffic and resource overhead must be as small as possible and for security and data privacy reasons, sensitive data should be kept on-premise.

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 prototypical implementation of an orchestration engine for a fog node, called Motey, that can deploy Docker containers to the same node or on any other fog node in the cluster. 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 dependency to another node.

One of the biggest challenge in this project was to create a fast and lightweight connection between the nodes, that was achieved by the ZeroMQ protocol. The node discovery, that enables each node to have knowledge of all the others, is realized with the MQTT protocol and is used to establish the mentioned inter node connection. The abstraction of the underlying virtualization tools partly kept unsolved, because only a few virtualization tools supports the ARM CPU architecture.

Motey can be seen as a good starting point for a complex environment made for Fog Computing. It has also space for some improvements, for example the autonomous behavior can be improved a lot. One possibility could be the ability to respond to real-time requirements even in case of the absence of the centralized cloud level. At the end the created project successfully reveal that the developed concept works out pretty well in a prototypical quality. Motey is a well developed, tested and documented and can be considered as a pretty solid basis for further development.

Zusammenfassung

Das Internet der Dinge (IoT) ist eins der bedeutendsten Themen der letzten Jahre. Um die enorme Bandbreite im Bereich IoT zu begrenzen wurden mehrere Standards und Unterkategorien eingeführt, z.b. die Industrie 4.0, welche Smart Factories beinhaltet. Sie hat zum Ziel, die Produktionskette in Fabriken zu optimieren, Prozessautomatisierung zu schaffen und intelligente Verbindungen zwischen verschiedenen Abteilungen und Firmen zu schaffen. Um diese Ziele zu erreichen ist eine hochgradig verteilte Architektur notwendig, welche tausende von Nodes, die mit Sensoren, Maschinen oder smarten Objekten verbunden sein kann, umfasst.

Das Konzept des Fog Computing bringt dabei die Cloud und die dazugehörigen Funktionalitäten näher an die darunterliegende Netzwerkschicht, um kleinere unabhängige Fog Clouds zu schaffen. Virtuelle Maschinen sind ein gängige Methode in Cloud System, jedoch nur bedingt im IoT Kontext anwendbar, da hier meist leistungsschwache Geräte verwendet werden, die diese nur bedingt ausführen können. Zudem sind die Geräte oftmals in einem heterogenen und hybriden Umfeld im Einsatz. Verlustbehaftete Signale und Kurzstreckentechnologien sind häufig im Einsatz und Nodes können hochfrequent erscheinen und verschwinden. Die Latenz muss klein gehalten werden, um Echtzeitanwendungen zu erstellen, der Datenverkehr und die Ressourcennutzung müssen so klein wie möglich gehalten werde und aus Sicherheits- und Datenschutzgründen sollten sensitive Daten vor Ort gehalten werden.

Diese Abschlussarbeit zeigt den Entwurf und die Umsetzung einer Fog Service orchestration Engine für Smart Factories auf. Das Ziel dieser Arbeit ist die prototypische Umsetzung einer orchestration Engine, genannt Motey, zu erstellen, die auf einer Fog Node ausgeführt wird, um Docker Container in einem Node Cluster zu orchestrieren. Weiterhin soll der Prototyp funktionale und nicht funktionale Abhängigkeiten beim aufspielen der Container beachten. Das können Hardwareanforderungen, eine benötigte Software oder Abhängigkeiten zu anderen Containern sein. Eine der größten Herausforderungen in diesem Projekt war es eine schnelle und leichtgewichtige Verbindungen zwischen den Nodes, welche mittels des ZeroMQ Protokolls umgesetzt wurde, zu erstellen. Die Node Discovery, welche es ermöglicht das jede Node Kenntnis über all die anderen hat, wurde mittels MQTT Protokoll umgesetzt und dient als Voraussetzung für die Inter-Node Kommunikation. Die Abstraktion der zugrundeliegenden Virtualisierungsoftware blieb teilweise ungelöst, da nur wenige Virtualisierungstools die ARM CPU Architektur der verwendeten Testgeräte unterstützen.

Motey kann jedoch als ein guter Ausgangspunkt für eine komplexe Fog Computing Umgebung gesehen werden. Es bietet Raum für Erweiterungen, z.B. kann ein erweitertes autonomes Verhalten implementiert werden. Dieses könnte bspw. die Fähigkeit zum reagieren auf Echtzeitanforderungen, selbst bei fehlenden Cloudebene, beinhalten. Zusammenfassend zeigt das erstellte Projekt erfolgreich, dass das erarbeitete Konzept in einer prototypischen Qualität funktioniert. Motey ist gut umgesetzt, getestet und dokumentiert und kann als solid Grundlage für eine Weiterentwicklung betrachtet werden.

Table of Contents

List of Figures

List of Tables

List of Listings

Acronyms

AE	Autoscaling Engine
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
BSS	Business Support Systems
CapEx	Capital Expenditure
CC	Constant Connection
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
GaaS	Gateway-as-a-Service
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
KVM	Kernel-based Virtual Machine
LXC	Linux Containers

M2M	Machine-to-Machine
MANO	Management And Orchestration
MLU	Machine Learning Unit
MQTT	Message Queue Telemetry Transport
NC	New Connection
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
PaaS	Platform as a Service
PC	Personal Computer
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
UDP	User Datagram Protocol
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 Ain’t Markup Language

CHAPTER 1

Introduction

1.1 Motivation

The Internet of Things (IoT) is one of the biggest topics in the recent years. Companies with a focus on that subject 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 selling hardware, software and comprehensive solutions in the IoT context by 2020.[1] 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.[cf. 2, p. 7] 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 interconnected to a distributed and autonomous value chain. Also, the automation level in a single factory will be increased which helps to have a more flexible and efficient production process. Currently a modern 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. 3, p. 363] A normal Cyber System (CS) is passive, means it can 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. 4, p. 1363 f.]

Another solution is to change the fundamental architecture of such a system from a monolithic to more distributed architecture. With Fog Computing the cloud moves away from centralized data centers to the edge issues in that context c of the underlying network.[cf. 5, 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"[6, p.236] enables a lot of new possibilities like pre-computation and storage of gathered data. This reduces traffic and the resource overhead in the cloud, it keeps sensitive data on-premise[cf. 6, p.236] and enables real-time applications to take decisions based on analytics running near the device. It also can achieve to lower the network 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. 7, 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 changed often and have to be deployed and removed in a dynamical way.

Virtual Machines (VMs) are a common approach in Cloud systems to provide elasticity of large-scale shared resources.[cf. 8, 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"[8, 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 these technologies can be applied to the subject of IoT, some of the main issues in that context can be potentially solved. For example, lossy signals can be compensated and traffic can be reduces by having services dynamically deployed to a cluster of nodes or an intermediate layer. The challenge is to deploy these services, connect them and let them interact with each other, even without the presence of a cloud layer. An autonomous behavior could be enabled if the nodes are able to communicate directly.

This thesis demonstrate the possibilities of container orchestration for the IoT and smart factories by creating a prototypical implementation of an orchestration engine that can be executed on fog nodes. Therefore, an engine will be created that can orchestrate containers based on functional and non-functional constraints onto 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 prototypical implementation of an orchestration engine for a fog node, hereinafter called prototype, that can deploy containers on the same node or on other network nodes. A fog node can be a low power device at the edge of a network, especially in the context 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 dependency to another node. The technical prerequisite and detailed requirements for

[1] `https://kubernetes.io`

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

the prototype, as well as the usability of a Graphical User Interface (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 of their knowledge and experience in the area of IoT and NFV. As the development method of choice a Kanban like process will be used. Kanban is flexible but also straight forward. There is less meeting overhead than in Scrum, but it also helps to have an eye on the planed and spend resources.

1.3 Scope

As mentioned before, the engine to be developed will be a orchestration engine prototype for a fog node. Open Baton serves as the template for that project and especially the modular architecture and the resulting extensibility will be targeted. Besides 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 an Application Programming Interface (API), where third party apps can communicate with and can control the engine itself. The functional and non-functional constraints can be stored and transferred for example as YAML Ain't Markup Language (YAML) schemes, that are part of the Topology and Orchestration Specification for Cloud Applications (TOSCA) standard. These schemes should be extendable and should fit the needs of the constraint functionality.

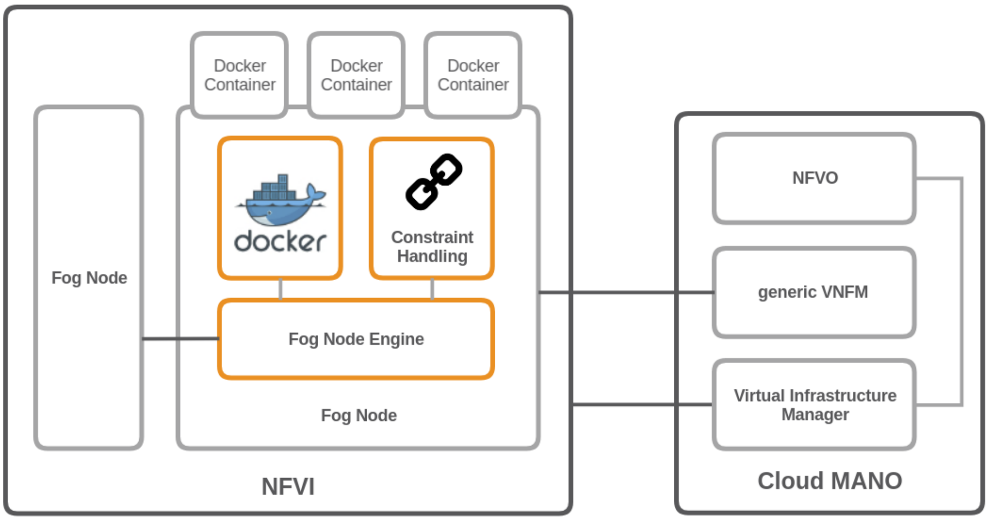


Figure 1: Conceptual architecture based on a draft of the FOKUS

Figure 1 shows a conceptual architecture design that is based on a draft of the FOKUS. On the right side is the abstract cloud infrastructure. This could be for example Open Baton or any other MANO compliant Framework. The cloud infrastructure will be considered while developing the prototype, but the development or integration of such a component is out of scope for this thesis. The Network Function Virtualization Infrastructure (NFVI) on the left

[3] `https://www.docker.com`

side includes the Fog Node. A single fog node should have the fog node engine, which is the prototype to be developed, as well as the Docker engine. The constraint handling is part of the fog node engine and the concrete implementation has to be elaborated. The fog node engine can orchestrate the local Docker containers, or if it necessary, it can move containers over 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 that guarantees to be executable on arbitrary devices, but theoretically this could be achieved by using a platform independent programming language and corresponding frameworks.

1.4 Outline

Chapter 2 is the technical foundation for the prototype to be developer. An introduction is given to relevant topics, like the IoT in the context of Industry 4.0, CPSs and Fog Computing as subtopics, virtualization, especially container virtualization and orchestration and NFV. This is followed by a comparison of several established tools available on the market, that can be used to speed up the development process. Some lightweight messaging concepts, like Message Queue Telemetry Transport (MQTT) and ZeroMQ, that probably better fit into an IoT application, than the common Hypertext Transfer Protocol (HTTP), are shown. The section will be finalized by an overview of realted work. 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 gathered insights as well as an outlook for further improvements of the plugin will be argued in chapter 7.

[4] `https://www.raspberrypi.org`

CHAPTER 2

State of the art

This chapter will give an overview into the technical concepts utilized in this thesis. In the first section the IoT and related subtopics like smart fctories and smart cities are considered. CPSs are important for the development of smart factories and are also covered in this section. Virtualization in general is the main topic of the section 2.2. First VMs are highlighted, followed by Container Virtualization. Both are related to each other and share some basic ideas. Container Orchestration shows exemplary possibilities of Container Virtualization. The last subsection NFV concludes with an introduction of the virtualization of network node functions to create communication services. Afterwards a brief overview of some major existing tools, as well as an excursion to two important messaging systems, will be given. The chapter will be finished with showing up the current state of research in the research area.

2.1 Internet of Things

IoT has grown strongly in the media and economy in the recent years. In the year 2008 the number of devices, that were connected to the Internet was higher than the human population.[cf. 9, p. 3] Cisco Internet Business Solutions Group predicted that the number will grow up to 50 billion in 2020, which equates to around 6 devices per person.[cf. 9, p. 4] Most of today's interactions are Human-to-Human (H2H) or Human-to-Machine (H2M) communication. The IoT on the other hand aims for the Machine-to-Machine (M2M) communication. This allows every physical device 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. 10, p.206][cf. 11, p.2] When objects are able to capture and monitor their environment, a network can perceive external stimuli and respond to them.[cf. 12, p. 40] Therefore, a new dimension of information and communication technology will be created, where users have access to everything at any time, everywhere. This omnipresent information processing in everyday life is also known as "Ubiquitous Computing", which was first mentioned in "The Computer for the 21st Century"[13] by Mark Weiser. 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.

At the beginning, the IoT started with the appearance of bar codes and Radio Frequency Identification (RFID) chips.[cf. 11, p. 13] In the second step, that is more or less the state of the art, sensors, physical devices, technical devices, data and software are connected to each other.[cf. 11, p. 13] This was in particular achieved cloud computing, which provides the highly efficient memory and computing power that is indispensable for such networks.[cf. 10, 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. 11, p. V] Just as the Ubiquitous Computing, it will take some time until it is omnipresent.

2.1.1 Industry 4.0 and smart factories

The industry we know is changing and 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 were 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. 14, p. 1] In the recent years the size, cost and power consumption of chipsets were reduced, which made it possible to embed sensors into devices and machines much easier and cheaper.[cf. 6, 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. 14, p. 1] "Industrie 4.0 is a collective term for technologies and concepts of value chain organization."[cf. 15, 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.[16, cf.] It also aims for an intelligent connection between different companies and units. Autonomous production and logistics processes create a real-time lean manufacturing ecosystem that is more efficient and flexible.[16, 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."[16] At the end, the system can use customer requirements in every step in the process to be flexible and responsive.[16, 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. 15, p. 11]

Table 1 shows the six design principles of the Industry 4.0 components. They can help companies to identify and implement Industry 4.0 scenarios.[cf. 15, p. 11]

1. *Interoperability* CPS of various manufacturers are connected with each other. Standards will be the key success factor in this subject.[cf. 15, 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. 15, 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. 15, 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. 15, 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. 15, 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. 15, p. 11]

Another important aspect of Industry 4.0 is the implementation of process automation with focus on three distinct aspects. Vertical integration, that contains the connection and communication of subsystems within the factory, enables flexible and adaptable manufacturing systems.[cf. 17, p. 7 ff.] 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. 17, p. 7 ff.] Finally end-to-end engineering, planning, and process control for each step in the production process.[16, cf.]

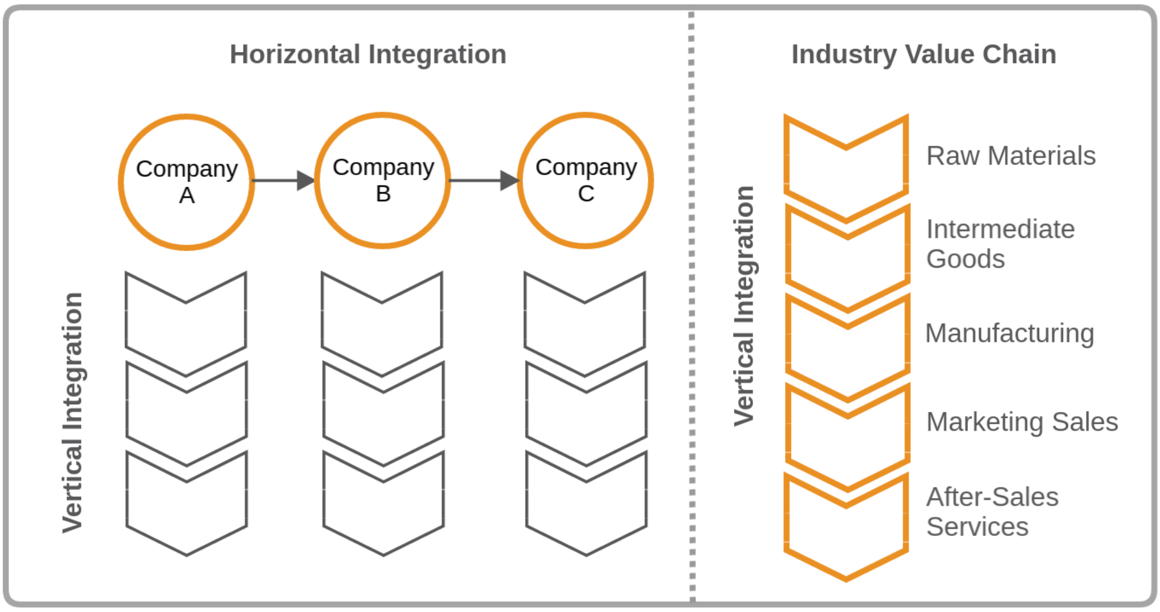


Figure 2: Horizontal vs. Vertical Integration. Adapted from: [18]

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 a more specific example of the vertical integration. From a technical perspective this means each machine in a factory has exactly to know what to do. The underlying system has to be modular and has to move away from a monolithic centralized system to a decentralized system, that is located locally near the machines itself.[cf. 5, p. 380] The communication path between them has to grow shorter. The machines have to be self organized and should communicate between each other, even if the core system is not available, because of lossy signals or other connection issues. This can enable flexibility, a better fault tolerance and individualized mass production.[16, cf.]

2.1.2 Cyber Physical Systems

As we already now, in smart factories every physical device is connected with 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. 4, p. 1363] Previously, a CS was passive, which means there was no communication between the physical and the virtual world.[cf. 4, p. 1364] While new technologies in the physical world, like new materials, hardware and energy supply, are developed, the technologies in the virtual world are also being improved, for example through the use of new protocols, networking, storage and computing technologies.[cf. 4, p. 1364] This adds more intelligence in such systems, as well as a more flexible and modular structure. A CPS can organize production automatically and autonomously, which eliminates the need of having a central process control.[14, cf.] Thereby the system can handle lossy signals and short range radio technologies, which are widely used in such a context.[7, 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 are based on a monolithic architecture. Over time the system was broken down to a more distributed 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. The main goals are to reduce the traffic and the amount of data that 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. 6, p. 236][cf. 7, p. 325][cf. 14, p. 4] In contrast to these goals, the definition and understanding of Fog Computing differs. One perspective is, that the processing of the data take place on smart devices, e.g. embedded systems, etc., at the end of the network or in smart router or other gateway devices.[cf. 14, p. 4] Another interpretation is that Fog Computing appears as an intermediate layer between smart devices and the cloud.[cf. 6, 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. 7, 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)."[19, p. ES-1]. This means a VM, also referred as guest system, can be executed on a real system, that 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. 20, 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. 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 illustrates both concepts. The *Bare-Metal Virtualization* on the left side, that is numbered among the system virtualization, use a *Hypervisor* that is executed between the soft- and hardware layer.[cf. 21, p. 1771] A VM will be directley executed from the *Hypervisor*. Each VM, in turn, has an OS, necessary libraries and the application inside them. All of the VMs are completely isolated form each other.[cf. 22, p.275] Some examples for system virtualization are VMWare[1], Oracle Virtual Box[2], XEN[3] or Microsoft Hyper-V[4].

Container Virtualization, that is one method of the process virtualization, is shown on the right side of figure 3. It use a *Container Engine*, that is executed on top of a *Host OS*. The

[1] `http://www.vmware.com`

[2] `https://www.virtualbox.org`

[3] `https://www.xenproject.org`

[4] `https://www.microsoft.com/de-de/cloud-platform/server-virtualization`

Container Engine itself manages the containers and isolates them.[cf. 20, p. 1] Depending on the engine, a container can have the libraries and application inside them or only the application. Libraries can be containerized and shared by the engine. With *Container Virtualization*, there is no need to start a *Guest OS*, which makes it more lightweight and less resource consuming compared to system virtualization.[cf. 20, p. 2 ff.][cf. 23, p. 1] Examples for process virtualization could be the Java Virtual Machine (JVM)[5], the .Net framework[6] or Docker[7].

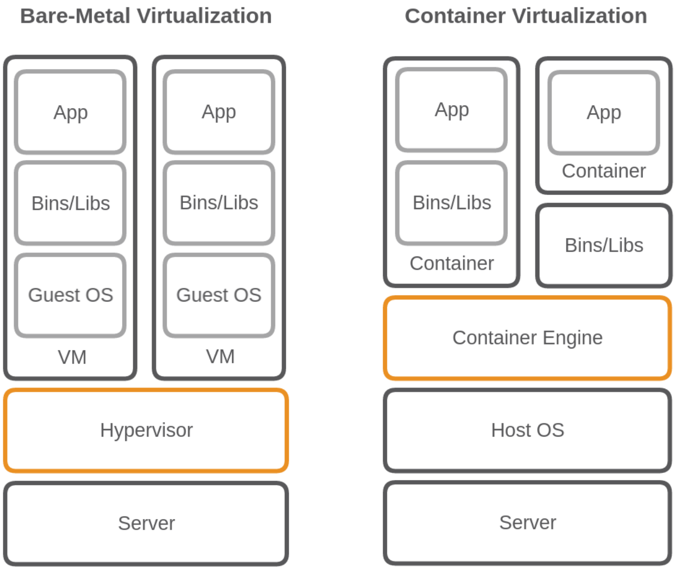


Figure 3: Structure bare-metal virtualization vs. container virtualization. Adapted from: [24, p. 2]

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.[cf. 21, p. 1771] Besides that, no human interaction is necessary during the provisioning process.[cf. 21, p. 1771] Elasticity, which scales a system in a cost-efficient manner in both directions, up and down, is also enabled. Customers, as well as the provider, profit from such a system. Security, based on the isolation of the VMs, is another 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.[cf. 23, p. 1] Running VMs increases the overhead and reduces the overall performance of a system.[cf. 23, p. 1] Therefore the specific use case have to consider that behavior.

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

[5] https://www.java.com
[6] https://www.microsoft.com/net
[7] https://www.docker.com

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 runs 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, that are necessary for the application to be executed.[cf. 5, 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 is based on fast and lightweight process virtualization to encapsulate an entire application with its dependencies on a ready-to-deploy virtual container.[cf. 25, p. 72] Such a container can be executed on the host OS, that allows an application to run as a sand-boxed user-space instance.[cf. 26, p. 1] All containers share a single OS kernel, so the isolation supposed to be weaker compared to hypervisor based virtualization.[cf. 20, 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. 20, p. 2]

2.2.3 Container Orchestration

Containers by itself help to develop and deploy applications, but containers release their full potential when they are used together with an orchestration engine. Before orchestration engines, the deployment of an application or service was realized via Continuous Integration (CI) and deployment tools like Vagrant[8] or Ansible[9]. Deployment scripts or plans were created and executed every time an application changed or should be scaled up on a new machine.[vf. 25, p. 70] 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.[vf. 25, p. 70] This flexibility enables a new level of abstraction and automatization of deployment.[vf. 25, p. 70] There are a bunch of orchestration engines in the market. For example Cloudify, 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. 27, p. 8][28, cf.] "These NFs are referred as Virtual Network Functions (VNFs)."[27, 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

[8] https://www.vagrantup.com

[9] https://www.ansible.com

for each function, e.g. routers, firewalls, storages, switches, etc.[28, cf.] For example, high-performance firewalls and load balancing software can run now on commodity PC hardware and traffic can be off-loaded onto inexpensive programmable switches.[29, cf.]

Some benefits are speed, agility and cost reduction in deployment as well as execution manner.[29, cf.] Using homogeneous hardware simplifies the process of planning and reduces power, cooling and space needs.[29, cf.] Through virtualization providers can utilize resources more effectively, by allocating only the necessary resources for a specific functionality.[29, cf.] Overall NFV can reduce Operating Expense (OpEx) as well as Capital Expenditure (CapEx) and can decrease the time necessary to deploy new services to the network.[29, cf.] To achieve NFV the ETSI has defined a framework, the Network Function Virtualistion Management And Orchestration (NFV-MANO)[10], coordinating and orchestrating the NF into the cloud. The Organization for the Advancement of Structured Information Standards (OASIS) created TOSCA, a data model and templates description that can be used for NFV.

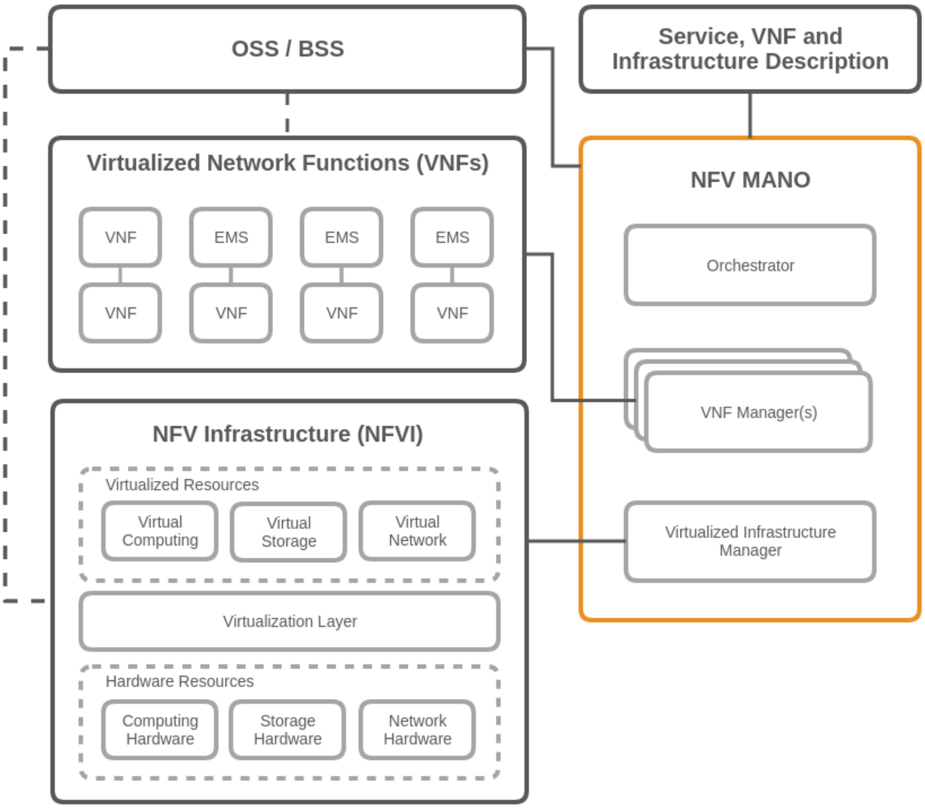


Figure 4: NFV architecture. Adapted from: [30]

Figure 4 shows the NFV architecture in total. The whole structure can be separated into smaller subsections.

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

OSS / BSS refers to the Operations Support Systems (OSS) and the Business Support Systems (BSS) of a telecommunication operator.[31, 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.[31, cf.]

VNFs are the virtualized network elements, for example a virtualized router or a virtualized firewall. Even if only a subfunction or a subcomponent of a hardware element is virtualized, it is called VNF.[31, cf.] Multiple subfunctions can act together as one VNF.

Element Management System (EMS) is responsible for the functional management of a single or multiple VNFs.[31, cf.] This includes fault, configuration, accounting, performance and security management.[31, cf.] Furthermore the EMS itself can be a VNF or it can handle a VNF through proprietary interfaces.[31, 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 can 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).[31, 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"[31]. 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.[32, 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 that is managing network services of VNFs. It is responsible for the global resources management, such as computing and networking resources among multiple VIMs.[31, cf.] The orchestrator interacts with the VNFM to perform actions, but not with the VNFs directly.[31, cf.] TOSCA is often used with NFV-MANO frameworks like Cloudify[11] or Open Baton.[32, cf.]

TOSCA is developed by the OASIS and can be used to deliver a declarative description of a NFV application topology for network or cloud environments.[32, cf.] It is not part of the NFV-MANO standard, but it works pretty well together with it, to automate the deployment and management of NFs and services. In figure 4 it is represented by the *Service, VNF and Infrastructure Description* block. Besides that, it can also be used to define a workflow which should be automated in a virtualized environment.[32, 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.[32, cf.] To

[11] http://getcloudify.org

define schemes, relationships and the configuration of such an infrastructure, it uses YAML files for ease the usage.[32, cf.]

2.3 Existing tools

Using frameworks and third party tools can reduce the time spend on developing software by solving well known or sometimes very specific problems. Based on the framework and tools they can be more secure, because several development iterations were done during creating them and they provides a standardized system through which users can develop applications. They can also allow to create a prototype of an application in a short amount of time. A drawback is, that the user has to spend some time to learn the concepts, functions and how to use a framework, to achieve the benefits. Another downside could be, that specific problems are not solved by the library and sometimes complicate workarounds have to be made to add missing features. Moreover, it can take some time to find an appropriate framework that fit the needs, is well tested and has a good documentation. Therefore, finding and learning good frameworks and third party tools can be a time consuming task. Some of them are elaborated in the following.

2.3.1 Linux Containers

When we talk about container virtualization nowadays, Docker is one of the most famous tools in the market. It is based on Linux Containers (LXC)[12], a technology which uses kernel mechanisms like *namespaces* or *cgroups* to isolate processes on a shared OS.[cf. 5, 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. 5, 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."[25, p. 72] There are several other container virtualization tools in the market like OpenVZ[13] or Linux-VServer[14]. 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 is based on LXC. It extends LXC with a kernel- and application-level API that is used to isolate the CPU, memory, I/O and network layers.[cf. 33, p. 82] Docker also uses the *namespace* or *cgroup* mechanisms to isolate the underlying operating environment.[cf. 33, p. 82] Similar to VMs, containers are executed from images. A major benefit of Docker is, that Docker images can be combined like building blocks. Each image can be used to be the basis for a new one. Figure 5 illustrates the concept for the image of the pretty famous Django[15] web framework. In that case, the Django image, which is the

[12] `https://linuxcontainers.org/`

[13] `https://openvz.org/Main_Page`

[14] `http://www.linux-vserver.org`

[15] `https://www.djangoproject.com/`

resulting image, based on the Python 3.4 image, that 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 is created. File system operations such as creating new files, modifying or deleting existing files are written directly to this layer.[34, cf.] The other images do not get involved. This chaining mechanism of images allows Docker to ease the use of dependencies and administrative overhead.

Besides 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, that 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"[35]. The Docker client is a Command Line Interface (CLI) tool, that interact via a REST API with the daemon.[35, cf.]

Another component, the Docker Registry, is basically a library of Docker images. The most popular is called Docker Hub[16]. There, each package can be available for public or private use. It is also possible to have a local repository stored on the same machine as the Docker daemon or on an external server.[35, cf.] There is also an Docker Store[17] available, where customers can buy and sell trusted and enterprise ready containers and plugins. With the Docker CLI client, it is pretty easy to *search* for new containers and to *pull* containers from or *push* containers to a predefined repository.

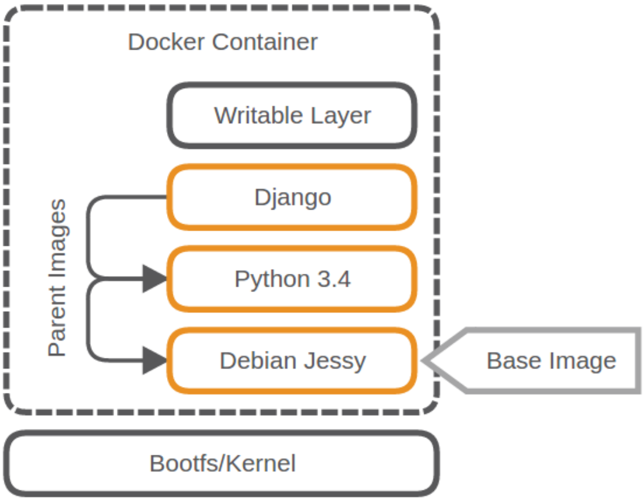


Figure 5: Docker container structure.

With Docker Compose multiple Docker Containers can be executed as a single application. Therefore, a YAML 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[18] image. The main benefit is the ease to configure dependencies between several containers

[16] https://hub.docker.com
[17] https://store.docker.com
[18] https://www.mongodb.com

and configuration steps. This concept is similar to deployment tools like Vagrant[19], Ansible[20] or Puppet[21].

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. 24, 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. 24, p. 2]

2.3.3 Kubernetes

Kubernetes is an open source container cluster manager, that was released 2014 by Google. It is "a platform for automating deployment, scaling, and operations"[36, 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.[cf. 36, 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 itself controls the one or multiple nodes, monitors the system, schedules resources or pulls new images from the repository, to name only a few tasks.

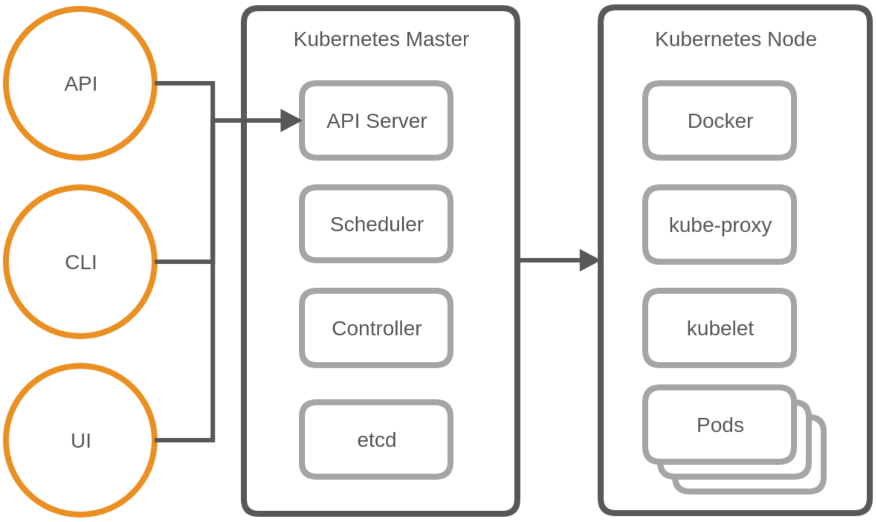


Figure 6: Kubernetes architecture. Adapted from: [37, p. 4]

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 a single one, so called Pod.[cf. 38, p. 7] "Pods are always co-located and co-scheduled, and run in a shared context."[39] One node again can execute multiple Pods. Pods are only temporary grouped containers with a non-stable Internet Protocol (IP) address. After a Pod is destroyed it can never be resurrected.

[19] https://www.vagrantup.com

[20] https://www.ansible.com

[21] https://puppet.com

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 that has access to the service does not care about changes in the service. Any change, either 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. 37, p. 11] Labels can be used to organize and to select subsets of Kubernetes Objects, such as Pods or Services.[40, cf.] They are simply key-value pairs, which should be meaningful and relevant to users, but do not imply semantics to the core system.[40, 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. 38, p. 7] "This reflects services as defined in the Kubernetes API on each node and can do simple TCP,UDP stream forwarding or round robin TCP,UDP forwarding across a set of backends."[41] The Replication Controller is one of the major controllers in a Kubernetes System. It ensures that a specified number of pod replicas are running and available at any time.[42, cf.] For example, if one node disappears, 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. This functionality increases the stability, the availability and the scalability of the system in an autonomous manner. The last important component in Kubernetes is the rolling update machanism. With rolling updates the system can update one pod at a time, rather than taking down the entire service and update the whole system.[43, 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.[44, cf.] No additional software is necessary to have a bunch of machines working 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 keeps 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 scheme 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[22] is an open source ETSI NFV compliant MANO Framework[45, cf.]. "It enables virtual Network Services deployments on top of heterogeneous NFV Infrastructures."[45] It works together with OpenStack and provides a plugin mechanism, that allows to add additional VIMs.[45, cf.] OpenStack is implemented as the VIM and it is the first Point of

[22] https://openbaton.github.io

Presence (PoP).[45, cf.] All the resources in the NFVI are controlled by the VIM, in this case OpenStack.

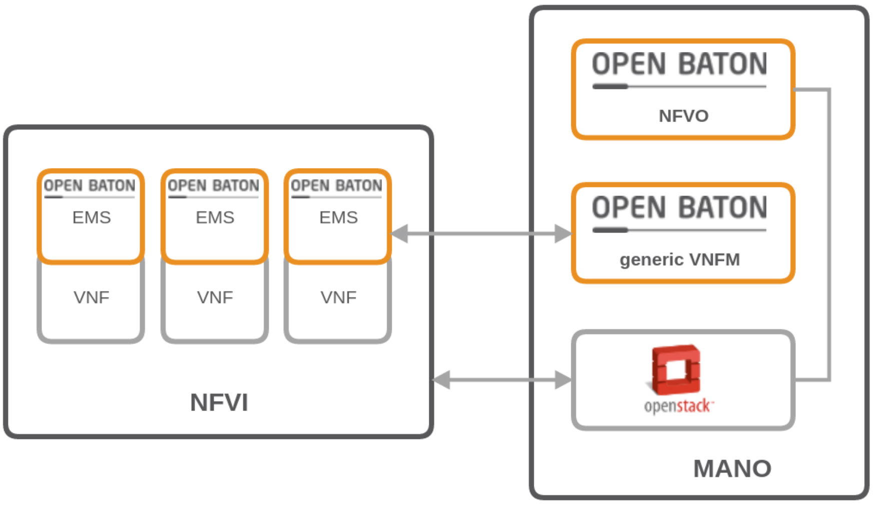


Figure 7: Open Baton abstract architecture. Adapted from: [45]

In the default configuration, Open Baton provides a generic VNFM with a 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) to communicate with the core system. Figure 7 illustrates the abstract architecture of Open Baton together with OpenStack and a generic VNFM. The Network Function Virtualistion Orchestrator (NFVO) is designed and implemented as described in the ETSI MANO standard.[45, cf.] 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.[45, cf.] The necessary informations 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.[45, cf.] The VIM Driver mechanism allows to replace OpenStack with external heterogeneous PoPs, but without the need of modifying the orchestration logic.[45, cf.] Beside the generic VNFM, also the Juju[23] VNFM can be used to deploy Juju Charms or Open Baton VNF packages. Open Baton also provides a marketplace[24] for free and open source VNFs, which can directly be loaded into the system.

[23] https://www.ubuntu.com/cloud/juju

[24] http://marketplace.openbaton.org

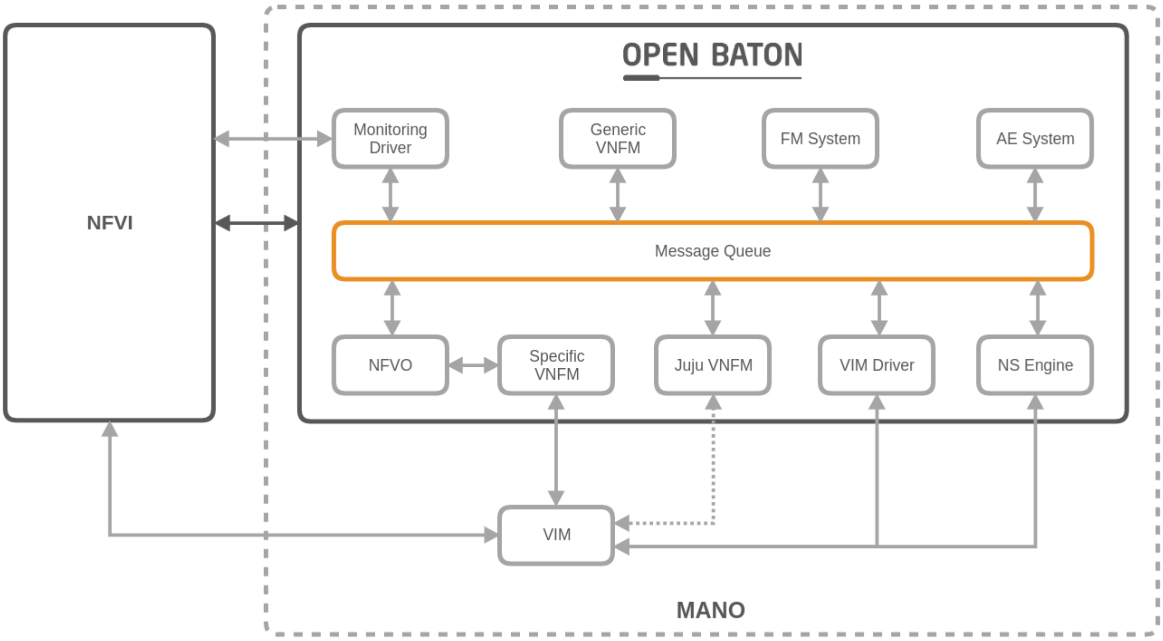


Figure 8: Open Baton detailed architecture. Adapted from: [45]

Furthermore, Open Baton comes with a modern and easy to use GUI and user management. To start a NFVI the included dashboard, REST API or CLI can be used. 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 allocate the necessary resources and instantiate the VM afterwards. After the machines are finally booted, the EMSs will be installed to communicate with the VNFMs. Open Baton now can send node lifecycle events to all the VNFMs responsible for the VNFs, that are part of the network service. Finally, the VNFMs processes the VNFs, that is defined by a TOSCA YAML description, 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.

2.4 Messaging

Messaging protocols are crucial for the IoT area. On the one side bandwidth in an IoT network can be 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.[cf. 46, p. 71] Hundreds or thousands of devices can probably send data at the same time.[cf. 46, p. 71] 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 it is only raw data, like the temperature or the humidity or something similar. All of these circumstances can end up in a huge network traffic. Therefore, it is important that the applications in an IoT context use a lightweight protocol, that has less data overhead and can deliver data packages efficently. Based on the use case, it is acceptable to not receive every single package. For example, in some use cases, where the temperature is measured via a sensor, they will not change drastically in a few millisecond. It would be enough to get the data every second and, especially in this case, it would be probably fine to loose some packages over time. In other cases, like in a healthcare environment, it can be crucial to receive every single package. But, if there are hundreds of sensors sending data, it could be more useful to reduce the bandwidth and keep the latency as low as possible, than getting all the data. If it is important to react to events or issues as fast as possible, the latency should be as low as possible. This also suggests to use a lightweight and fast protocol. Two protocols, that could fit the needs, will be discussed in the following paragraphs.

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.[47, cf.] Meanwhile, MQTT is an OASIS standard[25], which is often used in an IoT and M2M context.[cf. 48, p. 5] It has a publish/subscribe architecture, that makes it easy to implement and allows thousands of remote clients to be connected to a single server at the same time.[cf. 48, 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. For example, a basic workflow could be, that a consumer subscribes 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.[49, cf.] In direct comparison, this concept decreases the traffic, the amount of connections at the server and the delay of the message to be send to the clients.

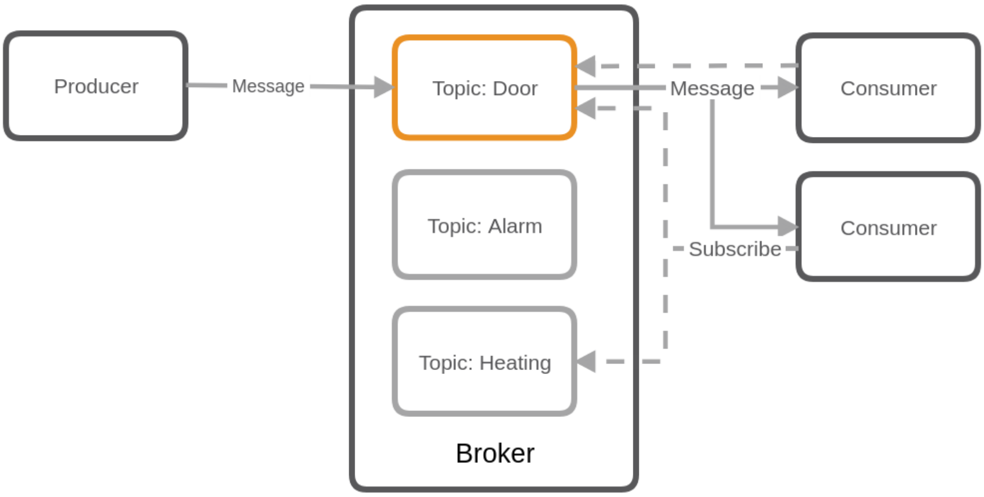


Figure 9: MQTT publish/subscribe architecture. Adapted from: [49]

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

Beside the publish/subscribe architecture and the lightweight protocol, MQTT has only few, but useful features. **Quality of Service (QoS)** defines the level of reliability with which messages are delivered.[49, cf.] There are three different levels, whereas each level differs in reliability and resources usage.[49, cf.] In an IoT context, 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 arrive once, but it can also be lost during transfer. A single message will be send once and the publisher does not check the success of receiving it. This pattern is also called *Fire and Forget* and it is fast and resource friendly.[49, cf.]
At *QoS level 1 - at least once*, it is guaranteed that a message will be received at the consumer. After the publisher sends 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 has the same packet identifier included, so that the publisher will know that the message was successfully delivered. If a message gets lost during transmission, it 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 resource consuming level is *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 not at all. This level has a two-level confirmation process, where at first the consumer confirms the receipt of the message and afterwards the producer confirms the receipt of the confirmation. Therefore, both sides can be sure to not to 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 first 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 instance the information that the connection got lost or something similar. The message persistence will be used, if one consumer loses 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 to be established, even if the consumer disappears. In this case, all the messages incurred, will be stored and delivered, if the consumer resumes the session. The session can be identified with a unique client identifier.[49, 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, it can also be used on devices without a TCP/IP stack, like embedded devices, such as an Arduino.[49, cf.] There, the protocol can be used via a bus or a serial port.[49, cf.] MQTT itself supports the protection of the messages via username and password and the communication can be encrypted with SSL or TLS on the transport layer.[49, cf.] The broker can additionally use client certificates to authenticate them or restrict the access via access control lists, such as IP filtering.[49, cf.] MQTT is available for most of the common programming languages and platforms.

2.4.2 ZeroMQ

ZeroMQ is a 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.[50, cf.] Every message will be sent over sockets via several patterns, like publish-subscribe, fan-out or a simple request-reply.[50, cf.] ZeroMQ has an asynchronous I/O model, which allows to create high-performance and scalable multicore applications.[50, cf.] To distinguish it from messaging frameworks like AMQP[26], ZeroMQ has no dedicated broker in between. The benefit is, that there is no single point of failure, no bottleneck and no need to maintain another component, but ZeroMQ still has the advantages of 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 exchanges the messages directly via memory between threads.[51, cf.] This 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 exchanges 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.[52, cf.]

Pragmatic General Multicast (PGM) is a multicast communication transport using the PGM standard protocol[27] and the datagrams are layered directly on top of IP datagrams.[53, 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 User Datagram Protocol (UDP) datagrams.[53, cf.]

ZeroMQ has several basic patterns and most of them can be combined. Since a detailed discussion of all of them is beyond the scope of this study, only the most relevant ones are considered.
Request-Reply is comparable with a HTTP request-response. The client sends a message to the server, which does some work and sends back a message afterwards. It is represented by the REQ-REP socket pairs in ZeroMQ.

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 know the publisher and has to connect with it. A direct connection between the publisher and the consumer will be established. Similar to MQTT, the publisher will send out the message to every subscribed consumer. It is also possible to subscribe to more than

[26] https://www.amqp.org

[27] https://tools.ietf.org/html/rfc3208

one topic. In ZeroMQ this pattern is represented by PUB-SUB socket pairs. Figure 10 shows this pattern.

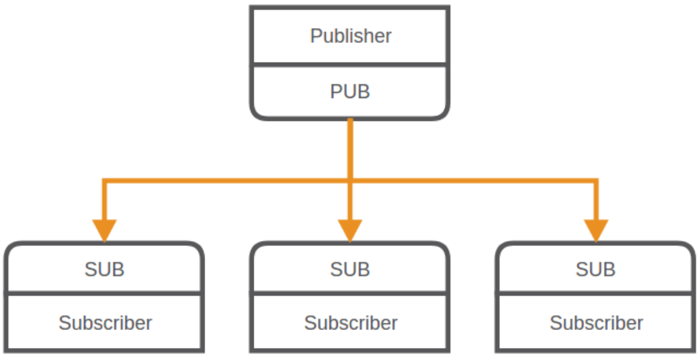


Figure 10: ZeroMQ Publish-Subscribe architecture. Adapted from: [50]

The **Pipeline** pattern is also known as the *Divide and Conquer* pattern. In figure 11 a *Ventilator* can produce multiple tasks, that can be done in parallel.[50] A set of workers can process these tasks and distribute the work between them.[50] Finally, a sink can collects the results from the workers.[50] Benefit of this pattern is, that the workers divide the tasks among itselfs. This means, it will increase the calculation time based on the connected workers. Furthermore, the workers can pull a new task as soon as they have finished one. This means, a worker has a minimal idle time. Queuing is natively provided by ZeroMQ. It is also possible to add and remove workers dynamically. This makes an application much more scalable. Finally, the sink pulls the data from the worker, in a so called *fair-queuing*. This means, the sink will pull one package from each worker successively, then it starts from the beginning and will pull again only one package, even if one or multiple workers should offer multiple packages. The whole pattern is shown in figure 11

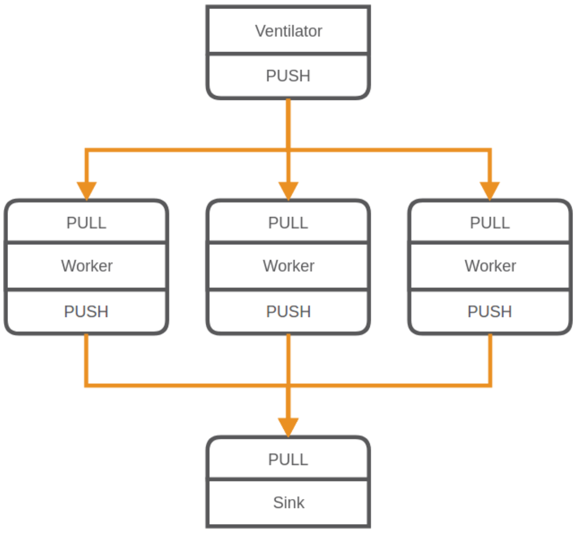


Figure 11: ZeroMQ Pipeline architecture. Adapted from: [50]

Exclusive pair connects two sockets exclusively, for example two threads in a process.[50]
Combinations of them are also possible in ZeroMQ. All of these patterns can be combined to create much more advanced combinations. Figure 12 illustrates such a combination.

There is a combination between two publish-subscribe patterns and a proxy in between. This can be helpful to create a nested topology, for instance 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[28] of ZeroMQ.

By default ZeroMQ has no encryption or authentication mechanism build in. There is a dedicated project called CurveZMQ[29], which enables these functionalities and it is also created by the ZeroMQ maintainers. Since ZeroMQ version 4.x, CurveZMQ comes as a built-in feature. Finally, ZeroMQ has libraries for most of the common programming languages.

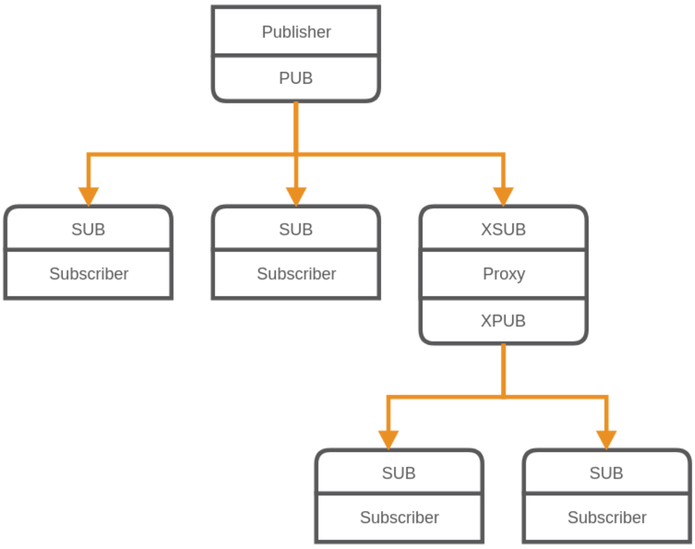


Figure 12: ZeroMQ combination of patterns. Adapted from: [50]

2.5 Related work

An overview of how to apply the concept of microservices for the IoT is shown in [54]. One of the presented patterns uses container virtualization, to split up the services and use them on low power devices. Also in this work, Docker was used as the tool of choice. It describes, that the usage of container virtualization enables a better testability, eases the service deployment and improve the scalability of the tools.[cf. 54, p. 5] The authors also mentioned, that using tools like Docker at the edge of a network, e.g. on sensors and actuators, is nearly impossible

[28] `http://zguide.zeromq.org/page:all`
[29] `http://curvezmq.org`

due to the overhead and dependencies of such tools.[cf. 54, p. 5] Therefore, the concept of fog and edge computing is presented as a viable solution, for executing services on low power devices, like smartphones or small Personal Computers (PCs).[cf. 54, p. 5]

Another study [5] gives some insights about containers and clusters for the edge cloud. It defines edge computing as moving application from a centralized cloud to the edge of the underlying network, in a way that, analytics and knowledge generation services are placed at the source of the data e.g. sensors and actuators.[cf. 5, p. 380] Besides that, the edge network might be disconnected from the cloud environment and still has to perform without any issues. That implies, that edge computing must be able to compute and store data "to address data collection, (pre-)processing, and distribution"[5, p. 380]. If these requirements are fulfilled, such a network is called "edge cloud", which is technically a micro-cloud.[cf. 8, p. 121] Multiple edge clouds can be located side by side in a network. In general, edge devices like a Raspberry Pi or any other small PC, that enables the computation and storage of data, have several sensors and actors connected to them. A Raspberry Pi is powerful enough to do the job. At the same time, it is cheaper and consumes less energy than a normal PC and is easy to integrate into an existing environment.[cf. 8, p. 118]

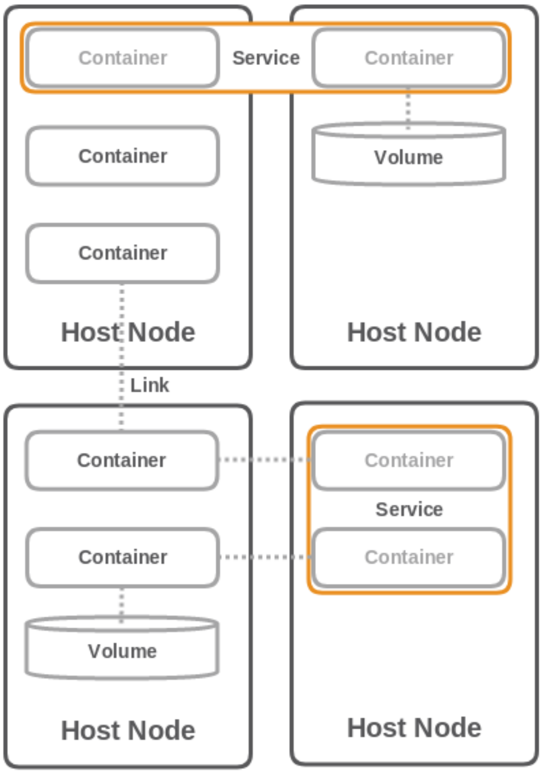


Figure 13: Container-based cluster architecture. Adapted from: [5, p. 384]

Furthermore, this paper describes a so called container-based cluster architecture[cf. 5, p. 384]. Figure 13 outline this. There, each node can be the host of one or multiple virtualization containers. Multiple containers can be grouped together to services, even if the containers are deployed on different nodes.[cf. 5, p. 384] Volumes store data for applications that can persist data on a node.[cf. 5, p. 384] Finally, containers can have dependencies, again also across multiple nodes, to communicate with each other.[cf. 5, p. 384] This behavior is similar to the basic concept of services and pods in Kubernetes.

For the orchestration and services description, the authors point out TOSCA, which is supported by Cloudify. A TOSCA orchestration plan is defined in YAML and is used to deploy the containers on the nodes. Besides that, TOSCA can also describe the infrastructure of the system.

In a another publication, the same authors are use this technological basis to create a Platform as a Service (PaaS), based on a Raspberry Pi cluster[8]. The presented architecture is used to be a bridge "between IoT, local compute devices and data centre clouds"[8, p. 117]. As mentioned before, a Raspberry Pi is powerful enough to do this job. For example, to gather data at the edge of a network and to enable a communication channel with the cloud infrastructure.[cf. 8, p. 117] Container virtualization is used to make the software to be deployed portable and interoperable.[cf. 8, p. 117] Therefore, Docker is used as the tool of choice. To deploy the containers to the edge cloud cluster, a custom tool was developed, instead of using an existing solution, like Kubernetes.[cf. 8, p. 122] This allows more flexibility in monitoring and maintaining the nodes.[cf. 8, p. 122] Furthermore, as the storage solution they used OpenStack Swift. The outcome of the paper was a working environment, that was able to deploy and monitor an edge cloud infrastructure. As mentioned in the paper, there is significant space for improvements, like an standardized and transparent orchestration engine, some data and network management aspects and better semantic descriptions. Overall, this work shows the need of a solid infrastructure at the network edge, to enable edge computing capabilities.

The project of [55] focuses on service discovery, by implementing them via the Serfnode[30] for distributed systems of microservices. Service discovery is a mechanism to detect services and service providers in a network without a centralized management level. Each consumer has knowledge of any other provider and the related capabilities of that provider. Serfnode is a decentralized open source service discovery solution based on the Serf project[31]. It can be used to enable edge nodes to discover other service providers in a network in real time, to facilitate communication.[cf. 55, p. 34] Serfnode itself can be deployed as a Docker container on a node. Unfortunately, there is no Raspberry Pi Docker image available, at the times this thesis was created. After the container is deployed, Serfnode can be configured to enable a service discovery mechanism, that can provide and gather information about one or multiple Docker containers across a cluster, without modifying the original containers.[cf. 55, p. 34] There are several service discovery tools in the market, but Serfnode distinguishes itself as lightweight, platform-agnostic and easy to incorporate with Docker.[cf. 55, p. 34]

It also contains a monitoring and self-healing mechanisms.[cf. 55, p. 34] In addition, Serfnode uses an event handling mechanism that reacts to join, leave and fail events of a node.[cf. 55, p. 38] For example it informs each node in the cluster about a newly appeared node at start-up.[cf. 55, p. 37] Also in this project, YAML is used as the description language for the service description. Beside Serfnode, some other service discovery tools like Consul[32], Synapse[33] and CoreOS[34] are described, with the outcome, that they better fit in a cloud envi-

[30] https://github.com/waltermoreira/serfnode

[31] https://www.serf.io

[32] https://www.consul.io

[33] http://airbnb.io/projects/synapse

[34] https://coreos.com/fleet/docs/latest/examples/service-discovery.html

ronment and are to heavy for an edge node cluster.[cf. 55, p. 36] Also the whole orchestration part is not covered by Serfnode itself.

Another Docker based solution is presented in [56], where the main focus is on the distributed architecture. Basically, also these system services, that are based on Docker images, should be deployed to the edge of the network and the responsibilities of the different components should be separated. Three different layers are provided to enable that behavior:

1. The sensing layer. That is used to gather the data from sensors or to display informations. It should represent a CPS.

2. The mediation layer. It acts as a gateway between the sensing layer and the cloud infrastructure. It contains three software components: a SDN controller, a database and a Machine Learning Unit (MLU).[cf. 56, p. 1534] The SDN controller is used to manage the different software-based network components, like a router or a firewall. The database is used to store the data from the sensing layer. Finally, the MLU provides local intelligence, for example to detect and deploy real-time requirements.[cf. 56, p. 1534]

3. The cloud computing layer is presented as the enterprise layer. This is used for computational tasks, like data-mining and evaluation.[cf. 56, p. 1534] Also the results from the MLU can be analyzed and necessary actions can be performed.[cf. 56, p. 1534] Finally, this layer is also responsible for the orchestration of the services. Further, it works as a centralized control point, available to monitor the system and to perform task on the network. Fault management can be enabled by providing redundancy at different layers and also through the application of orchestration rules.[cf. 56, p. 1535]

The test setup was also made with Raspberry Pis, that represent the edge network devices as well as the gateway. The Docker images are orchestrated with Docker Swarm and custom performance test scripts are created to measure some performance values. As communication protocol a REST based interface was created. The scripts are creating data on the sensing layer, send them over to the gateway, where the data is stored and finally will be fetched from the enterprise layer.[cf. 56, p. 1535] The measured performance for writing the data, starting a container and terminate them again, is presented in the work. Overall the proposed architecture demonstrates that a distributed architecture, composed of several well known components, can be deployed on different devices and can still perform well. Unfortunately, the MLU component was not described and measured results were not adequately analyzed. Nevertheless, the presented architecture worked and the feasibility of such a system was shown.

[57] demonstrate the feasibility of Fog Computing architecture with an extended Kura[35] version as the gateway software and Docker as the virtualization tool. The device layer is realized again with some Raspberry Pi nodes. The main focus of the work is on the gateway layer. Kura itself is an open source IoT gateway software, that can aggregate the data of multiple connected devices and can also control them.[cf. 57, p. 2] It uses MQTT with a publish/subscribe architecture to get the information and also to send it to the cloud.[cf. 57, p. 2] In a sensor-to-cloud architecture, the latency is a drawback and a stable connection can not be guaranteed.[cf. 57, p. 1] The advantage of having a gateway in a Fog Computing

[35] https://eclipse.org/kura

environment is, that the latency can be reduced by shorten the connection distance as well as having a local endpoint, to bypass connection losses.[cf. 57, p. 1]

As presented in the paper, Kura has some limitations. For example be one MQTT broker can be executed and the one has to be placed in the cloud.[cf. 57, p. 3] This can lead to be a bottleneck regarding network traffic. The data from the gateways can only be forwarded to the cloud directly and there is always a persistent socket connection to the cloud, that also ends up in a unnecessary traffic overhead.[cf. 57, p. 3] Therefore, the authors extended Kura and made it able to be used with a gateway-side MQTT broker. This allows to aggregate the data locally in the cluster.[cf. 57, p. 3] There no longer is a need for having a stable connection to the cloud. Which in return enables real-time behavior by analyzing the gathered data and also to respond to it.[cf. 57, p. 3] It is also possible to prioritize the messages to be send and to decide, when to send the data to the cloud.[cf. 57, p. 3] Especially the latter can help to reduce the traffic to the cloud. Finally, this structure is called a "Gateway-cloud" but in fact it is pretty similar to the edge cloud mentioned in [8].

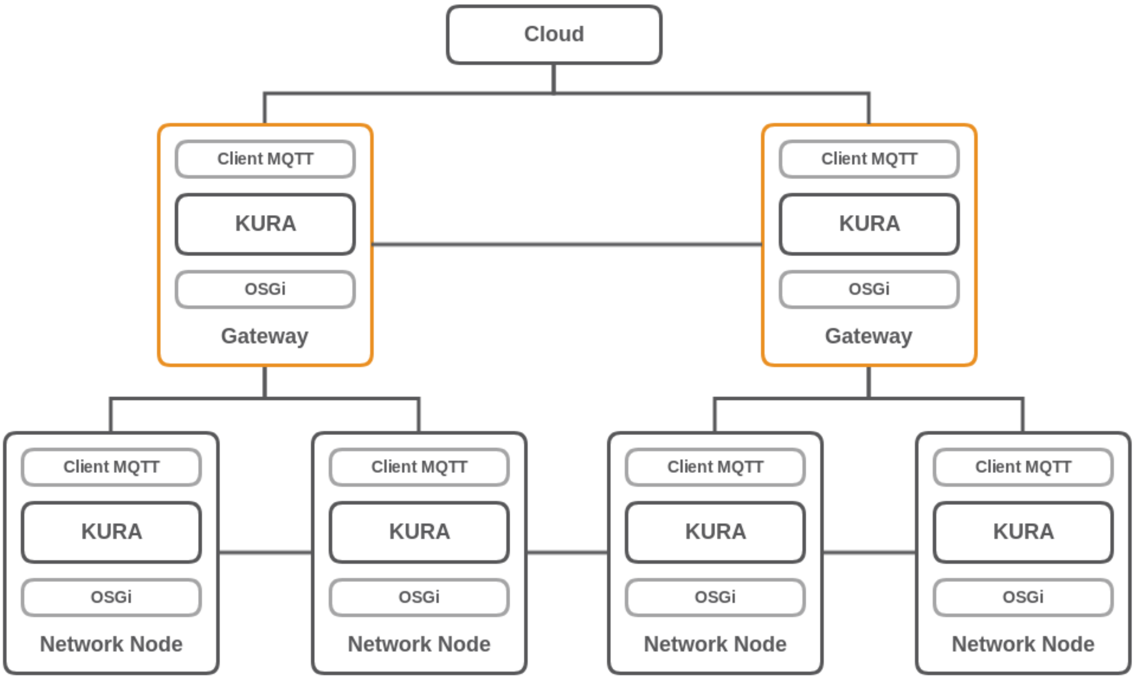


Figure 14: Mesh topologies for Kura gateways. Adapted from: [57, p. 4]

With the help of the new Kura version, new network topologies for the gateways are possible. Two of them, the "Cluster Organization" and the "Mesh Organization", are mentioned in the work and the latter is shown in figure 14.[cf. 57, p. 4] The Cluster Organization is based on a tree hierarchy. Therein, multiple fog devices are connected to one gateway. Further, multiple gateways can exist side by side. They are all again connected to another local gateway, that aggregates the data from all the other gateways. This one is finally connected to the cloud infrastructure. In the mesh organization every gateway and every node can be interconnected to each other, as shown in figure 14. There are also one or more gateways, that are connected to the cloud. If there would only be one track in figure 14, the architecture would change from a mesh organization to a cluster organization. Depending of the use case of an application, the topology can be changed.

For the virtualization part [57] Docker is used as the tool of choice. A fog node skeleton is provided, where each component is separated in the gateway and virtualized by Docker.[cf. 57, p. 6] The orchestration of the containers is again realized by the cloud level. To do so, Docker Swarm, Kubernetes and Apache Mesos are analyzed.[cf. 57, p. 6 f.] As the result Docker Swarm is used, due to the fact, that it is already integrated in Docker and has a sufficient performance.[cf. 57, p. 7] Conclusively, the project demonstrated that a gateway layer is feasible to implement and makes a system robust and independent from the cloud layer.

The authors of [58] also created a gateway layer for the IoT, but additionally made it available as a Gateway-as-a-Service (GaaS). This means, the gateway is still a layer between the end user and the application or the cloud layer and the device layer. Moreover, it can start and stop necessary containers on demand. This is called "On-demand activation" of applications.[cf. 58, p. 3] To do so, a socket-activation framework is used.[cf. 58, p. 3] Unfortunately, this framework is not named in the paper. The applications, that should be activated on-demand, are virtualized with Docker, as in the other papers, and they are also executed on Raspberry Pis. As also discussed in the other papers, a container orchestration tool is used, sadly also this tool remains unspecified.

Another approach in [58] is the multi-tenant platform. Due to the isolation of the Docker containers, the gateway can be shared between different tenants, that are used by different end users.[cf. 58, p. 2] This is useful to provide data for different users through the same gateway. It is also possible to start multiple containers with the same application to provide customized usability. For example in the performance tests of the work the authors started multiple containers with a Wildfly[36] web server and an Elasticsearch[37] instance.[cf. 58, p. 2]

The performance results are similar to the test results of the other references. Docker is comparable with a native environment, beside the network throughput, which is lower.[cf. 58, p. 4] This also affects the power consumption during network communication, which is a bit higher than in a native environment.[cf. 58, p. 4] Overall, Docker performs good enough to be used for this use case.[cf. 58, p. 4]

[36] http://wildfly.org

[37] https://www.elastic.co/products/elasticsearch

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 are important steps to successfully realize a project, not only in the software development area. 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 support OpenStack as the ETSI MANO VIM layer. Most MANO tools like Open Baton use 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 time 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 NF, 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, this could be 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, that 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 node for the desired NF. The whole functionality should work similar to the labels in Docker Swarm. Therefore, a Docker Swarm node can 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[cf. 59, p. 52 f.]. Figure 15 describes the five steps of a typical lifecycle based on these architecture.

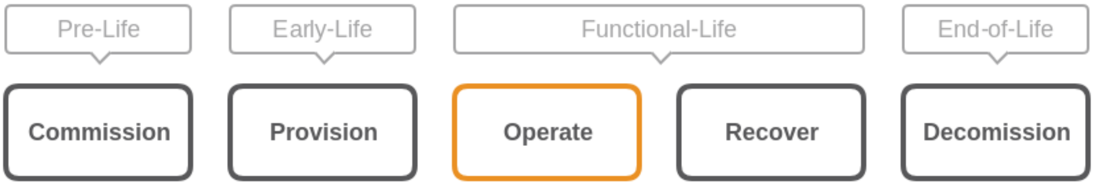


Figure 15: Node management lifecycle. Adapted from: [59, p. 52]

- The **Commission** is the earliest phase in a lifecycle mostly used to perform action like identification, certificates or calibration of time.[cf. 59, 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 advertises features and capabilities.[cf. 59, 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.[cf. 59, p. 53]
- In contrast to that the **Recover** phase performs action if something operates out of norm.[cf. 59, p. 53] The node should be able to recover to the normal state.[cf. 59, p. 53]
- Finally the **Decomission** phase is used for cleaning up sensitive data on the node and to unregister from other components.[cf. 59, p. 53]

In addition to the node lifecycle, an exemplary lifecycle for services and images is shown in the ETSI MANO specification[cf. 60, p. 67 ff.] This lifecycle handles several state from the instantiation of a service, further querying some data, up to the termination of service. Some of the specifications are tightly coupled to NFVs. However, this lifecycle specification is a good starting point to elaborate a custom solution.

The last functional component to be developed will be the GUI. This should only be used to demonstrate the basic functionalities of the prototype. It is not designed to use it in production. Therefore, the GUI will not have any security mechanisms like authorization, authentication or user management. This includes that existing nodes will be displayed with all the related services. Additionally, it can be used to deploy new services 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. It also 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

The non-functional requirements are also addressed and they are 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, it will not be production ready. Nevertheless the prototype will be developed with stability and robustness in mind and the code will be well tested and documented.

Performance: As a crucial requirement, extra attention will be payed for making the application as lightweight as possible and reduce the dependencies and tool chain. 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. The source code will be well documented and an user guide will be created.

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, security will be considered. Especially in the IoT security becomes important due to several bad examples in the last years[1][2]. Therefore, particular scenarios will be considered and recommendations will be addressed. The focus of the prototype will be on showing up the functionality of the application, without implementing security related mechanisms right down to the latest detail.

[1] https://www.corero.com/resources/ddos-attack-types/mirai-botnet-ddos-attack.html

[2] https://security.radware.com/ddos-threats-attacks/brickerbot-pdos-back-with-vengeance

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. This makes it easier to add new or replace existing components.

Scalability: The prototype will also be developed to be executed on several 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 of the prototype to be developed. These use cases describe 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: It will be the most common use case in this consideration. A service, for example a NF, should be deployed to a single node. Figure 16 illustrates this use case. Therefore, the prototype will be installed on the node itself. A cloud service, such as Open Baton, will orchestrate the deployment. In doing 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 and parse them afterwards. All the provided 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.

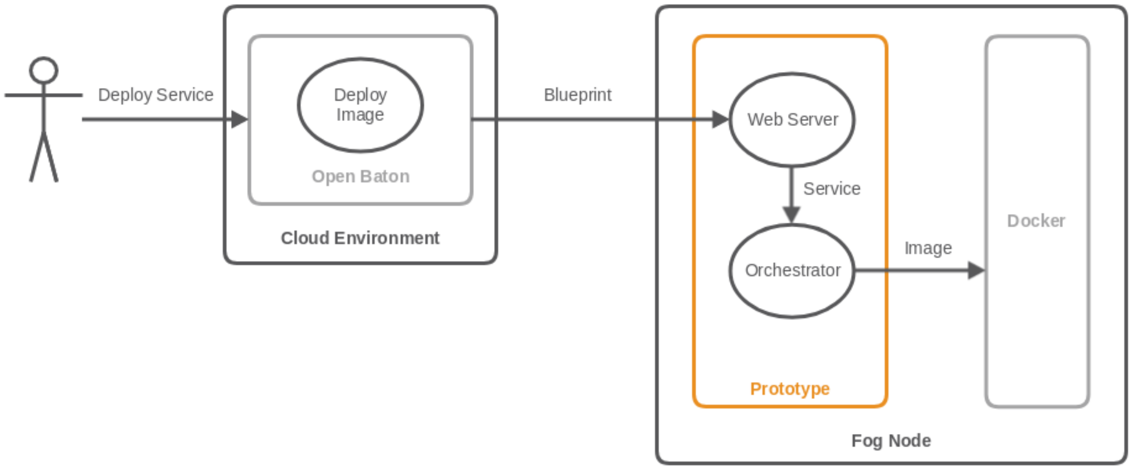


Figure 16: Use case: Service deployment from a cloud orchestrator

Service deployment based on capabilities: Also in this case a cloud service like Open Baton will deploy a service to a node cluster. Figure 17 illustrates this use case. Again, one node will get the blueprint and parse it. Each image has labels, hereinafter called *capabilities*, attached to it. During the parsing of the blueprint, the application is checking, if the node can fulfill all of these *capabilities*. If an image can not used by this node, the application will search for a node, that is able to deploy the image. For example, if one image needs a ZigBee dongle to measure data, than the capability *ZigBee* is added to the image in the blueprint. The first node, Node A, has no ZigBee dongle connected and therefore, it 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 it is also not able to do so. Node C is able and will respond with a related message. Node A sends over the image to be deployed and Node C will execute it. From that point on, Node A is still responsible for the whole service, but Node C is executing the container that was started from the image and will frequently be asked for the state of the container. This process can be repeated for one or multiple images. If each image was deployed, the service is up and running and can be observed again by the user.

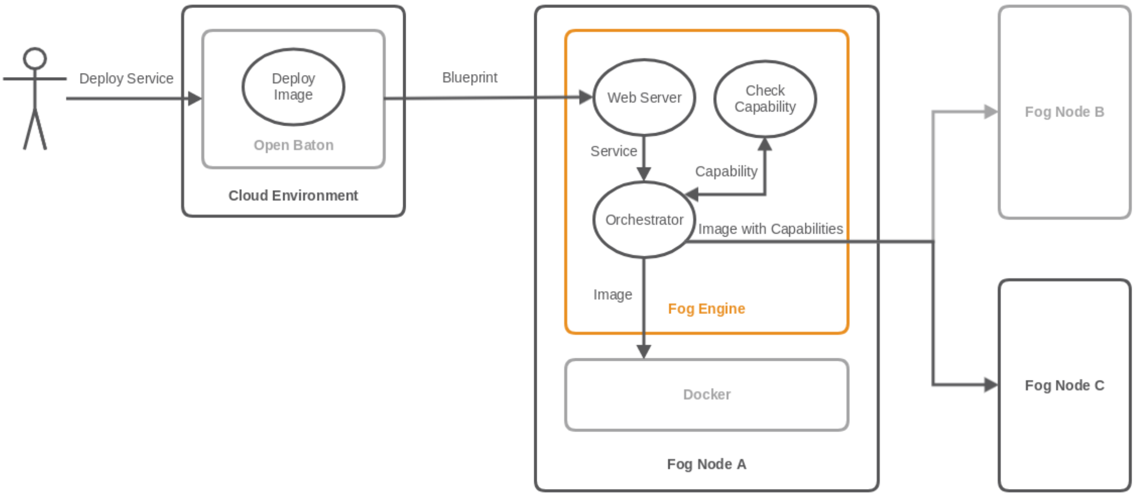


Figure 17: Use case: Service deployment based on capabilities

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 send out a message with some meta information to the broker, with for example the IP address of the node, that send it 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 this 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 transparently and comprehensibly. Therefore, the different components should expose their lifecycle state if requested. 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. When it is assumed, that the node is currently setting up all the necessary configurations and tools, it 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 tools and framework to highlight the main differences to the application to be developed. As mentioned before, the intended prototype 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, developed by Google and one of the most popular orchestration tools on the market. It has an active community and is used by several well known companies[61] like eBay[3] and Wikimedia[4]. Due to the fact, that it is exclusively made for Docker, it means that the system is not made to switch easily the underlying container engine if needed. The prototype will fill this gap. It will be developed to be able to execute multiple virtualization methods. An abstract virtualization layer is planed to be added for that need. This makes the system more fleible and extandable for further virtualization methods.

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 in the prototype.

Kubernetes is made for the cloud, which means it is not intended to be used on low power devices. There are some attempts[62][63][64] to do so, but until now there is no official solution for that. The prototype will be developed with the needs of an IoT environment in mind. It will be executable on low power devices and pay attention for lightweight communication prototcols. Furthermore, Kubernetes is not ETSI MANO compliant, but provides an easy to use web UI.

[3] http://www.ebay.com

[4] https://www.wikimedia.org

Docker Swarm is pretty similar to Kubernetes from a functional point of view. It is open source, has an active community and it is also made exclusively for Docker. As mentioned before, this disadvantage will be eliminated by the prototype. The biggest benefit compared to Kubernetes is, that it is natively included in 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 as Kubernetes. 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 manually via the Docker client or the REST API. Labels, or so called capabilities, can be more sophisticated in the prototype and can be added into the deployment schema of a service. This makes the system more flexible and allows deployments to handle concrete needs.

Just as Kubernetes, Docker Swarm is not ETSI MANO compliant and provides no build-in GUI. There are several third party GUIs that fix that issue, but this implies an additional setup and maintenance effort. Due to the fact that Docker Swarm 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 compared to Docker Swarm or Kubernetes. There are much more dependencies and configurations to be made.

Beside the setup effort, it is an well known and established tool for NFVs, it has an active community and it is 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 itself. There are some efforts to move OpenStack over to the IoT infrastructure[65][66], but also in these attempts it will only be used in the cloud level. Also compared to OpenStack, the prototype will have advantages to be used in an IoT environment. It has made to be executed on low power devices and has the same flexibility as OpenStack by switching the underlying virtualization engine.

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.[67, 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 can be used together with several well known tools like OpenStack, Docker or even Kubernetes.[67, cf.] Because of this flexibility, Cloudify can also be used in an IoT environment if an appropriate VIMs plugin is used. Downside is, that Cloudify itself needs an orchestration tool like OpenStack to be used as NFVO. Without an underlying orchestration tool it is limited.

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, so that the user can manage the whole system, as well as a clean command line tool. By using the YAML format to build service blueprints, the creation of them is similar to well known Ansible or Vagrant deployment schemes. 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.

Due to the fact that the prototype will have a REST API, it could be possibly be integrated into Cloudfiy with a minimal development effort. Beside that, it can enrich some of the Cloudify functionalities, for example by adding the capability behavior or an inter-node communication.

CHAPTER 4

Concept and Design

This chapter introduces the architectural design of the prototype respecting the previously defined requirements. Therefore, the used development environment will be analyzed. Followed by the definition of an abstract 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 use and fast to implement, but it should also consider the knowledge of the FOKUS as well as the developer.

4.1.1 Programming Language

Python is one of the most used programming languages with an active community.[68, cf.] It is a dynamic typed programming language including an automatic memory management, which uses reference counting and garbage collection at the same time.[69, cf.] Python can use several programming paradigms like the object oriented programming or functional programming. Core philosophy is making it simple, beautiful, explicit and 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.[70, 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.

Open Baton, as one of the actively maintained project at the FOKUS, is written in Java and 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 be a basic web client, which uses Vue.js[2] as its main framework and some smaller tools like the pretty famous bootstrap framework.

4.1.2 Continuous Integration

Nowadays, Continuous integration is 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[3] will be used to host and maintain the git repository for the Motey project, the pretty famous and seamless integrated Travis CI[4] 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. Beginning with starting a VM. 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.

4.2 Architecture of the system

The overall architecture of the system can be separated into two levels. Figure 18 will point out the architecture. The first level is the *centralized fog level*. For example, this could be 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 handling

[1] https://pypi.python.org/pypi/pip
[2] https://vuejs.org
[3] https://github.com
[4] https://travis-ci.org

of deployment plans for the NFs. Open Baton will also have an overview of all existing nodes and can manage and maintain them.

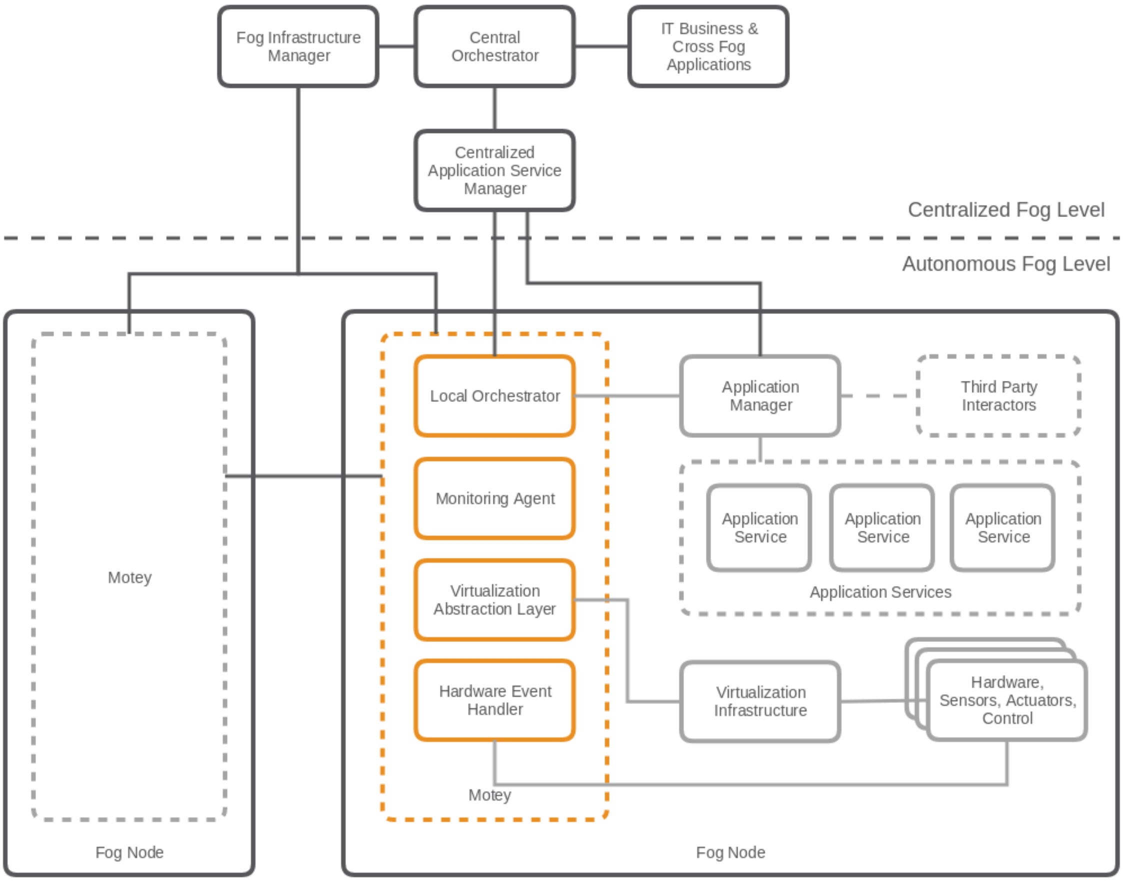


Figure 18: Abstract architecture design based on a draft of the FOKUS

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 18. Each node must have a running instance of the OpenIoTFog Agent to be part of the system. Besides 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. This infrastructure can create the containers and VNFs. Further, it is also possible to have additional third party tools installed which can interact with 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.

The *Local Orchestrator* is one of the most important components. 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 way, 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 others. Besides 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[5] 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 will be provided. 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) decouples the components and increases 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 the facts metnioned before 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, the Yapsy library will be used to realize the plugin functionality. 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 it is lightweight by design. Each supported virtualization engine needs its 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.

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

4.2.2 Communication layer

The communication layer is a pretty important component of the Motey engine. Therefore, three different communication points are necessary to provide the basic functionalities 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 itself to the cluster. This means in the moment of the startup of a node, it will send out a message, with an information request 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 back to the broker. The broker then sends this information message to all nodes 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. The MQTT broker Mosquitto[6] will be the tool of choice. It is an eclipse[7] 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"[71].

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 inter-node communication, also an intra-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 act as the subscriber so that multiple publishers can push messages to them. Besides 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 on the market 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[8] will be used. It is a lightweight and robust Python web server, which is open source, well documented and under constant development. The REST API itself will follow the Hypermedia As The Engine Of

[6] https://mosquitto.org

[7] http://www.eclipse.org

[8] http://flask.pocoo.org

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 replaced easily by any other technology if necessary. This also makes the code more readable and leads up to a cleaner code structure.

4.2.3 Data layer

Usually 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[9] 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 section 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 itself. Primarily 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. Besides that, the orchestrator will also handle the lifecycle of a service and the related images during the deployment phase. Figure 19 shows up the lifecycle management of a service from the starting phase to the execution phase.

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

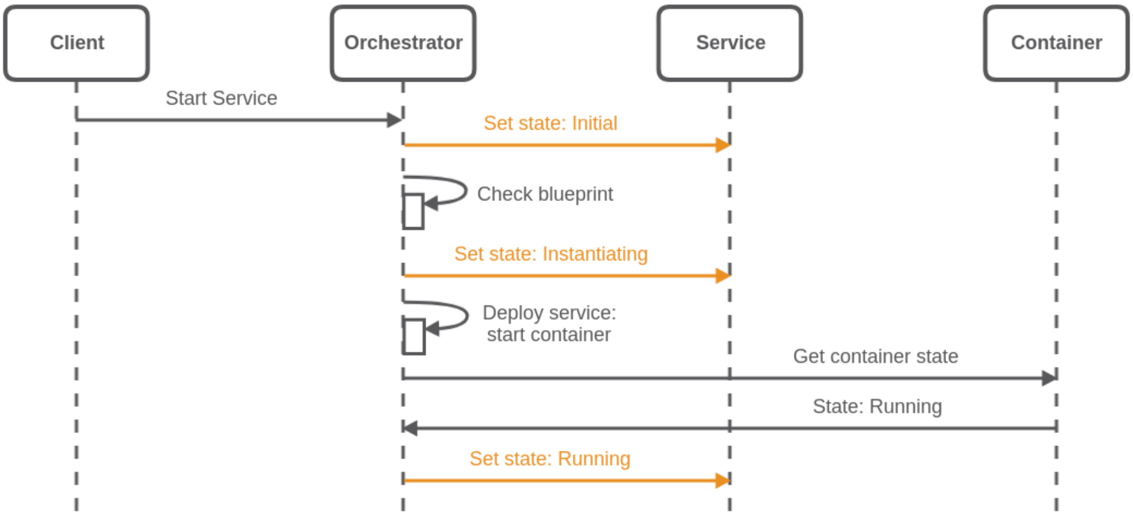


Figure 19: Lifecycle management sequence diagram: starting a service

At first the client requests the orchestrator via the REST API, to 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 *initial* state. Afterwards the blueprint gets checked by the orchestrator. If it is not valid, the state of the service will be set to *error*. In this case everything is 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. So the terminate request is sent to the node. Again the orchestrator will get it and will set the state of the service to *stopping*. Then it will start to shutdown all the related images. Even 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 20 illustrates this behavior.

The lifecycle states of the containers are not handled by the orchestrator itself, they are 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 container changes 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.

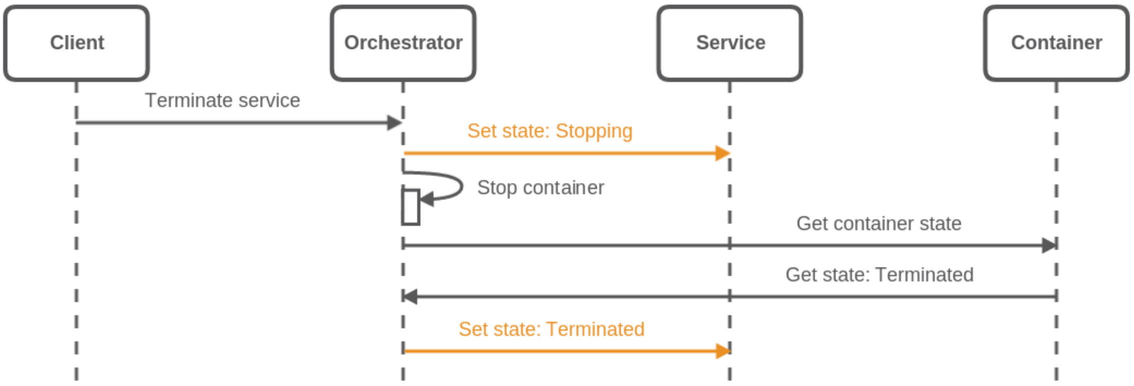


Figure 20: Lifecycle management sequence diagram: stopping a service

4.2.6 User interface

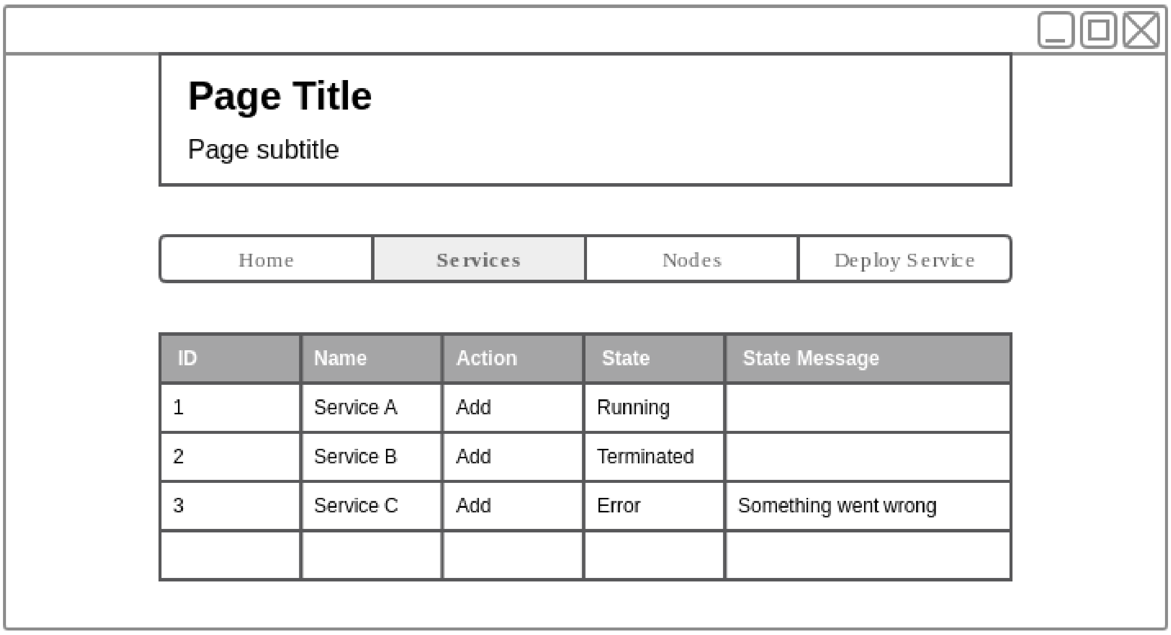
As mentioned in section 3.1 the GUI will only be used for demonstration purposes and is not designed for use them in production environments. Figure 21 show the mockup of the web GUI.

The page has 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 a blueprint. The created REST API will be used to get all the necessary informations to be displayed and to deploy new services.

A web layout based on Bootstrap[10] as the Cascading Style Sheets (CSS) framework will be created. Further, Vue.js[11] is used as the JavaScript frontend framework. According to Wappalyzer[12], Bootstrap is the most used CSS framework on the market, with a share of 65.2% (as of 26. June 2017). It is elaborated and there are many templates available.

Vue.js on the other side is relatively new on the JavaScript frontend framework market. It is much slimmer then tools like Angular.js[13] or React[14], 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 be 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 fact that it is only a prototypical implementation for demonstration purposes only.

[10] http://getbootstrap.com
[11] https://vuejs.org
[12] https://wappalyzer.com/categories/web-frameworks
[13] https://angular.io
[14] https://facebook.github.io/react



Page Title

Page subtitle

ID	Name	Action	State	State Message
1	Service A	Add	Running	
2	Service B	Add	Terminated	
3	Service C	Add	Error	Something went wrong

Figure 21: Mockup of the web GUI

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[15] or the BrickerBot[16] infected thousands of devices in 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* lists the most common vulnerabilities in IoT projects[17]. Some of them are 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

[15] https://www.corero.com/resources/ddos-attack-types/mirai-botnet-ddos-attack.html

[16] https://security.radware.com/ddos-threats-attacks/brickerbot-pdos-back-with-vengeance

[17] https://www.owasp.org/index.php/OWASP_Internet_of_Things_Project

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.

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 will 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 the 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 vary.

5.2 Project structure

The project can be found on Github at `https://github.com/Neoklosch/Motey`. The main folder contains the Motey engine. Besides that, the project contains several directories with helpful scripts and tools. A brief overview about the project structure will be 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 evaluation 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 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. Listing 5.1 shows the CLI 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.

```
Motey command line tool.

Usage:
  motey start
  motey stop
  motey restart
  motey -h | --help
  motey --version

 Options:
   -h, --help       Show this message.
   --version        Print the version.
```

Listing 5.1: Command line interface documentation for the daemon process

dependency-injector is a micro framework 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 mock the object with the help of a mocking library. To realize DI in Motey, a so called *app_module.py* was created, which uses the *dependency-injector* framework to create the dependency graph. Several Inversion of Control (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.

```python
class DIRepositories(containers.DeclarativeContainer):
    capability_repository = providers.Singleton(CapabilityRepository)
    nodes_repository = providers.Singleton(NodesRepository)
    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 it. This can be realized via a URL to the REST API or via a 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. It will support extensions to add functionalities, so that the developer can choose the tools of choice.[72, cf.] Nevertheless, Flask is production ready and is used in several big projects like Pinterest[73] or Twilio[74].

Flask can use so called *Blueprint* to configure new routes in the web server. 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 a 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 added via the *add_url_rule* to the Flask web server. 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 simplify the usage. A wrapper class to centralize the usage of the library was created and the configuration, as well as some smaller improvements, were made in this wrapper class. The whole configuration of the client can be configured via the

[1] http://www.eclipse.org/paho

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 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 script handles also the deployment of the documentation. Therefore, the files will be generated, the current branch will be switched to *gh-pages*, which is used to display the Github page at `https://neoklosch.github.io/Motey/`. Afterwards, new commit will be pushed with an auto-generated commit message. Finally, the branch will be switched back again. "Read the Docs" has an active webhook that generates the builds, the current documentation and displays 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. Nevertheless, 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 an 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 be replaced by a different one, without refactoring several classes 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 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

[2] `http://readthedocs.org`

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

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

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 have 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*.

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. The daemon creates a pid file which can be configured via the *config.ini* file. The dependencies of the most important components are shown in figure 22.

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 a 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 use the communication layer.

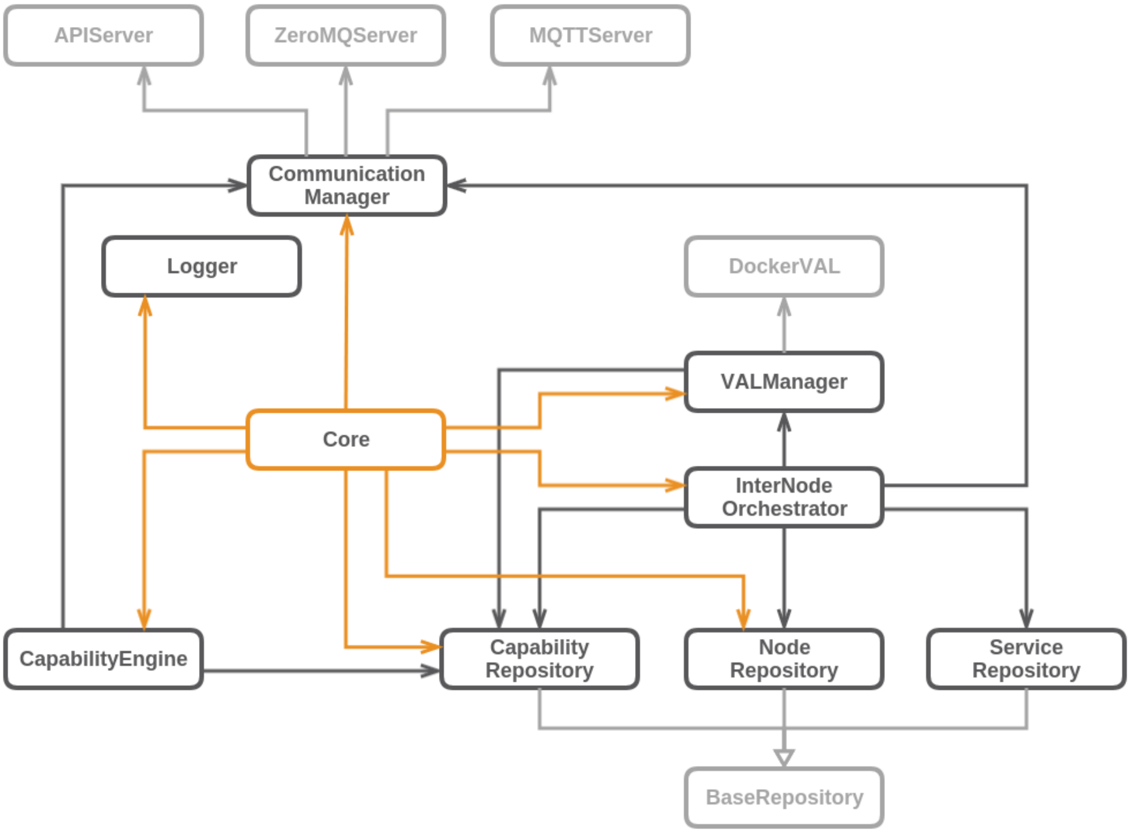


Figure 22: Motey class diagram

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* is used. The repositories are a special case of the decorator pattern, because they cover the usage of the *TinyDB* library, but they are also a centralized place for using the library, instead implementing them all over in the engine. The *CapabilityEngine* and the *InterNodeOrchestrator* are the only components that use the *CommunicationManger*. The *Core* also imports that class but only to start it. Beside the *Core* the *InterNodeOrchestrator* is the central place in the app. Each event will be executed in that layer. Therefore, most of the other components converge 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. Besides 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.

```
[GENERAL]
app_name = Motey
pid = /var/run/motey.pid

[LOGGER]
name = Motey
log_path = /var/log/motey/
file_name = application.log

[WEBSERVER]
ip = 0.0.0.0
port = 5023

[MQTT]
ip = 172.18.0.3
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 are imported in 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.

In the blueprint, the capabilities assigned to an image, can have three different conditions:

1. there are no capabilities assigned to an image, therefore the current node starts the image and manages the resulting container.
2. all the assigned capabilities can be fulfilled by the current node. This node then starts the image and is responsible for the resulting container.
3. one or more capabilities can not be fulfilled by the current node. Another node that meets the requirements must be found to handle the deployment of the image.

The capabilities of the current node are fetched via the *CapabilityRepository*. In each case the image will always get an IP address of the node that is handling it, even if it is the current node. Later on the communication layer will deploy the image via a 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 its capabilities. They will be passed back to the orchestrator, which compares them with the capabilities of the image. If all of them are fulfilled, 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. If non of the nodes fit, 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 an *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. Figure 23 shows the mapping logic of the lifecycle states.

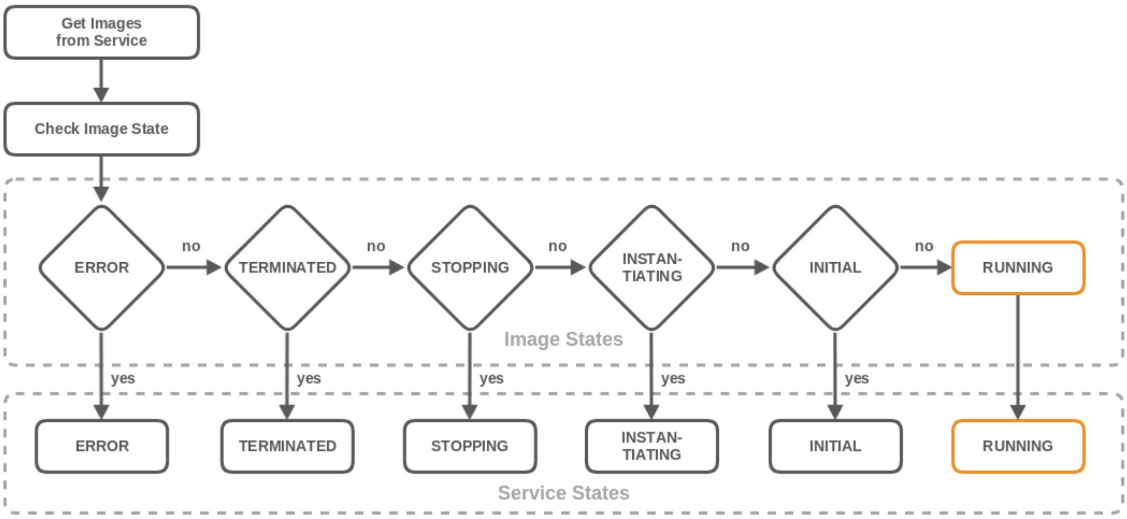


Figure 23: The mapping of the service lifecycle state

Important while mapping the lifecycle states is, that each status will be requested via a ZeroMQ endpoint. Also in this case a request-reply call between the nodes will be executed. This is performed by the *CommunicationManager* again. After all states are received and stored in a list, the mapping starts. Then, the list will be searched for each state. If one of the shown states is found, the related state in the service is set. If for example one of the images is in the *STOPPING* state, also the service is marked as *STOPPING*. 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 web server, to block the main thread, because it has to run infinitely 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, to 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 respons 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 executes 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.

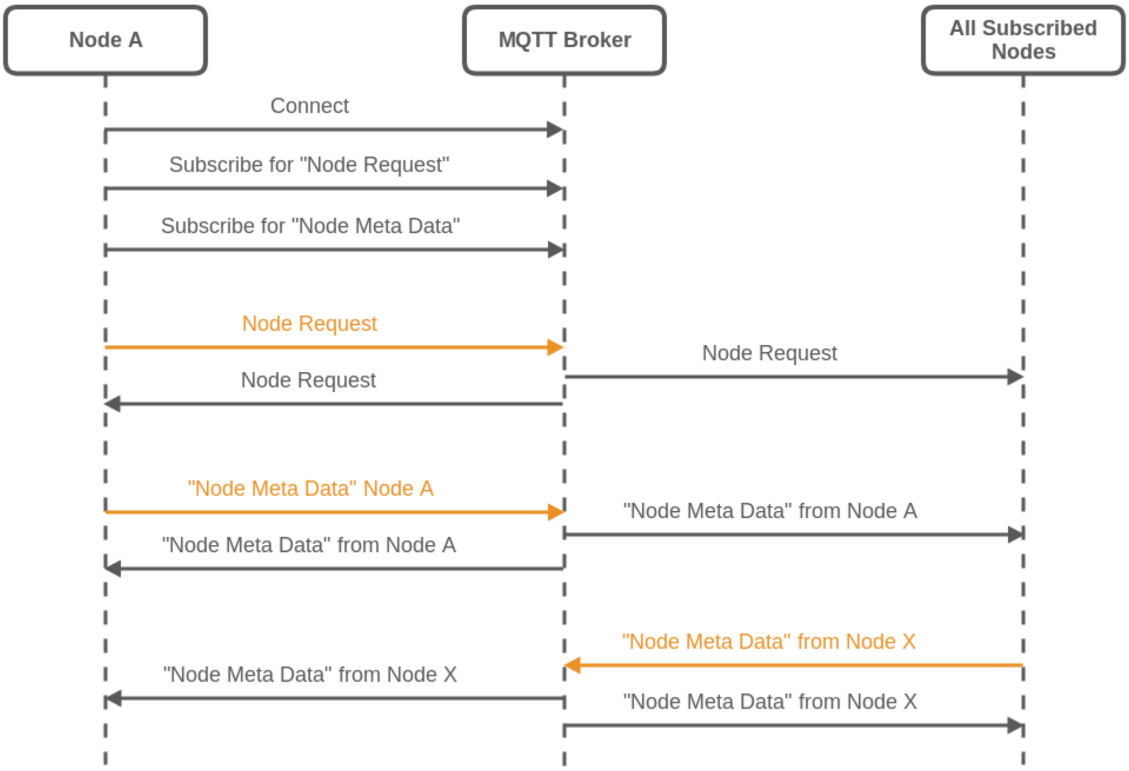

Node A
MQTT Broker
All Subscribed Nodes
Connect
Subscribe for "Node Request"
Subscribe for "Node Meta Data"
Node Request
Node Request
Node Request
"Node Meta Data" Node A
"Node Meta Data" from Node A
"Node Meta Data" from Node A
"Node Meta Data" from Node X
"Node Meta Data" from Node X
"Node Meta Data" from Node X

Figure 24: Node discovery sequence diagram

Figure 24 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 itself. 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.

This is also the reason why the "*Node Meta Data*" *from Node A* and "*Node Meta Data*" *from Node X* are duplicated in figure 24. Besides that, the procedure is straight forward: At first one node sends out a *Node Request*. Each subscribing 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 node has 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 disappear. All the nodes react to this by removing the meta data about this node from the database.

ZeroMQServer Sections 2.4.2 and 4.2.2 gave a detailed overview about ZeroMQ and how it works. Now the concrete implementation will be discussed. The wrapper class *ZeroMQServer* binds multiple sockets to the related ports. There are four sockets bound 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 publishers 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 5.8 shows 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 5.8: 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* has to be prepend to the message, followed by the delimiter. Listing 5.9 show such a message.

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

Listing 5.9: Example ZeroMQ capability event message

Therefore, the topics and the delimiter have 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. The current state of an image will be detected by the virtualization engine. Afterwards, it will be 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.

An important aspect, while using the Request-Reply pattern, is, that there can only be one connection established at the same time. In addition, a reply must send by the consumer after a request was received. As long as there was no message sent back, the connection will be blocked and the consumer can not receive any new messages.

As discussed by the other servers, also this one has to be executed in threads. The 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 replacing or adding a new communication component. Besides that, the manager simply forwards the methods to the specific server. It is also used as a central place to start and stop all servers.

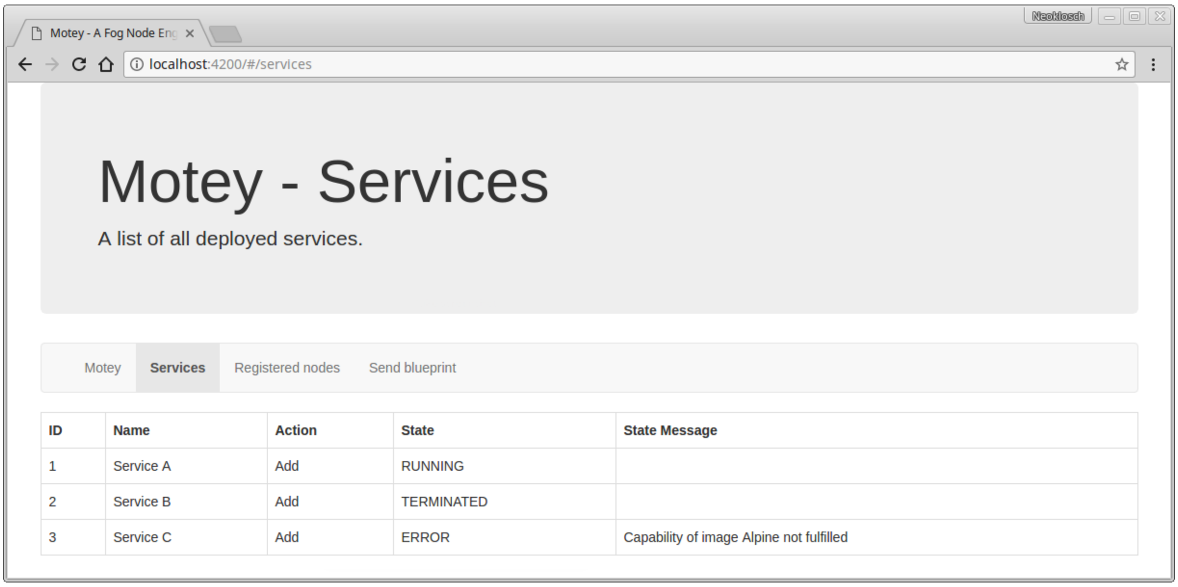
5.4.5 Capability management

Similar to the InterNodeOrchestrator, the *CapabilityEngine* is used as a connector between the communication layer and in this case the *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 react to add and remove requests of any external application. Due to the fact that the endpoints are implemented via the ZeroMQ IPC protocol, only applications 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 eemoval of an entry is pretty similar. The received JSON data will be again validated 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 25 shows the final 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 screen-shot, 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.



Motey - Services

A list of all deployed services.

ID	Name	Action	State	State Message
1	Service A	Add	RUNNING	
2	Service B	Add	TERMINATED	
3	Service C	Add	ERROR	Capability of image Alpine not fulfilled

Figure 25: Screen shot of the Motey GUI

The *index.html* file also contains 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 fetch the JSON data from the REST API and map them to the related template in the HTML file. The *BlueprintTemplate* controller gets the data from the textarea and sends 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, using a *.travis.yml* configuration file. The content of the file is shown in listing 5.10.

```
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 5.10: Travis CI configuration file

There are some important parts in it. At first the script needs root privileges to build and upload Docker images. Line 1 adds this privilege and in line 2 the Docker service is requested. The programming language the application 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 build one Docker image and should only upload them once, only one python version needs to be used. Otherwise multiple python version would be executed and after each successfully build version, a container would be build and uploaded.

Line 9 defines the installation script. This will executed after the virtual environment is 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 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`.

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

[6] `https://docs.djangoproject.com/en/dev/internals/contributing/writing-code/coding-style`

The *after_success* block is used to create and upload the Docker image. Besides 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 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 handed over. Due to the fact that this file is under public version control, this sensible information should not be committed. 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 the Raspberry Pi is using a CPU with an ARM architecture. Therefore, Docker images that should support ARM CPUs must have specific dependencies and have 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 normal image, because it loads 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 5.11 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 5.11: Dockerfile to create the Motey Docker image

The *FROM* command in 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. Additionally, 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 have to be executed, that are shown in listing 5.12.

```
python3 setup.py build
python3 setup.py install
```

Listing 5.12: 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.

5.5 Code Verification

Code verification is a quite important technique in every development project. There are several possibilities to check and to 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[7] is used. PEP 8 is used in the Python standard library code and is well established in the community.

```
$ 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: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 5.13: Output of the style check validation from Travis CI build process #148[75]

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

A standardized code style is recommended in team projects as well as in open source projects. But also in private projects 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 5.10. If there is any style check violation, the CI build process will fail and the new Docker image will not be uploaded. Listing 5.13 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 are the unit tests. A unit test is, as the name implies, a software test method, which tests each component of an application. This method is a white box test, because the source code of the components is well known and only the results are tested. 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. 76, p. 320] All the used objects in a class should be as independent as possible from each other. 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 object 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 objects easier to mock, 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 detected much faster. If an error occurs 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 reduce 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 tests are part of the CI, each build will be checked. This reduces the risk of the deployment of faulty code. Finally, unit tests can be a good starting point for new team members, because it helps to understand the functionality of a class. A unit test can act as some kind of documentation.

The downsides are, that unit tests are coded. This also mean, unit tests can have errors and they have to be maintained if the source code of the main project is changed. Furthermore, poorly written unit tests or tests that are written unmotivated, can end up in a wrong feeling of safeness. Some error case could not be caught and the submitted code is still faulty. Additionally, the test setup is not a realistic environment. Integration test are in general more accurate in terms of the human behavior. They test complete workflows and not only single components or functions. Also the dependencies between the components and the interaction between them can be tested much better with integration tests. In many projects, the most critical part of unit tests are, that they take time. For each written component, the related 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 5.14: Extract from the Motey unit test of the ServiceRepository

Listing 5.14 shows an extract from the *ServiceRepository* unit test. The test starts with a *setUp* method (line 3) that is executed before each test. In this case, some static variables will be created (line 4 - 6) and the internally used libraries are mocked (line 7 - 9). Finally, the *ServiceRepository* itself will be created. The *test_has_entry* method is one of multiple tests. This method checks if a value exists in the database. Therefore, the result of the database search method is mocked (line 13) and the *has* command is executed (line 15). Finally, the results of the method are checked. At first the script tests if a specific method is called, in this case the *search* method of the database (line 17). Afterwards, the result of the *ServiceRepository has* method is checked (line 18). If everything went fine, the test will execute with a status 0 as shown in listing 5.13.

Unit testing in general can be realized in two different ways. The first possibility is to write the tests after the coding of the component is done. This is the standard way in many projects these days. The advantage is, that the code is done and normally no big changes come after that. This also mean, changes on the test suite are rare.

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. In an extreme programming environment, TDD, in combination with pair programming and code reviews, is 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 end up in a well tested code base and an assured code stability.

CHAPTER 6

Evaluation

In this chapter the implementation of the plugins, based on the previously defined requirements and concepts, are evaluated. Therefore several performance tests will be analyzed and the code verification will be shown. Afterwards the final system is demonstrated.

6.1 Test Environment

Motey and the performance scripts in section 6.2 were 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 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 device is also running 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

In the following section some performance relevant tests are shown and analyzed. Especially the performance of the used virtualization method is crucial for the system, as well as the connection performance of the used protocols is important. All evaluations are made with low power devices in mind. This means a small overhead and a fast and lightweight solution is preferred.

6.2.1 Docker vs. Hypervisor-Based Virtualization

Due to the fact that Docker is the core component in the created prototype, it is important to verify the performance on low power devices. Unfortunately a performance comparison with established VM tools like VMWare or Xen on a Raspberry Pi is not possible, because nearly all of them don't support the ARM CPU architecture. There are several performance tests on x86 architecture. IBM for example compared VMs, or more specific Kernel-based Virtual Machines (KVMs), with Docker in a research report[77] from 2014. The conclusion was that KVM has a significant overhead to every I/O operation. Therefore it is less suitable for latency-sensitive workloads or high I/O rates. Figure 26 show the throughput for random I/O operations for a native, a Docker virtualized and a KVM virtualized system.

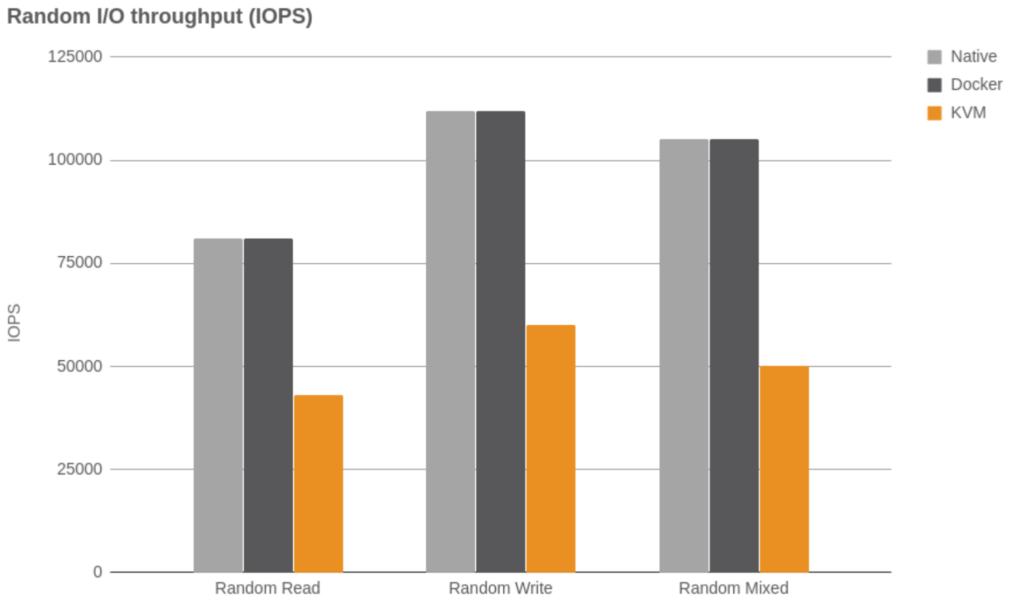


Figure 26: KVM and Docker I/O throughput comparison by IBM. Adapted from: [77, p. 6]

Compared to the native system, Docker has "nearly no overhead"[77, p. 6]. But also Docker has some drawbacks. For example it has an overhead for workloads with high package rates.[cf. 77, p. 6] The overall conclusion by IBM was, that Docker is more performant than a KVM virtualized system.

Another benchmarking comparison[78] between Docker and KVM that was also made by IBM, shows a much clearer result. In this test Docker is much more performant in nearly every point of view. It has less overhead in terms of CPU usage[cf. 78, p. 25], memory performance[cf. 78, p. 50] and boot time[cf. 78, p. 24]. Only the network throughput is nearly the same.[cf. 78, p. 52] Concluding also the container size is much smaller in Docker than in KVM.[cf. 78, p. 66]

[79] consolidates this statement of IBM. Docker has nearly no performance loss compared to the Hypervisor-based virtualization, in this case KVM.[cf. 79, p. 3 ff.] The only thing where Docker can not keep up with a native environment, is while performing network request and response. In direct comparison, Docker only can perform around half of the transactions as a native environment is able to do.[cf. 79, p. 6] Compared to KVM, Docker has a significant performance advantage. In all of the tested areas "the hypervisor-based solution showed a significant overhead that cannot be easily mitigated"[79, p. 6]. Overall Docker is much more lightweight and due to the fact that it supports the ARM architecture, it is well suitable for the usage on low power devices.

6.2.2 Performance HTTP

To test the HTTP performance of the application, two simplified test scripts are created. Both are located in the *performance_tests/http* folder in the Motey project. The *server.py* script starts a Flask server and waits for a POST request. If it gets one, it will send back a static container id. The *client.py* script is used to send the JSON data to the server. It is an extract of a Docker image command, as it would be used in the Motey application. Afterwards 10.000 requests will be send to the server. This execution is one iteration of the test. In total 10 iterations are performed to get a result. An overview of all test results are shown in section A.1.1. Figure 27 shows the result of the tests.

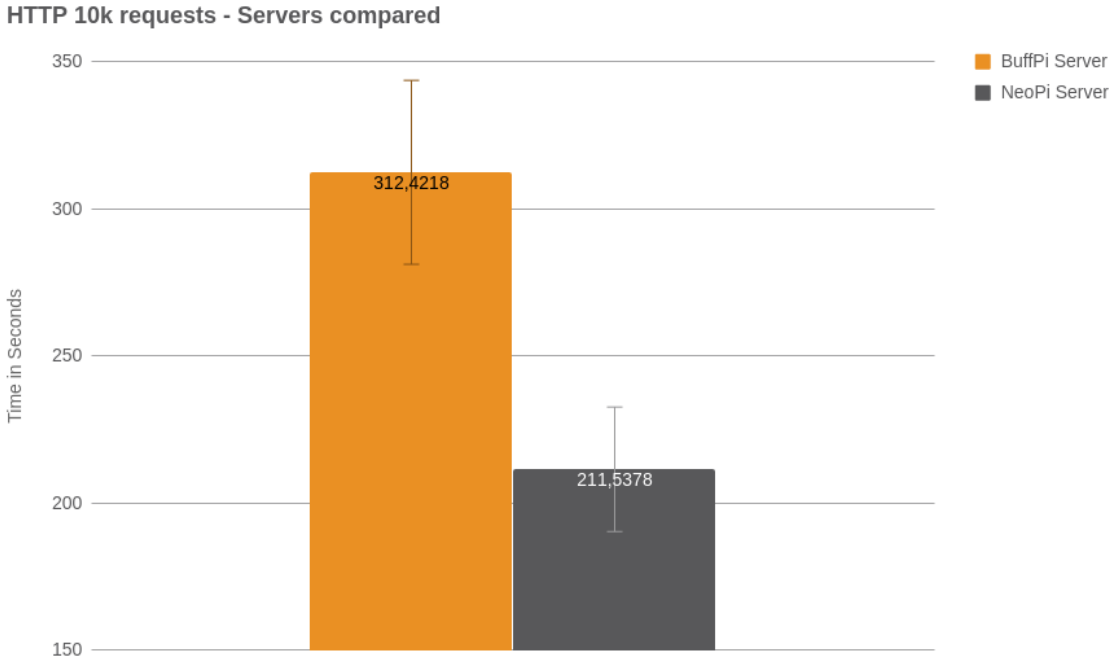


Figure 27: HTTP 10k requests - Server comparison

The two bars indicating the time needed to send 10.000 requests: the left one, that represents a setup, where the more powerful BuffPi was the server and the NeoPi was the client, took in total more time, then the setup where the NeoPi is the server. This is a surprising outcome, because the assumption was that the server takes more computation time then the client. But the results prove the opposite. If the less powerful device is the server the packages are much faster sent and vice versa. An explanation for this could be, that the client has to establish a connection to the server for each of the 10.000 requests and then has to wait for the response. This took computation and also idle time, until the servers respond once. Additionally, all the requests are done sequently, so that the server must handle only a single request at a time. For a web server, that is made to handle several hundred request at the same time, a single request should be handled really fast and with only a few resources. This means the server has a lot of idle time and does not need much resources to handle the request. In this particular case, where the server has only to handle a single request at the same time, there are no advantages for using the more powerful device for the web server. Due to the fact, that more work is made on client side, the packages can be transferred faster, if the client is more powerful. This will change if there is more than a single client connected to the server and more than one request has to be performed at the same time. Overall using HTTP as the protocol of choice, the messaging is pretty slow. To send the 10.000 request, even the faster setup took 03:31 minutes. This result depends on the packet overhead in HTTP and also the need for waiting for the request. Such a setup fits well for simple status or rarely sent deployment requests, but not for high frequently used inter-node connections or in a low latency environment.

6.2.3 Performance ZeroMQ

Also for the ZeroMQ performance test both Raspberry Pis are used to communicate with each other. Similar to the HTTP test, the BuffPi and the NeoPi were switched between server and client. To test the ZeroMQ performance, the test has to be splitted up into two different variants. The first one is called *Constant Connection (CC)*. This means, once a ZeroMQ connection is established, all of the 10.000 request are send via this one session. *New Connection (NC)* is used to simulate the case where each connection has to be closed, before a new request will be send. This is more realistic to the implementation in Motey, because the connection to one node will established in the moment they are needed and closed as soon as the request is done.

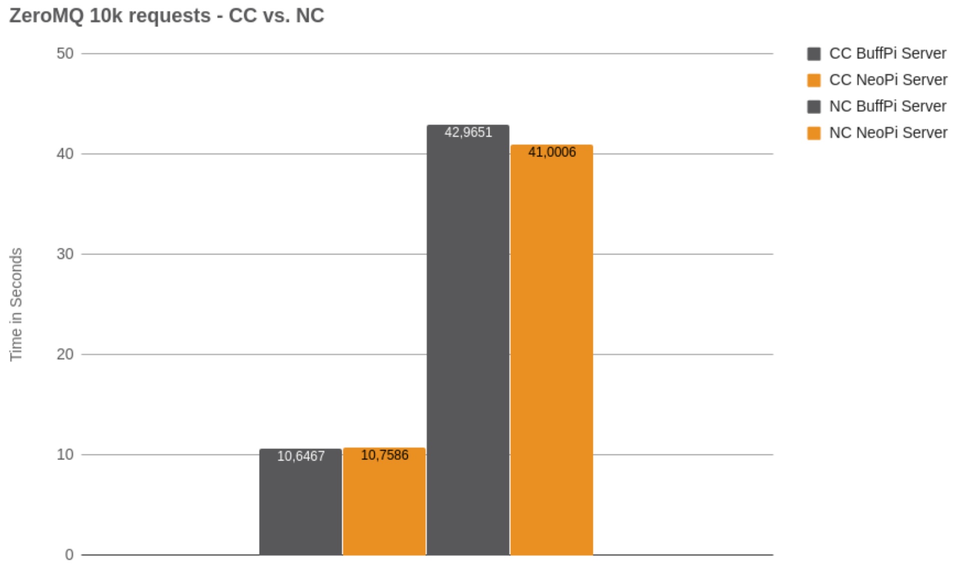


Figure 28: ZeroMQ comparison between new connection and constant connections

Besides that, the test setup is the same like in the HTTP performance test. The scripts are located in *performance_tests/zeromq* folder, the *server.py* script is the same for both variants, only the client scripts differ. Each client sends out 10.000 requests and each iteration is performed ten times. The total test results are in section A.1.3.

The difference between the CC and NC is significant. The CC is around four times faster then the NC. Certainly this relates to the connection and disconnection phase of the socket. On the other side the difference between the two devices is minimal and can be neglected in this evaluation. The comparison between ZeroMQ and HTTP is much more crucial. Compared to the CC, HTTP is around 20 times slower. Even compared to the NC it is around 5 times slower. Figure 29 shows the test results.

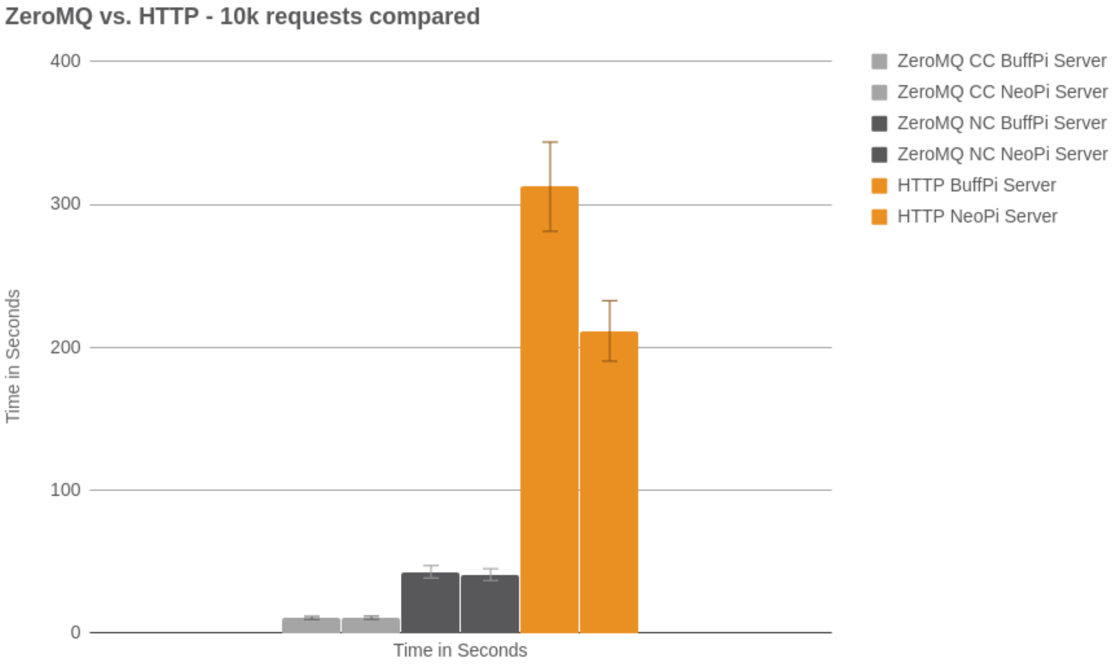


Figure 29: Performance comparison ZeroMQ vs. HTTP

In addition to that, HTTP has a packet overhead that will consume more network bandwidth. ZeroMQ is well suitable for low latency connections and on the same side more lightweight. In the test ZeroMQ is always used in TCP mode. The creator of ZeroMQ promises that the IPC and especially the inter-thread protocol are much faster than TCP.[80, cf.] But still for the TCP mode, the results are self-evidently.

6.2.4 Performance MQTT

The last test setup was used to test the MQTT performance. Due to the fact that MQTT needs a broker to be executed, a third device the *Laptop* was added. Each test was setup in three different ways. The Laptop, the BuffPi and the NeoPi, each of them was the broker once. The BuffPi or the NeoPi were always the publisher and the receiver. Furthermore MQTT has three different types to be executed, which is called the QoS level. The QoS was explained in section 2.4.1. The initial assumption was that each level has a different performance. Also in this setup each request was sent 10.000 times and in ten iterations. All results are shown in section A.1.2. Figure 30 shows the average time for each QoS level, where BuffPi was the broker as well as the publisher and the NeoPi was the consumer.

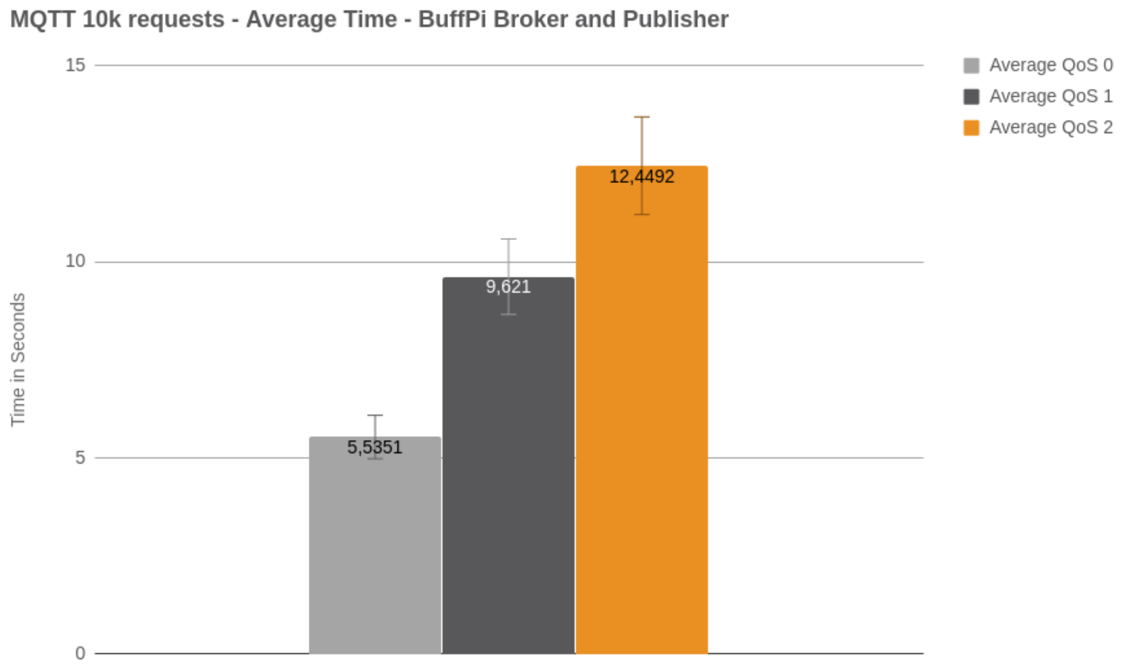


Figure 30: MQTT average time for all QoS levels

As expected the QoS level performance differ significantly. The difference between QoS 0 and QoS 2 is more than two and a half of the time, used for sending out the 10.000 requests. Therefore the QoS level should be consciously used in the required use case. Due to the fact that the MQTT system is used for the node discovery, a lower QoS level is suitable. QoS level 1 is recommended for that application. And even compared to the HTTP connection, MQTT is much faster and more reliable for the node discovery.

On the other side, the device that is executing the broker, is more crucial. Figure 31 shows the difference between the three devices.

The Laptop that has the most powerful hardware, is also faster by sending out the messages. NeoPi the weakest device in the test setup took nearly twice as much as the Laptop. Compared to HTTP even the NeoPi is much faster, but for a huge cluster the broker should be as powerful as possible. Therefore, if the broker should be located inside the cluster, then the most powerful device should be used for that job. It would be optimal to use a dedicated node for that job, to not divide the available resources with another application.

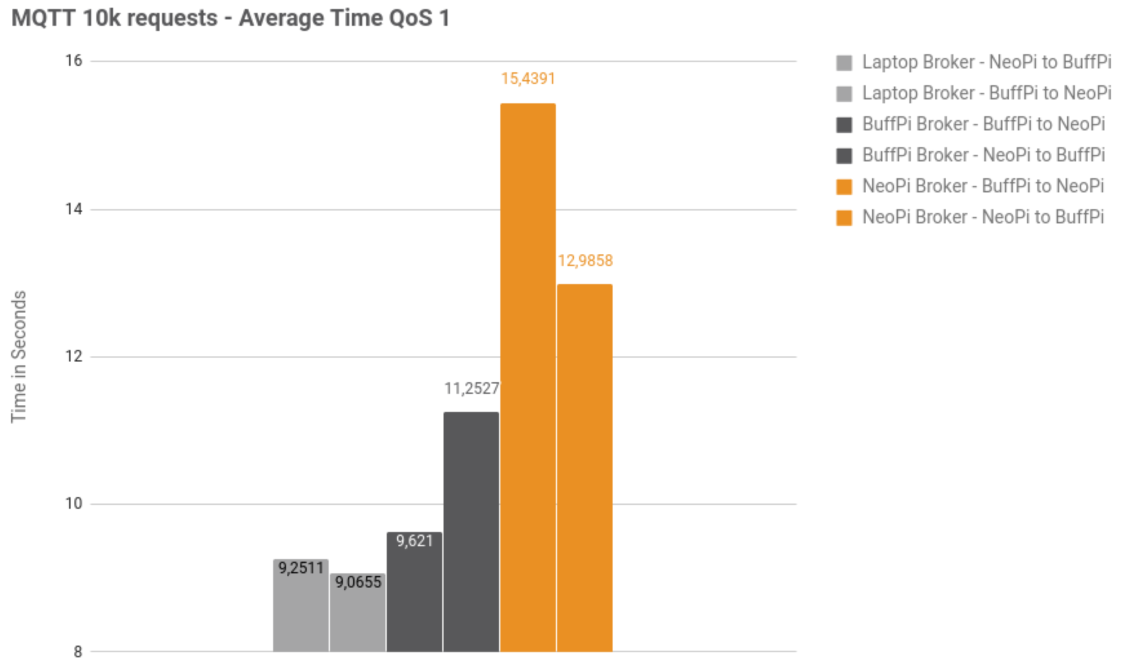


Figure 31: MQTT average time for QoS 1

One important learning about MQTT is, in QoS level 1 and 2 messages have to be persisted in a local cache. This took some time and it is possible that if the receive message on the broker exceeds the message queue. If this happens, all following messages from that node will be dropped. To fix that issue the total amount of messages in the queue must be extended. Therefore, the Mosquitto config has to be configured by adding the following lines:

```
max_inflight_messages 1000
max_queued_messages 10000
```

Listing 6.1: Mosquitto config modification to fix the messages dropped issue

The first parameter is adjusting the maximum number of "messages that can be in the process of being transmitted simultaneously"[81]. The *max_queued_messages* modifies the maximum number of messages, that can be hold in the queue.[81, cf.] This number had to be increased in the tests, because the scripts send out a huge amount of publish calls in a very short time. In general these numbers had to be adjusted specifically for each use case. Furthermore, these configs are only important if a QoS level 1 or 2 is used, because level 0 does not persist any messages.

CHAPTER 7

Conclusion

This final chapter sums up the thesis as well as the created prototype called Motey. The first section outlines the initial idea behind the project and reproduces the different work stages. Issues that remained unsolved are described in the last section as well as an outlook for the Motey project is shown.

7.1 Summary

Objective of this thesis was the "Design and Development of a Fog Service Orchestration Engine for Smart Factories". Therefore, a prototype for such an orchestration engine, called Motey, was created. The application was created in cooperation with the Fraunhofer-Institut für Offene Kommunikationssysteme (FOKUS). The main idea of the project was worked out in an iterative process, together with supervisors of the FOKUS. In several meetings the objectives were analyzed and elaborated. Motey is designed to be executed on each fog node and is able to instantiate an inter-node connection. Each node has knowledge of all the other nodes and can communicate with them right at the moment they have to.

The application is also able to handle different so called capabilities. That are functional and non-functional requirements a node can fulfill. These capabilities are used to deploy images, for example NFs, to one or more nodes. The node that receives a deployment schema, checks the requirements of the contained images and deploys them on the same node, if it is possible, or on other nodes that are able to deploy them. Labels can be configured from within the node itself or from any external application. Finally, an image will be started via the related virtualization tool. In case of the prototype and as a primary requirement of this project, Docker is the default engine. For the inter-node communication the ZeroMQ and MQTT protocols are used. They are much faster than the pretty famous HTTP protocol, as shown in the performance evaluation section 6.2. A HTTP REST API is also available, to enable compatibility with well known cloud orchestration engines like Open Baton or OpenStack.
The whole application is well documented and tested. Each component has a related unit test and the documentation is available online at `https://neoklosch.github.io/Motey/`. A CI pipeline is used to automatically build each new version, to test it and finally to deploy it as a Docker image to the Docker Hub. This is useful to reduce the overhead of repeating tasks,

to guarantee the correctness of the current version and also to have an up-to-date version available via the Docker infrastructure.
Biggest challenges in this project were, to create a fast and lightweight inter-connection between the nodes, that was achieved by the protocols of choice ZeroMQ and MQTT, as well as the abstraction of the underlying virtualization tools. The latter partly kept unsolved, because only a few virtualization tools supports the ARM CPU architecture. Whereas ARM is a common architecture on low power devices. Therefore, the integration of famous virtualization tools is not possible at the moment. Due to the fact that IoT becomes more and more important in the next years, hopefully also the tools will adjust respectively. However, Docker as the main virtualization tool in this thesis, is available for ARM CPUs yet and also implemented in Motey. As a lightweight solution for virtualization, this is a good starting point so far.

7.2 Outlook

Motey is a well developed and pretty solid basis for further development. As in every bigger project, Motey has significant room for improvements. For example, the autonomous behavior can be improved a lot. One possibility could be the ability to respond to real-time requirements, even in case of the absence of the centralized cloud level. One of the nodes could become a master node. These nodes then can act as a delegate for the centralized level. It can track the state of the other nodes and can react to unexpected issues. Another possibility could be an improved access control or rights management. Some nodes could have more rights than others. Therefore, a hierarchy could be created and sensitive images or images that handle sensitive data, could be easily identified. For security reasons, that becomes very important.

The node discovery can be improved by making the MQTT broker replaceable. This could be achieved by having an logic, that moves the broker over to a node that is up and running if the broker disappears. Such a behavior makes the system reliable and again more autonomously. Finally, the integration of third party applications can be ensured. For example, the integration of Open Baton, as the centralized orchestrator, could be realized. Also node specific tools, like a hardware detection engine, that can deploy images or at least add new capabilities to Motey based on the connected hardware, are imaginable.

Motey can be seen as a good starting point for a complex environment, made for Fog Computing. It is easy to extend, is lightweight during execution and networking. Due to the fact that it can be deployed via Docker out of the box, it is easy to test and the documentation and the unit test should help to understand the fundamentals pretty easy. At the end, the created project successfully reveal, that the developed concept works out pretty well in a prototypical quality. The result can thus be considered as solid basis for further development.

APPENDIX A

A Test Results

A.1 Performance Test Results

A.1.1 HTTP

Iteration	Type	Server	Client	Time	Amount	In Millisec	In Seconds	Average in Millisec	Average in Sec
1	HTTP	BuffPi	NeoPi	0:05:12.221162	10.000	312221	312,221	312421,8	312,4218
2	HTTP	BuffPi	NeoPi	0:05:12.578003	10.000	312578	312,578		
3	HTTP	BuffPi	NeoPi	0:05:13.298726	10.000	313298	313,298		
4	HTTP	BuffPi	NeoPi	0:05:10.225036	10.000	310225	310,225		
5	HTTP	BuffPi	NeoPi	0:05:11.273922	10.000	311273	311,273		
6	HTTP	BuffPi	NeoPi	0:05:13.983336	10.000	313983	313,983		
7	HTTP	BuffPi	NeoPi	0:05:11.431619	10.000	311431	311,431		
8	HTTP	BuffPi	NeoPi	0:05:13.558595	10.000	313558	313,558		
9	HTTP	BuffPi	NeoPi	0:05:11.762938	10.000	311762	311,762		
10	HTTP	BuffPi	NeoPi	0:05:13.889453	10.000	313889	313,889		
1	HTTP	NeoPi	BuffPi	0:03:29.268649	10.000	209268	209,268	211537,8	211,5378
2	HTTP	NeoPi	BuffPi	0:03:30.187172	10.000	210187	210,187		
3	HTTP	NeoPi	BuffPi	0:03:31.454076	10.000	211454	211,454		
4	HTTP	NeoPi	BuffPi	0:03:34.105147	10.000	214105	214,105		
5	HTTP	NeoPi	BuffPi	0:03:32.227625	10.000	212227	212,227		
6	HTTP	NeoPi	BuffPi	0:03:32.002913	10.000	212002	212,002		
7	HTTP	NeoPi	BuffPi	0:03:31.309765	10.000	211309	211,309		
8	HTTP	NeoPi	BuffPi	0:03:32.071410	10.000	212071	212,071		
9	HTTP	NeoPi	BuffPi	0:03:30.978702	10.000	210978	210,978		
10	HTTP	NeoPi	BuffPi	0:03:31.777458	10.000	211777	211,777		

A.1.2 MQTT

Iteration	Type	Server	Client	Client Receive	Time	Amount	QoS	In Millisec	In Seconds	Average in Millisec	Average in Sec
1	MQTT	Laptop	NeoPi	BuffPi	0:00:04.994859	10.000	0	4994	4,994	5461,9	5,4619
2	MQTT	Laptop	NeoPi	BuffPi	0:00:04.783035	10.000	0	4783	4,783		
3	MQTT	Laptop	NeoPi	BuffPi	0:00:05.927165	10.000	0	5927	5,927		
4	MQTT	Laptop	NeoPi	BuffPi	0:00:05.733730	10.000	0	5733	5,733		
5	MQTT	Laptop	NeoPi	BuffPi	0:00:05.634835	10.000	0	5634	5,634		
6	MQTT	Laptop	NeoPi	BuffPi	0:00:05.907534	10.000	0	5907	5,907		
7	MQTT	Laptop	NeoPi	BuffPi	0:00:05.420570	10.000	0	5420	5,42		
8	MQTT	Laptop	NeoPi	BuffPi	0:00:05.830361	10.000	0	5830	5,83		
9	MQTT	Laptop	NeoPi	BuffPi	0:00:05.550823	10.000	0	5550	5,55		
10	MQTT	Laptop	NeoPi	BuffPi	0:00:04.841393	10.000	0	4841	4,841		
1	MQTT	Laptop	NeoPi	BuffPi	0:00:09.430188	10.000	1	9430	9,43	9251,1	9,2511
2	MQTT	Laptop	NeoPi	BuffPi	0:00:09.118192	10.000	1	9118	9,118		
3	MQTT	Laptop	NeoPi	BuffPi	0:00:09.341204	10.000	1	9341	9,341		
4	MQTT	Laptop	NeoPi	BuffPi	0:00:09.179332	10.000	1	9179	9,179		
5	MQTT	Laptop	NeoPi	BuffPi	0:00:09.382317	10.000	1	9382	9,382		
6	MQTT	Laptop	NeoPi	BuffPi	0:00:09.146739	10.000	1	9146	9,146		
7	MQTT	Laptop	NeoPi	BuffPi	0:00:09.402362	10.000	1	9402	9,402		
8	MQTT	Laptop	NeoPi	BuffPi	0:00:09.249343	10.000	1	9249	9,249		
9	MQTT	Laptop	NeoPi	BuffPi	0:00:09.169960	10.000	1	9169	9,169		
10	MQTT	Laptop	NeoPi	BuffPi	0:00:09.095161	10.000	1	9095	9,095		
1	MQTT	Laptop	NeoPi	BuffPi	0:00:12.870583	10.000	2	12870	12,87	13380,4	13,3804
2	MQTT	Laptop	NeoPi	BuffPi	0:00:12.853302	10.000	2	12853	12,853		
3	MQTT	Laptop	NeoPi	BuffPi	0:00:13.009984	10.000	2	13009	13,009		
4	MQTT	Laptop	NeoPi	BuffPi	0:00:15.305975	10.000	2	15305	15,305		
5	MQTT	Laptop	NeoPi	BuffPi	0:00:12.951781	10.000	2	12951	12,951		
6	MQTT	Laptop	NeoPi	BuffPi	0:00:13.373944	10.000	2	13373	13,373		
7	MQTT	Laptop	NeoPi	BuffPi	0:00:13.686687	10.000	2	13686	13,686		
8	MQTT	Laptop	NeoPi	BuffPi	0:00:12.798258	10.000	2	12798	12,798		
9	MQTT	Laptop	NeoPi	BuffPi	0:00:12.961487	10.000	2	12961	12,961		
10	MQTT	Laptop	NeoPi	BuffPi	0:00:13.998203	10.000	2	13998	13,998		
Durchlauf	Testtyp	Server	Client	Client Receive	Zeit	Anzahl	QoS				
1	MQTT	Laptop	BuffPi	NeoPi	0:00:05.509739	10.000	0	5509	5,509	5543,8	5,5438
2	MQTT	Laptop	BuffPi	NeoPi	0:00:05.508638	10.000	0	5508	5,508		
3	MQTT	Laptop	BuffPi	NeoPi	0:00:05.587754	10.000	0	5587	5,587		

4	MQTT	Laptop	BuffPi	NeoPi	0:00:05.603266	10.000	0	5603	5,603		
5	MQTT	Laptop	BuffPi	NeoPi	0:00:05.458693	10.000	0	5458	5,458		
6	MQTT	Laptop	BuffPi	NeoPi	0:00:05.550002	10.000	0	5550	5,55		
7	MQTT	Laptop	BuffPi	NeoPi	0:00:05.510637	10.000	0	5510	5,51		
8	MQTT	Laptop	BuffPi	NeoPi	0:00:05.549627	10.000	0	5549	5,549		
9	MQTT	Laptop	BuffPi	NeoPi	0:00:05.602222	10.000	0	5602	5,602		
10	MQTT	Laptop	BuffPi	NeoPi	0:00:05.562656	10.000	0	5562	5,562		
1	MQTT	Laptop	BuffPi	NeoPi	0:00:08.902878	10.000	1	8902	8,902	9065,5	9,0655
2	MQTT	Laptop	BuffPi	NeoPi	0:00:09.123871	10.000	1	9123	9,123		
3	MQTT	Laptop	BuffPi	NeoPi	0:00:09.015040	10.000	1	9015	9,015		
4	MQTT	Laptop	BuffPi	NeoPi	0:00:08.916805	10.000	1	8916	8,916		
5	MQTT	Laptop	BuffPi	NeoPi	0:00:09.178874	10.000	1	9178	9,178		
6	MQTT	Laptop	BuffPi	NeoPi	0:00:08.892108	10.000	1	8892	8,892		
7	MQTT	Laptop	BuffPi	NeoPi	0:00:09.103947	10.000	1	9103	9,103		
8	MQTT	Laptop	BuffPi	NeoPi	0:00:08.920237	10.000	1	8920	8,92		
9	MQTT	Laptop	BuffPi	NeoPi	0:00:09.055381	10.000	1	9055	9,055		
10	MQTT	Laptop	BuffPi	NeoPi	0:00:09.551242	10.000	1	9551	9,551		
1	MQTT	Laptop	BuffPi	NeoPi	0:00:11.163587	10.000	2	11163	11,163	11474,5	11,4745
2	MQTT	Laptop	BuffPi	NeoPi	0:00:10.966241	10.000	2	10966	10,966		
3	MQTT	Laptop	BuffPi	NeoPi	0:00:11.916616	10.000	2	11916	11,916		
4	MQTT	Laptop	BuffPi	NeoPi	0:00:11.196631	10.000	2	11196	11,196		
5	MQTT	Laptop	BuffPi	NeoPi	0:00:11.065672	10.000	2	11065	11,065		
6	MQTT	Laptop	BuffPi	NeoPi	0:00:11.335435	10.000	2	11335	11,335		
7	MQTT	Laptop	BuffPi	NeoPi	0:00:11.916999	10.000	2	11916	11,916		
8	MQTT	Laptop	BuffPi	NeoPi	0:00:11.892799	10.000	2	11892	11,892		
9	MQTT	Laptop	BuffPi	NeoPi	0:00:11.688006	10.000	2	11688	11,688		
10	MQTT	Laptop	BuffPi	NeoPi	0:00:11.608120	10.000	2	11608	11,608		
Durchlauf	Testtyp	Server	Client	Client Receive	Zeit	Anzahl	QoS				
1	MQTT	BuffPi	BuffPi	NeoPi	0:00:05.490377	10.000	0	5490	5,49	5535,1	5,5351
2	MQTT	BuffPi	BuffPi	NeoPi	0:00:05.556821	10.000	0	5556	5,556		
3	MQTT	BuffPi	BuffPi	NeoPi	0:00:05.552181	10.000	0	5552	5,552		
4	MQTT	BuffPi	BuffPi	NeoPi	0:00:05.538765	10.000	0	5538	5,538		
5	MQTT	BuffPi	BuffPi	NeoPi	0:00:05.513784	10.000	0	5513	5,513		
6	MQTT	BuffPi	BuffPi	NeoPi	0:00:05.562727	10.000	0	5562	5,562		
7	MQTT	BuffPi	BuffPi	NeoPi	0:00:05.565102	10.000	0	5565	5,565		

8	MQTT	BuffPi	BuffPi	NeoPi	0:00:05.613164	10.000	0	5613	5,613		
9	MQTT	BuffPi	BuffPi	NeoPi	0:00:05.510732	10.000	0	5510	5,51		
10	MQTT	BuffPi	BuffPi	NeoPi	0:00:05.452501	10.000	0	5452	5,452		
1	MQTT	BuffPi	BuffPi	NeoPi	0:00:09.543109	10.000	1	9543	9,543	9621	9,621
2	MQTT	BuffPi	BuffPi	NeoPi	0:00:10.500300	10.000	1	10500	10,5		
3	MQTT	BuffPi	BuffPi	NeoPi	0:00:09.422465	10.000	1	9422	9,422		
4	MQTT	BuffPi	BuffPi	NeoPi	0:00:09.290589	10.000	1	9290	9,29		
5	MQTT	BuffPi	BuffPi	NeoPi	0:00:09.364733	10.000	1	9364	9,364		
6	MQTT	BuffPi	BuffPi	NeoPi	0:00:09.318762	10.000	1	9318	9,318		
7	MQTT	BuffPi	BuffPi	NeoPi	0:00:09.623347	10.000	1	9623	9,623		
8	MQTT	BuffPi	BuffPi	NeoPi	0:00:09.403539	10.000	1	9403	9,403		
9	MQTT	BuffPi	BuffPi	NeoPi	0:00:09.384825	10.000	1	9384	9,384		
10	MQTT	BuffPi	BuffPi	NeoPi	0:00:10.363415	10.000	1	10363	10,363		
1	MQTT	BuffPi	BuffPi	NeoPi	0:00:11.578684	10.000	2	11578	11,578	12449,2	12,4492
2	MQTT	BuffPi	BuffPi	NeoPi	0:00:11.764746	10.000	2	11764	11,764		
3	MQTT	BuffPi	BuffPi	NeoPi	0:00:11.636049	10.000	2	11636	11,636		
4	MQTT	BuffPi	BuffPi	NeoPi	0:00:11.826162	10.000	2	11826	11,826		
5	MQTT	BuffPi	BuffPi	NeoPi	0:00:14.028714	10.000	2	14028	14,028		
6	MQTT	BuffPi	BuffPi	NeoPi	0:00:13.949237	10.000	2	13949	13,949		
7	MQTT	BuffPi	BuffPi	NeoPi	0:00:14.023610	10.000	2	14023	14,023		
8	MQTT	BuffPi	BuffPi	NeoPi	0:00:11.916026	10.000	2	11916	11,916		
9	MQTT	BuffPi	BuffPi	NeoPi	0:00:11.925905	10.000	2	11925	11,925		
10	MQTT	BuffPi	BuffPi	NeoPi	0:00:11.847692	10.000	2	11847	11,847		
Durchlauf	Testtyp	Server	Client	Client Receive	Zeit	Anzahl	QoS				
1	MQTT	BuffPi	NeoPi	BuffPi	0:00:06.885060	10.000	0	6885	6,885	7490,3	7,4903
2	MQTT	BuffPi	NeoPi	BuffPi	0:00:09.281281	10.000	0	9281	9,281		
3	MQTT	BuffPi	NeoPi	BuffPi	0:00:07.155109	10.000	0	7155	7,155		
4	MQTT	BuffPi	NeoPi	BuffPi	0:00:07.941969	10.000	0	7941	7,941		
5	MQTT	BuffPi	NeoPi	BuffPi	0:00:06.992707	10.000	0	6992	6,992		
6	MQTT	BuffPi	NeoPi	BuffPi	0:00:08.923308	10.000	0	8923	8,923		
7	MQTT	BuffPi	NeoPi	BuffPi	0:00:06.618326	10.000	0	6618	6,618		
8	MQTT	BuffPi	NeoPi	BuffPi	0:00:06.613919	10.000	0	6613	6,613		
9	MQTT	BuffPi	NeoPi	BuffPi	0:00:08.518265	10.000	0	8518	8,518		
10	MQTT	BuffPi	NeoPi	BuffPi	0:00:05.977240	10.000	0	5977	5,977		

1	MQTT	BuffPi	NeoPi	BuffPi	0:00:10.470269	10.000	1	10470	10,47	11252,7	11,2527
2	MQTT	BuffPi	NeoPi	BuffPi	0:00:10.237973	10.000	1	10237	10,237		
3	MQTT	BuffPi	NeoPi	BuffPi	0:00:10.926891	10.000	1	10926	10,926		
4	MQTT	BuffPi	NeoPi	BuffPi	0:00:10.871666	10.000	1	10871	10,871		
5	MQTT	BuffPi	NeoPi	BuffPi	0:00:10.505045	10.000	1	10505	10,505		
6	MQTT	BuffPi	NeoPi	BuffPi	0:00:10.060886	10.000	1	10060	10,06		
7	MQTT	BuffPi	NeoPi	BuffPi	0:00:12.882999	10.000	1	12882	12,882		
8	MQTT	BuffPi	NeoPi	BuffPi	0:00:12.704596	10.000	1	12704	12,704		
9	MQTT	BuffPi	NeoPi	BuffPi	0:00:10.850831	10.000	1	10850	10,85		
10	MQTT	BuffPi	NeoPi	BuffPi	0:00:13.022687	10.000	1	13022	13,022		
1	MQTT	BuffPi	NeoPi	BuffPi	0:00:13.675367	10.000	2	13675	13,675	13968,4	13,9684
2	MQTT	BuffPi	NeoPi	BuffPi	0:00:13.581732	10.000	2	13581	13,581		
3	MQTT	BuffPi	NeoPi	BuffPi	0:00:13.660716	10.000	2	13660	13,66		
4	MQTT	BuffPi	NeoPi	BuffPi	0:00:13.633806	10.000	2	13633	13,633		
5	MQTT	BuffPi	NeoPi	BuffPi	0:00:15.078343	10.000	2	15078	15,078		
6	MQTT	BuffPi	NeoPi	BuffPi	0:00:15.590062	10.000	2	15590	15,59		
7	MQTT	BuffPi	NeoPi	BuffPi	0:00:13.625364	10.000	2	13625	13,625		
8	MQTT	BuffPi	NeoPi	BuffPi	0:00:13.519778	10.000	2	13519	13,519		
9	MQTT	BuffPi	NeoPi	BuffPi	0:00:13.943205	10.000	2	13943	13,943		
10	MQTT	BuffPi	NeoPi	BuffPi	0:00:13.380698	10.000	2	13380	13,38		
Durchlauf	Testtyp	Server	Client	Client Receive	Zeit	Anzahl	QoS				
1	MQTT	NeoPi	BuffPi	NeoPi	0:00:05.902972	10.000	0	5902	5,902	6288,2	6,2882
2	MQTT	NeoPi	BuffPi	NeoPi	0:00:06.094702	10.000	0	6094	6,094		
3	MQTT	NeoPi	BuffPi	NeoPi	0:00:06.041093	10.000	0	6041	6,041		
4	MQTT	NeoPi	BuffPi	NeoPi	0:00:05.951912	10.000	0	5951	5,951		
5	MQTT	NeoPi	BuffPi	NeoPi	0:00:06.428904	10.000	0	6428	6,428		
6	MQTT	NeoPi	BuffPi	NeoPi	0:00:06.630405	10.000	0	6630	6,63		
7	MQTT	NeoPi	BuffPi	NeoPi	0:00:06.536694	10.000	0	6536	6,536		
8	MQTT	NeoPi	BuffPi	NeoPi	0:00:06.399787	10.000	0	6399	6,399		
9	MQTT	NeoPi	BuffPi	NeoPi	0:00:06.441628	10.000	0	6441	6,441		
10	MQTT	NeoPi	BuffPi	NeoPi	0:00:06.460394	10.000	0	6460	6,46		
1	MQTT	NeoPi	BuffPi	NeoPi	0:00:14.428439	10.000	1	14428	14,428	15439,1	15,4391
2	MQTT	NeoPi	BuffPi	NeoPi	0:00:14.923507	10.000	1	14923	14,923		
3	MQTT	NeoPi	BuffPi	NeoPi	0:00:14.340367	10.000	1	14340	14,34		
4	MQTT	NeoPi	BuffPi	NeoPi	0:00:17.555054	10.000	1	17555	17,555		

5	MQTT	NeoPi	BuffPi	NeoPi	0:00:17.881900	10.000	1	17881	17,881		
6	MQTT	NeoPi	BuffPi	NeoPi	0:00:14.000897	10.000	1	14000	14		
7	MQTT	NeoPi	BuffPi	NeoPi	0:00:14.227708	10.000	1	14227	14,227		
8	MQTT	NeoPi	BuffPi	NeoPi	0:00:15.132524	10.000	1	15132	15,132		
9	MQTT	NeoPi	BuffPi	NeoPi	0:00:14.216478	10.000	1	14216	14,216		
10	MQTT	NeoPi	BuffPi	NeoPi	0:00:17.689077	10.000	1	17689	17,689		
1	MQTT	NeoPi	BuffPi	NeoPi	0:00:16.870064	10.000	2	16870	16,87	16818,3	16,8183
2	MQTT	NeoPi	BuffPi	NeoPi	0:00:15.870926	10.000	2	15870	15,87		
3	MQTT	NeoPi	BuffPi	NeoPi	0:00:15.793400	10.000	2	15793	15,793		
4	MQTT	NeoPi	BuffPi	NeoPi	0:00:16.722951	10.000	2	16722	16,722		
5	MQTT	NeoPi	BuffPi	NeoPi	0:00:17.165282	10.000	2	17165	17,165		
6	MQTT	NeoPi	BuffPi	NeoPi	0:00:17.026038	10.000	2	17026	17,026		
7	MQTT	NeoPi	BuffPi	NeoPi	0:00:17.369432	10.000	2	17369	17,369		
8	MQTT	NeoPi	BuffPi	NeoPi	0:00:16.814805	10.000	2	16814	16,814		
9	MQTT	NeoPi	BuffPi	NeoPi	0:00:17.370874	10.000	2	17370	17,37		
10	MQTT	NeoPi	BuffPi	NeoPi	0:00:17.184623	10.000	2	17184	17,184		
Durchlauf	Testtyp	Server	Client	Client Receive	Zeit	Anzahl	QoS				
1	MQTT	NeoPi	NeoPi	BuffPi	0:00:07.630934	10.000	0	7630	7,63	9280,1	9,2801
2	MQTT	NeoPi	NeoPi	BuffPi	0:00:08.877320	10.000	0	8877	8,877		
3	MQTT	NeoPi	NeoPi	BuffPi	0:00:10.610945	10.000	0	10610	10,61		
4	MQTT	NeoPi	NeoPi	BuffPi	0:00:08.964234	10.000	0	8964	8,964		
5	MQTT	NeoPi	NeoPi	BuffPi	0:00:09.164747	10.000	0	9164	9,164		
6	MQTT	NeoPi	NeoPi	BuffPi	0:00:10.115942	10.000	0	10115	10,115		
7	MQTT	NeoPi	NeoPi	BuffPi	0:00:09.199618	10.000	0	9199	9,199		
8	MQTT	NeoPi	NeoPi	BuffPi	0:00:08.254946	10.000	0	8254	8,254		
9	MQTT	NeoPi	NeoPi	BuffPi	0:00:10.678273	10.000	0	10678	10,678		
10	MQTT	NeoPi	NeoPi	BuffPi	0:00:09.310548	10.000	0	9310	9,31		
1	MQTT	NeoPi	NeoPi	BuffPi	0:00:13.037954	10.000	1	13037	13,037	12985,8	12,9858
2	MQTT	NeoPi	NeoPi	BuffPi	0:00:12.847555	10.000	1	12847	12,847		
3	MQTT	NeoPi	NeoPi	BuffPi	0:00:12.610719	10.000	1	12610	12,61		
4	MQTT	NeoPi	NeoPi	BuffPi	0:00:13.443811	10.000	1	13443	13,443		
5	MQTT	NeoPi	NeoPi	BuffPi	0:00:12.904153	10.000	1	12904	12,904		
6	MQTT	NeoPi	NeoPi	BuffPi	0:00:12.518878	10.000	1	12518	12,518		
7	MQTT	NeoPi	NeoPi	BuffPi	0:00:13.328668	10.000	1	13328	13,328		
8	MQTT	NeoPi	NeoPi	BuffPi	0:00:13.075662	10.000	1	13075	13,075		

9	MQTT	NeoPi	NeoPi	BuffPi	0:00:12.675193	10.000	1	12675	12,675		
10	MQTT	NeoPi	NeoPi	BuffPi	0:00:13.421942	10.000	1	13421	13,421		
1	MQTT	NeoPi	NeoPi	BuffPi	0:00:19.190555	10.000	2	19190	19,19	17671,5	17,6715
2	MQTT	NeoPi	NeoPi	BuffPi	0:00:18.298763	10.000	2	18298	18,298		
3	MQTT	NeoPi	NeoPi	BuffPi	0:00:16.647760	10.000	2	16647	16,647		
4	MQTT	NeoPi	NeoPi	BuffPi	0:00:17.980774	10.000	2	17980	17,98		
5	MQTT	NeoPi	NeoPi	BuffPi	0:00:17.552975	10.000	2	17552	17,552		
6	MQTT	NeoPi	NeoPi	BuffPi	0:00:17.600359	10.000	2	17600	17,6		
7	MQTT	NeoPi	NeoPi	BuffPi	0:00:17.284753	10.000	2	17284	17,284		
8	MQTT	NeoPi	NeoPi	BuffPi	0:00:17.333969	10.000	2	17333	17,333		
9	MQTT	NeoPi	NeoPi	BuffPi	0:00:17.702040	10.000	2	17702	17,702		
10	MQTT	NeoPi	NeoPi	BuffPi	0:00:17.129283	10.000	2	17129	17,129		

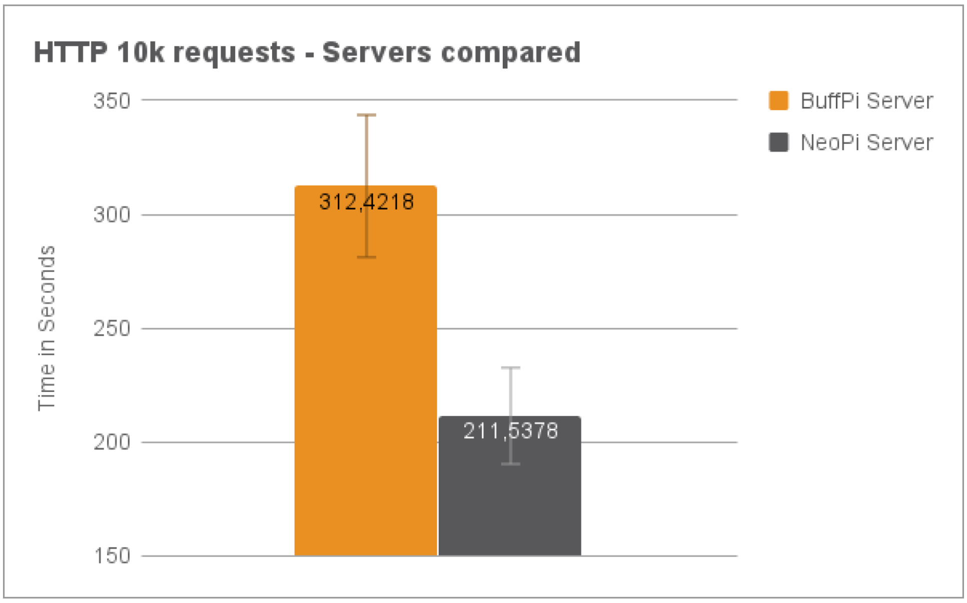
A.1.3 ZeroMQ

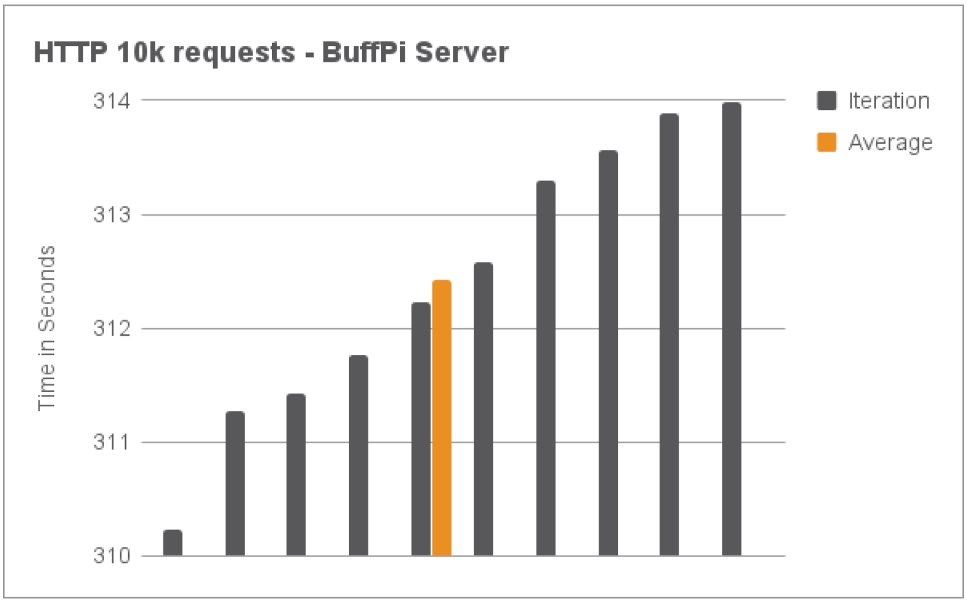
Iteration	Type	Server	Client	Time	Amount	In Millisec	In Seconds	Average in Millisec	Average in Sec
1	ZeroMQ CC	BuffPi	NeoPi	0:00:10.661297	10.000	10661	10,661	10646,7	10,6467
2	ZeroMQ CC	BuffPi	NeoPi	0:00:10.548843	10.000	10548	10,548		
3	ZeroMQ CC	BuffPi	NeoPi	0:00:10.734878	10.000	10734	10,734		
4	ZeroMQ CC	BuffPi	NeoPi	0:00:10.683753	10.000	10683	10,683		
5	ZeroMQ CC	BuffPi	NeoPi	0:00:10.646455	10.000	10646	10,646		
6	ZeroMQ CC	BuffPi	NeoPi	0:00:10.620864	10.000	10620	10,62		
7	ZeroMQ CC	BuffPi	NeoPi	0:00:10.693436	10.000	10693	10,693		
8	ZeroMQ CC	BuffPi	NeoPi	0:00:10.648262	10.000	10648	10,648		
9	ZeroMQ CC	BuffPi	NeoPi	0:00:10.614209	10.000	10614	10,614		
10	ZeroMQ CC	BuffPi	NeoPi	0:00:10.620353	10.000	10620	10,62		
1	ZeroMQ CC	NeoPi	BuffPi	0:00:10.758461	10.000	10758	10,758	10758,6	10,7586
2	ZeroMQ CC	NeoPi	BuffPi	0:00:10.812967	10.000	10812	10,812		
3	ZeroMQ CC	NeoPi	BuffPi	0:00:10.763016	10.000	10763	10,763		
4	ZeroMQ CC	NeoPi	BuffPi	0:00:10.625833	10.000	10625	10,625		
5	ZeroMQ CC	NeoPi	BuffPi	0:00:10.864128	10.000	10864	10,864		
6	ZeroMQ CC	NeoPi	BuffPi	0:00:10.779625	10.000	10779	10,779		
7	ZeroMQ CC	NeoPi	BuffPi	0:00:10.812529	10.000	10812	10,812		
8	ZeroMQ CC	NeoPi	BuffPi	0:00:10.598294	10.000	10598	10,598		
9	ZeroMQ CC	NeoPi	BuffPi	0:00:10.764893	10.000	10764	10,764		
10	ZeroMQ CC	NeoPi	BuffPi	0:00:10.811703	10.000	10811	10,811		
1	ZeroMQ New Co	BuffPi	NeoPi	0:00:43.858489	10.000	43858	43,858	42965,1	42,9651
2	ZeroMQ New Co	BuffPi	NeoPi	0:00:41.477946	10.000	41477	41,477		
3	ZeroMQ New Co	BuffPi	NeoPi	0:00:42.493571	10.000	42493	42,493		
4	ZeroMQ New Co	BuffPi	NeoPi	0:00:43.120301	10.000	43120	43,12		
5	ZeroMQ New Co	BuffPi	NeoPi	0:00:42.092123	10.000	42092	42,092		
6	ZeroMQ New Co	BuffPi	NeoPi	0:00:42.405344	10.000	42405	42,405		
7	ZeroMQ New Co	BuffPi	NeoPi	0:00:42.322813	10.000	42322	42,322		
8	ZeroMQ New Co	BuffPi	NeoPi	0:00:44.491670	10.000	44491	44,491		
9	ZeroMQ New Co	BuffPi	NeoPi	0:00:43.345463	10.000	43345	43,345		

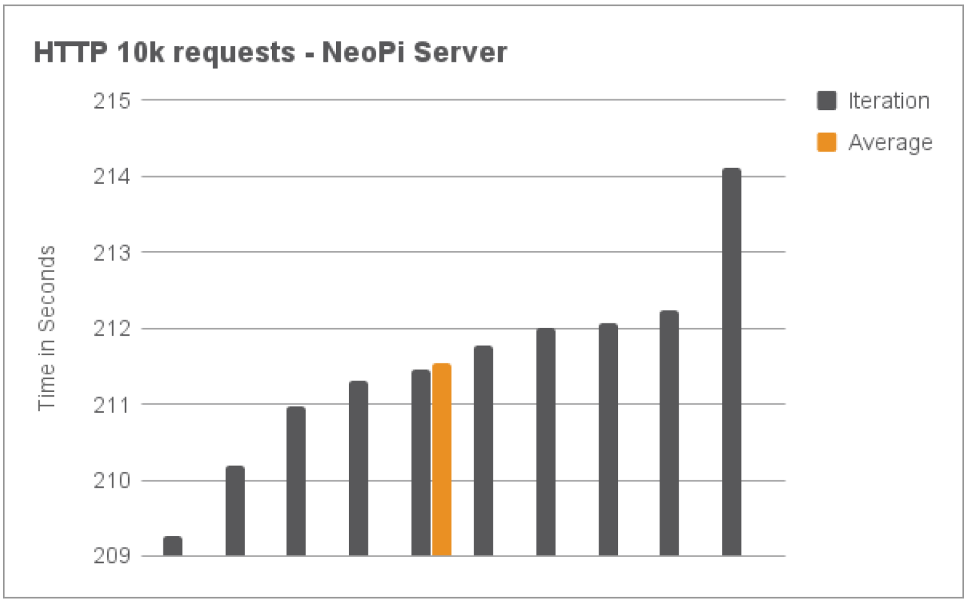
10	ZeroMQ New Co	BuffPi	NeoPi	0:00:44.048956	10.000	44048	44,048		
1	ZeroMQ New Co	NeoPi	BuffPi	0:00:40.479339	10.000	40479	40,479	41000,6	41,0006
2	ZeroMQ New Co	NeoPi	BuffPi	0:00:39.509120	10.000	39509	39,509		
3	ZeroMQ New Co	NeoPi	BuffPi	0:00:40.648344	10.000	40648	40,648		
4	ZeroMQ New Co	NeoPi	BuffPi	0:00:40.237376	10.000	40237	40,237		
5	ZeroMQ New Co	NeoPi	BuffPi	0:00:41.794357	10.000	41794	41,794		
6	ZeroMQ New Co	NeoPi	BuffPi	0:00:40.587207	10.000	40587	40,587		
7	ZeroMQ New Co	NeoPi	BuffPi	0:00:42.160765	10.000	42160	42,16		
8	ZeroMQ New Co	NeoPi	BuffPi	0:00:42.286994	10.000	42286	42,286		
9	ZeroMQ New Co	NeoPi	BuffPi	0:00:40.633044	10.000	40633	40,633		
10	ZeroMQ New Co	NeoPi	BuffPi	0:00:41.673469	10.000	41673	41,673		

A.2 Performance Test Graphs

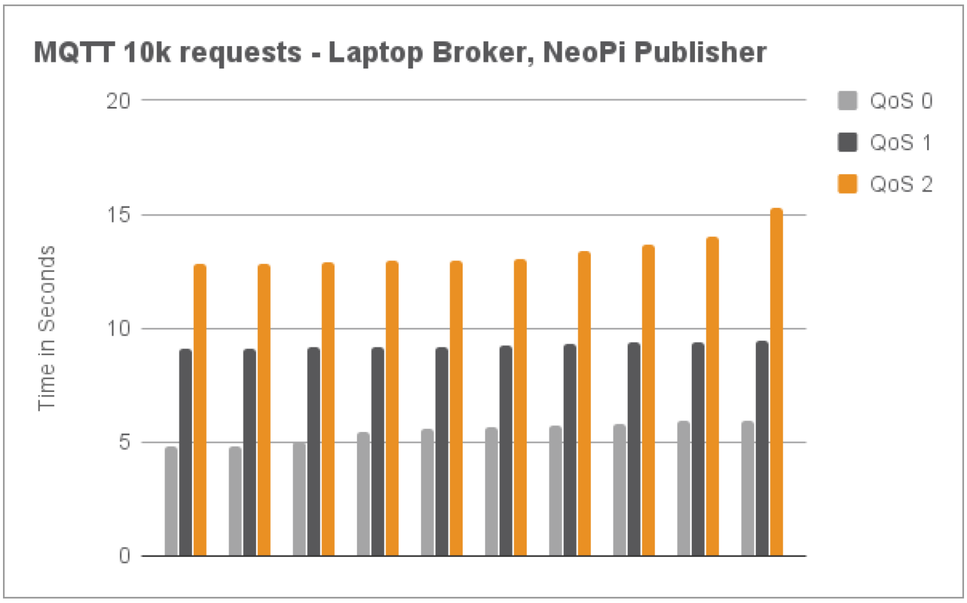
A.2.1 HTTP Graphs

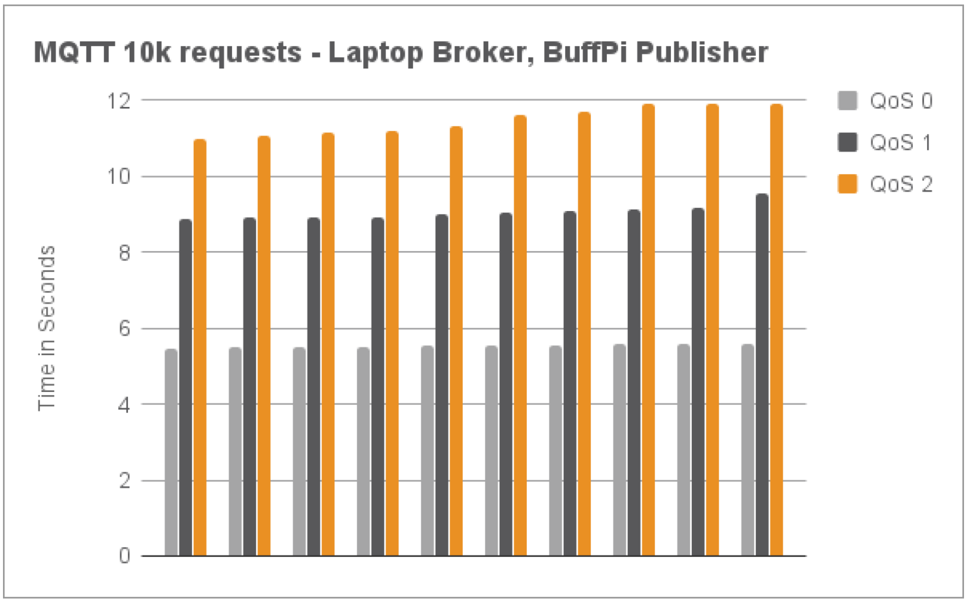


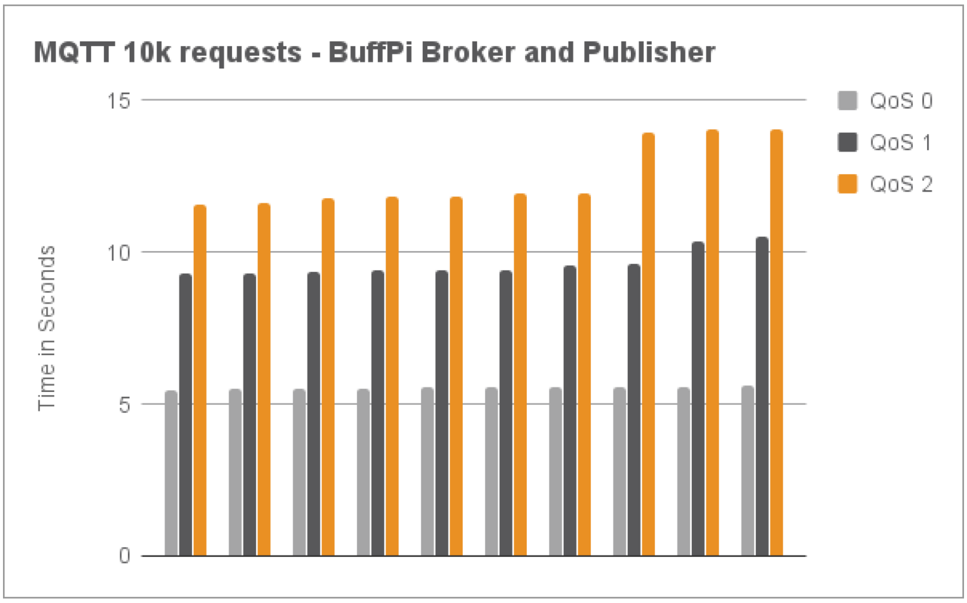


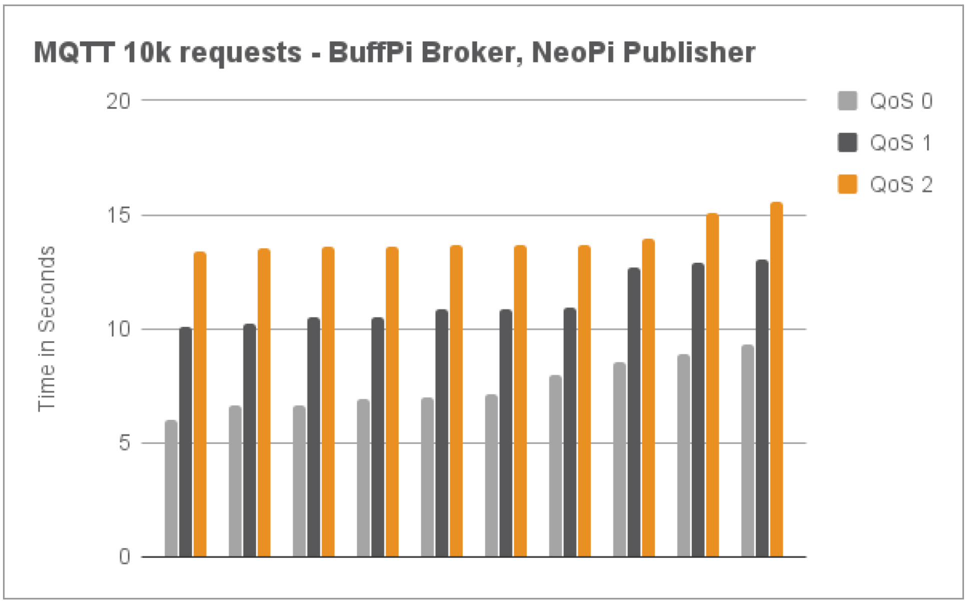

HTTP 10k requests - NeoPi Server
Time in Seconds
215
214
213
212
211
210
209
Iteration
Average

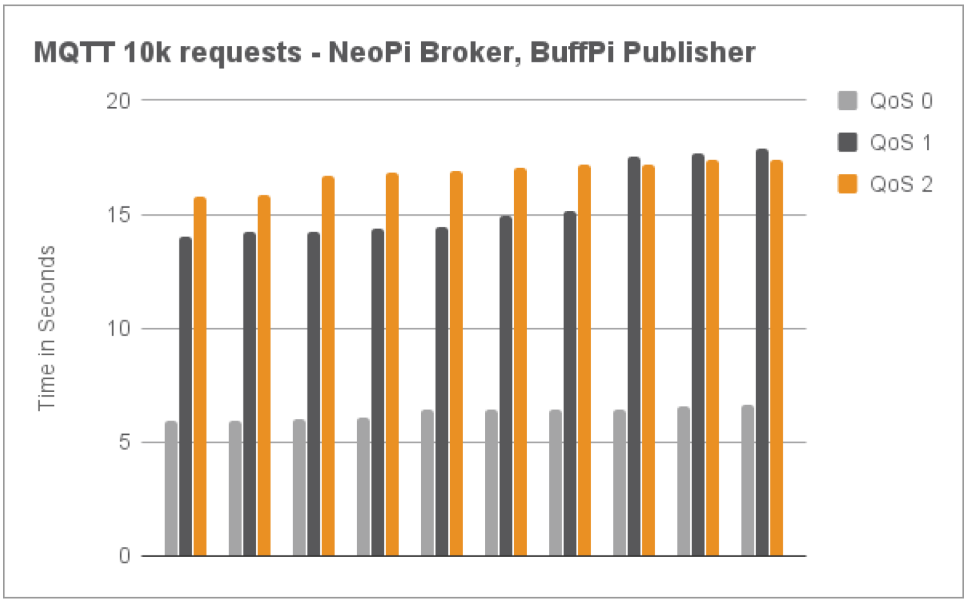
A.2.2 MQTT Graphs

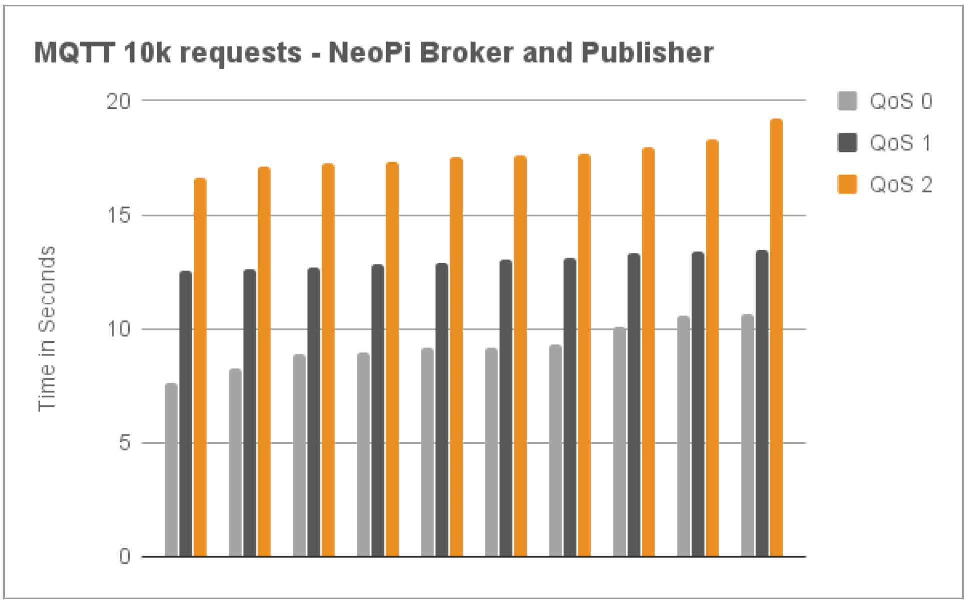


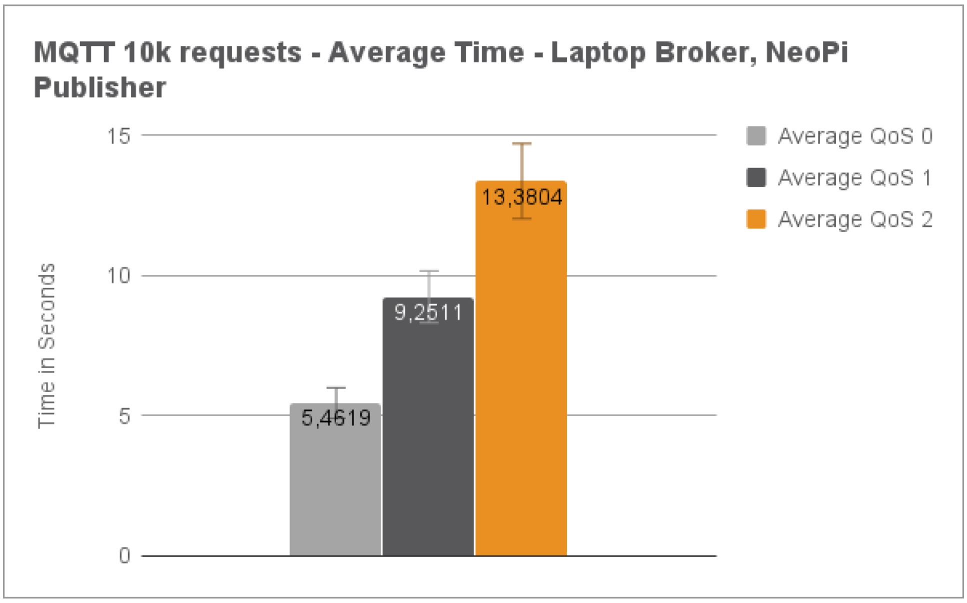


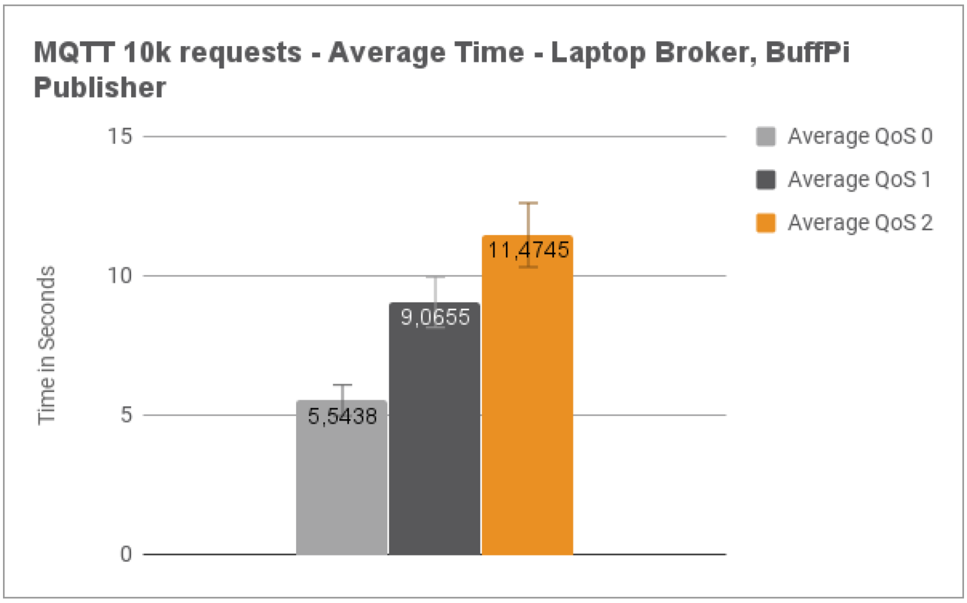

MQTT 10k requests - BuffPi Broker and Publisher
Time in Seconds
15
10
5
0
QoS 0
QoS 1
QoS 2

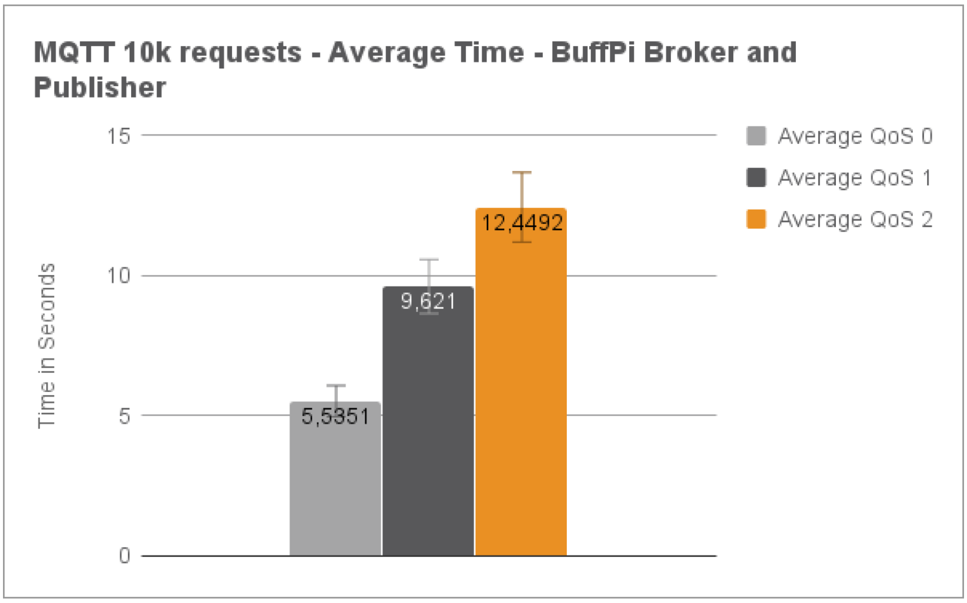

MQTT 10k requests - BuffPi Broker, NeoPi Publisher
Time in Seconds
20
15
10
5
0
QoS 0
QoS 1
QoS 2

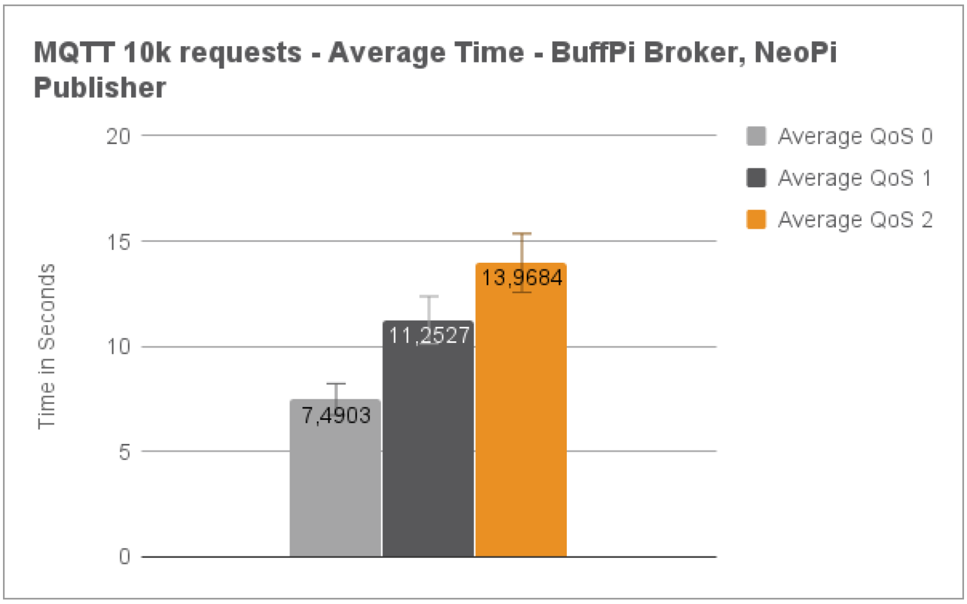

MQTT 10k requests - NeoPi Broker, BuffPi Publisher
Time in Seconds
20
15
10
5
0
QoS 0
QoS 1
QoS 2

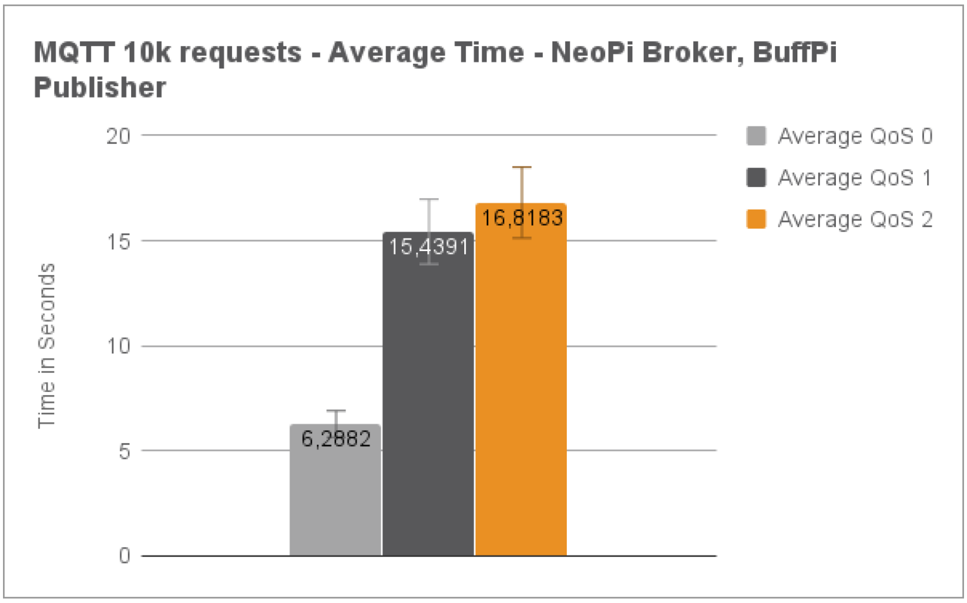

MQTT 10k requests - NeoPi Broker and Publisher
Time in Seconds
20
15
10
5
0
QoS 0
QoS 1
QoS 2

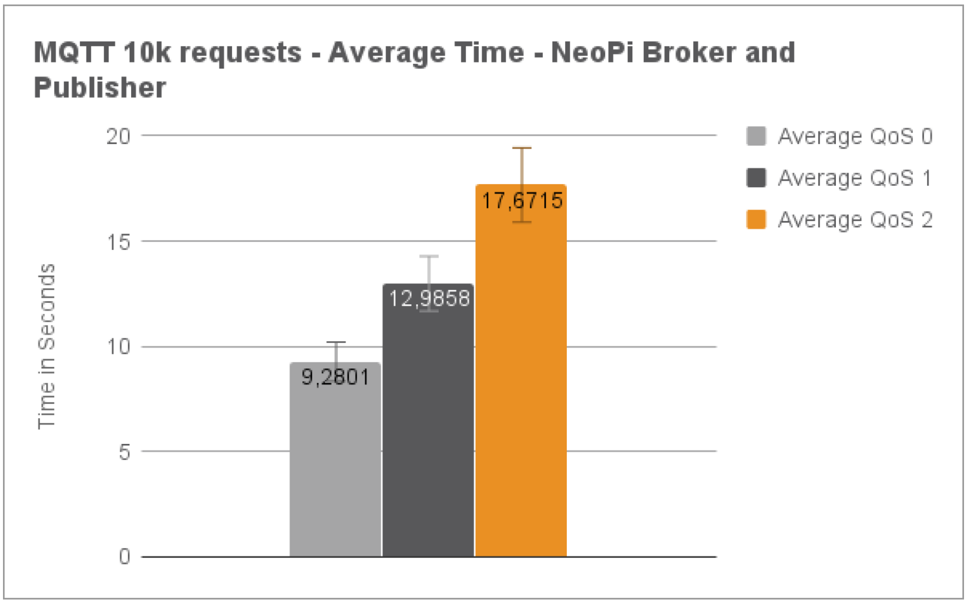

MQTT 10k requests - Average Time - Laptop Broker, NeoPi Publisher
Time in Seconds
15
10
5
0
5,4619
9,2511
13,3804
Average QoS 0
Average QoS 1
Average QoS 2

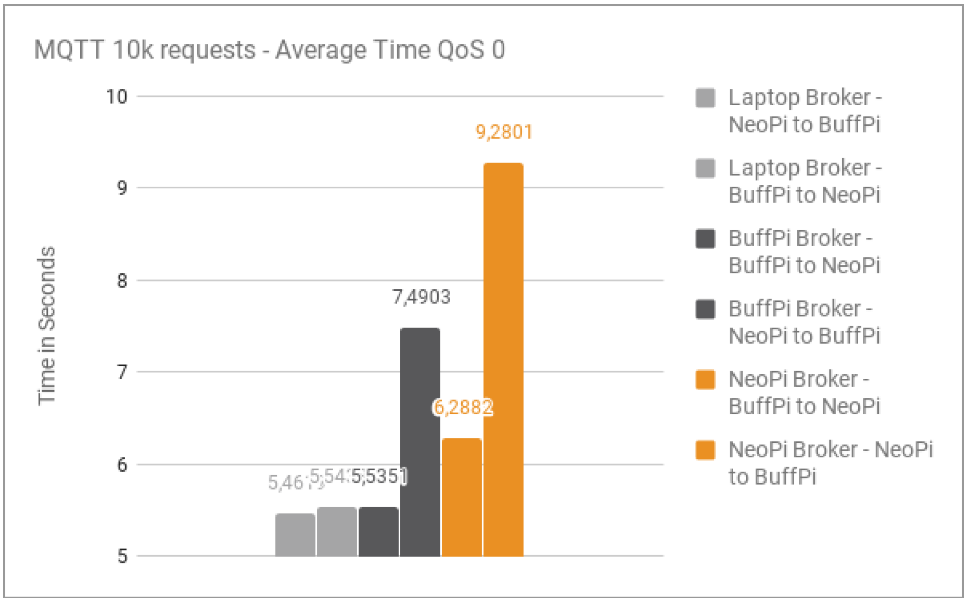

MQTT 10k requests - Average Time - Laptop Broker, BuffPi Publisher
Time in Seconds
15
10
5
0
5,5438
9,0655
11,4745
Average QoS 0
Average QoS 1
Average QoS 2

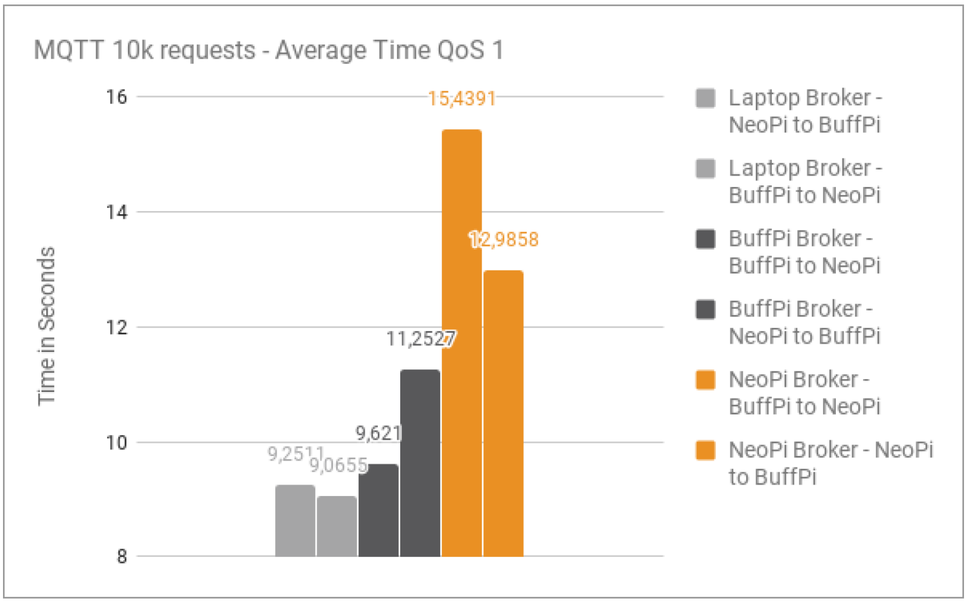

MQTT 10k requests - Average Time - BuffPi Broker and Publisher
Time in Seconds
15
10
5
0
12,4492
9,621
5,5351
Average QoS 0
Average QoS 1
Average QoS 2

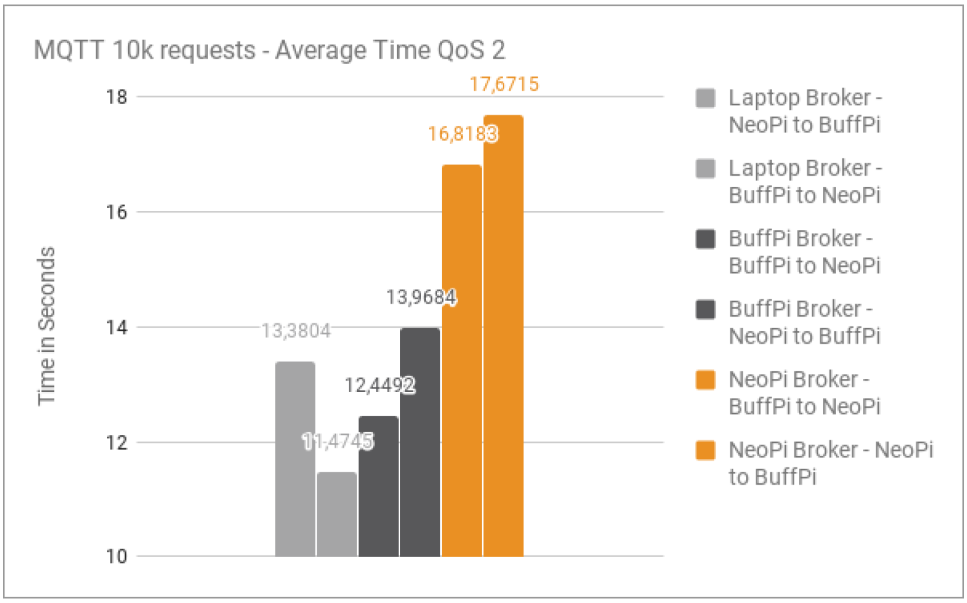

MQTT 10k requests - Average Time - BuffPi Broker, NeoPi Publisher
Time in Seconds
20
15
10
5
0
13,9684
11,2527
7,4903
Average QoS 0
Average QoS 1
Average QoS 2


MQTT 10k requests - Average Time - NeoPi Broker, BuffPi Publisher
Time in Seconds
20
15
10
5
0
6,2882
15,4391
16,8183
Average QoS 0
Average QoS 1
Average QoS 2

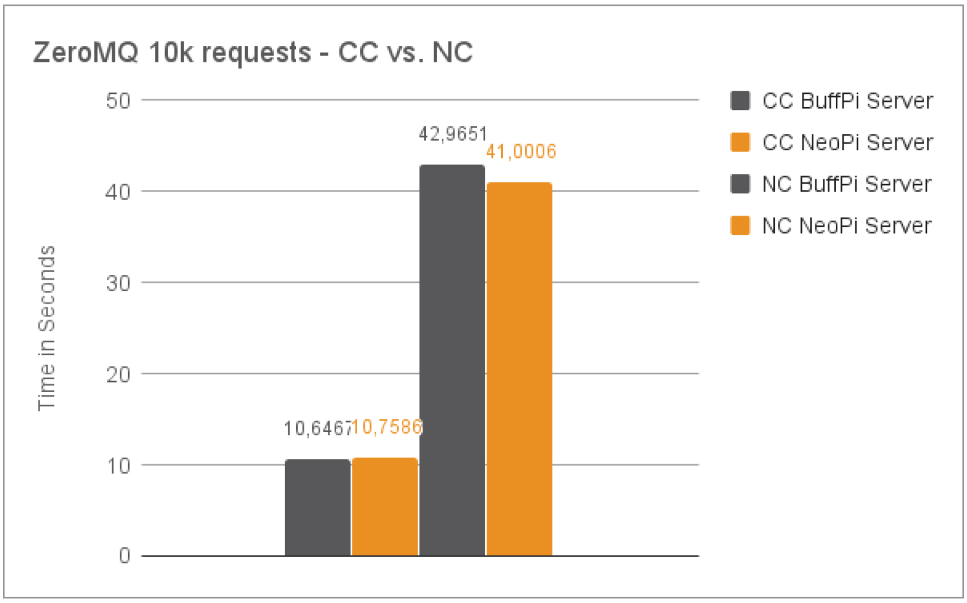

MQTT 10k requests - Average Time - NeoPi Broker and Publisher
Time in Seconds
20
15
10
5
0
9,2801
12,9858
17,6715
Average QoS 0
Average QoS 1
Average QoS 2

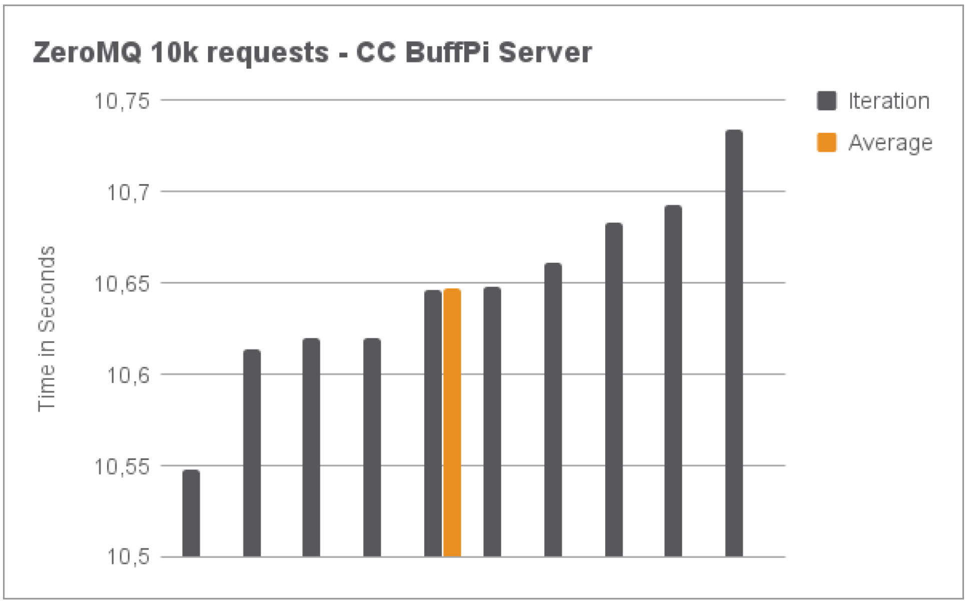

MQTT 10k requests - Average Time QoS 0
Time in Seconds
10
9
8
7
6
5
9,2801
7,4903
6,2882
5,46
5,54
5,5351
Laptop Broker - NeoPi to BuffPi
Laptop Broker - BuffPi to NeoPi
BuffPi Broker - BuffPi to NeoPi
BuffPi Broker - NeoPi to BuffPi
NeoPi Broker - BuffPi to NeoPi
NeoPi Broker - NeoPi to BuffPi

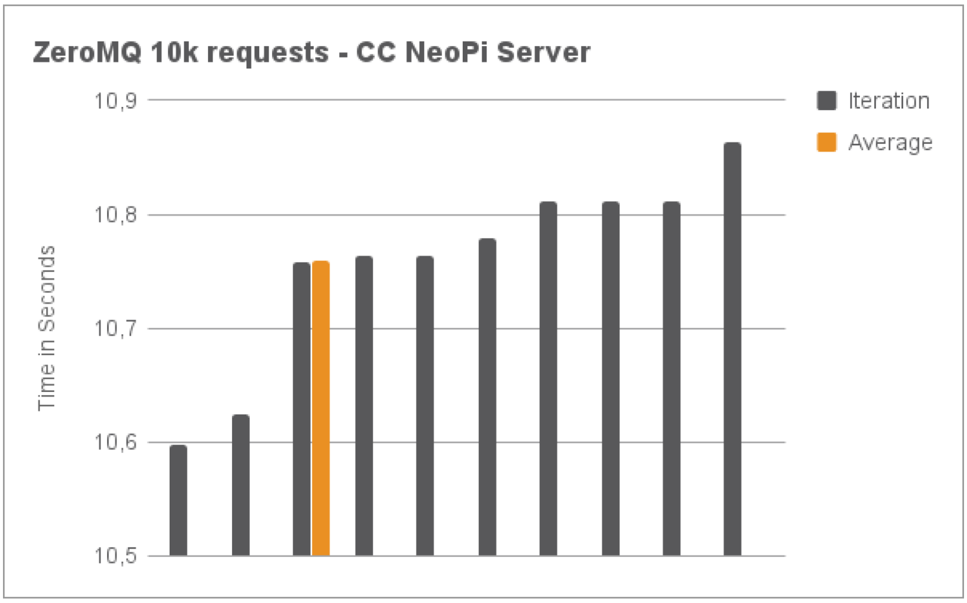

MQTT 10k requests - Average Time QoS 1
Time in Seconds
16
14
12
10
8
15,4391
12,9858
11,2527
9,621
9,2511
9,0655
Laptop Broker - NeoPi to BuffPi
Laptop Broker - BuffPi to NeoPi
BuffPi Broker - BuffPi to NeoPi
BuffPi Broker - NeoPi to BuffPi
NeoPi Broker - BuffPi to NeoPi
NeoPi Broker - NeoPi to BuffPi

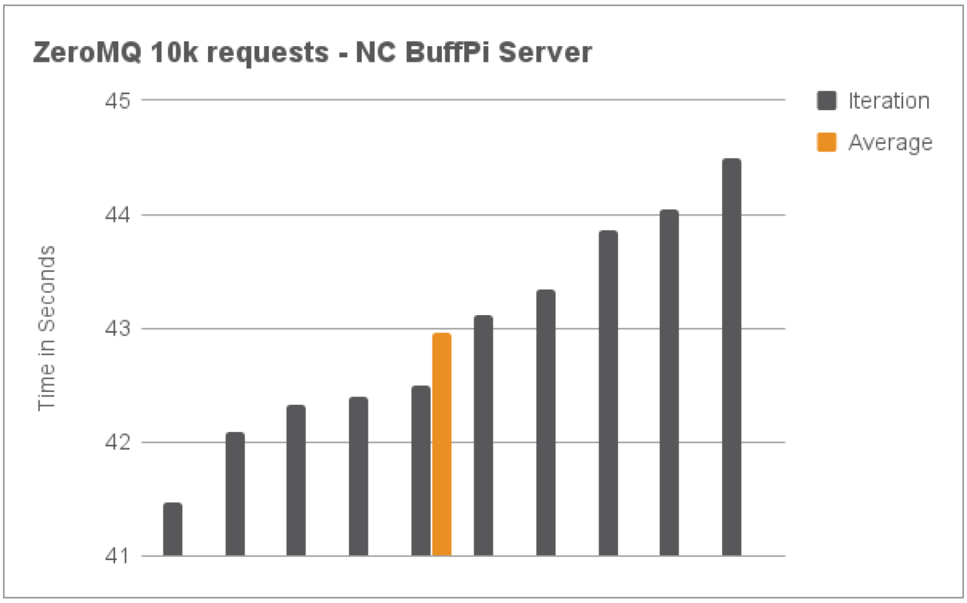

MQTT 10k requests - Average Time QoS 2
Time in Seconds
18
16
14
12
10
13,3804
11,4745
12,4492
13,9684
16,8183
17,6715
Laptop Broker - NeoPi to BuffPi
Laptop Broker - BuffPi to NeoPi
BuffPi Broker - BuffPi to NeoPi
BuffPi Broker - NeoPi to BuffPi
NeoPi Broker - BuffPi to NeoPi
NeoPi Broker - NeoPi to BuffPi

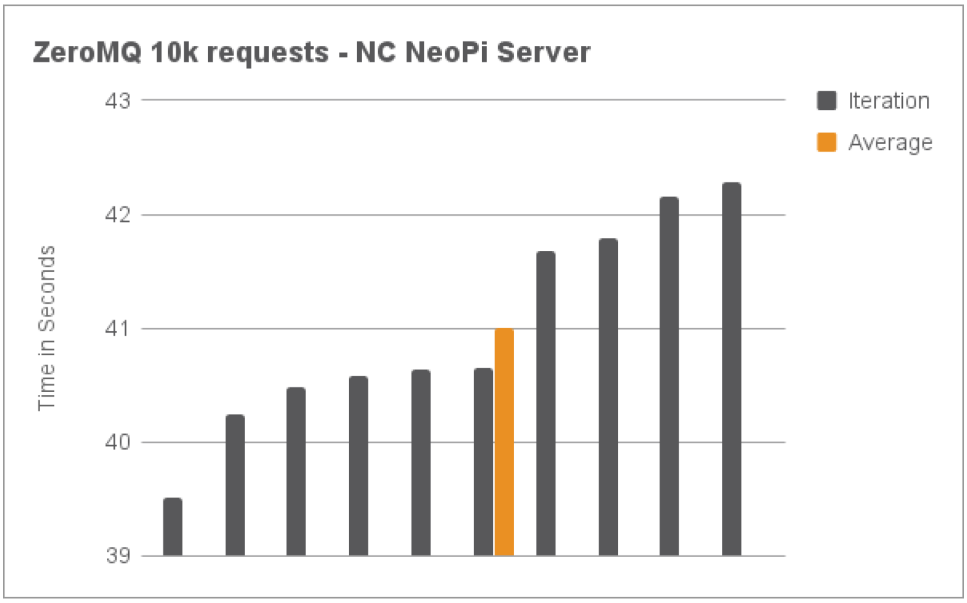
A.2.3 ZeroMQ Graphs

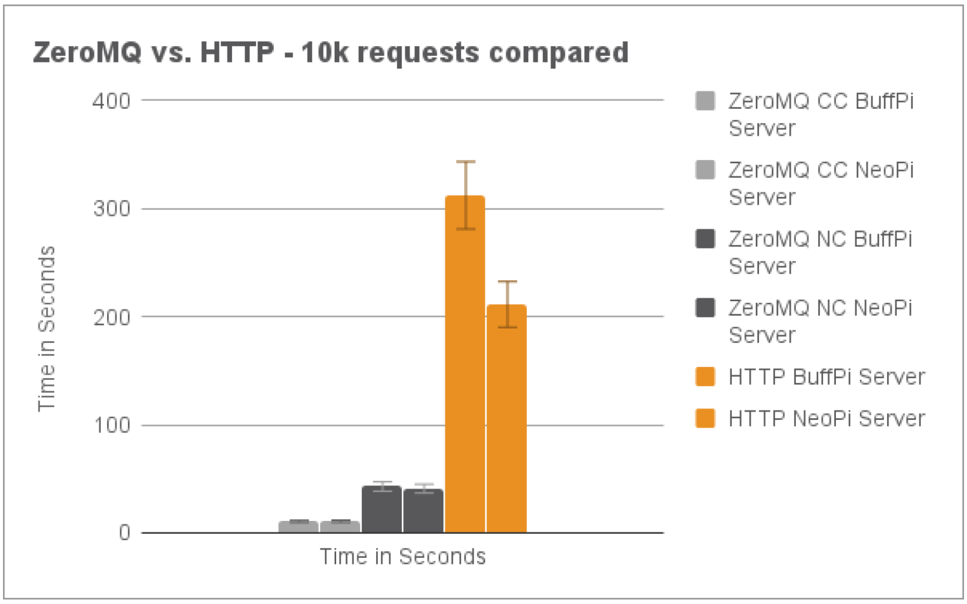





ZeroMQ 10k requests - CC NeoPi Server
Time in Seconds
10,9
10,8
10,7
10,6
10,5
Iteration
Average


ZeroMQ 10k requests - NC BuffPi Server
Time in Seconds
45
44
43
42
41
Iteration
Average


ZeroMQ 10k requests - NC NeoPi Server
43
42
41
40
39
Time in Seconds
Iteration
Average


ZeroMQ vs. HTTP - 10k requests compared
400
300
200
100
0
Time in Seconds
Time in Seconds
ZeroMQ CC BuffPi Server
ZeroMQ CC NeoPi Server
ZeroMQ NC BuffPi Server
ZeroMQ NC NeoPi Server
HTTP BuffPi Server
HTTP NeoPi Server

Bibliography

[1] 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).

[2] *Internet of Things - Pan European Research and Innovation Vision*. Tech. rep. Accessed: 2017-02-27. IERC - European Research Cluster on the Internet of Things, Oct. 2011. URL: http://www.theinternetofthings.eu/sites/default/files/Rob%20van%20Kranenburg/IERC_IoT-Pan%20European%20Research%20and%20Innovation%20Vision_2011.pdf (cit. on p. 1).

[3] 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).

[4] 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).

[5] 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. 2, 8, 11, 14, 25, 26).

[6] 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. 2, 6, 9).

[7] 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, 9).

[8] 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 pp. 2, 25, 26, 28).

[9] 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. 6).

[10] 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 p. 6).

[11] 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 p. 6).

[12] *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. 6).

[13] 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).

[14] 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, 9).

[15] 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).

[16] 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. 6–8).

[17] *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).

[18] O. Jurevicius. *Vertical Integration*. Accessed: 2017-02-13. Apr. 2013. URL: https://www.strategicmanagementinsight.com/topics/vertical-integration.html (cit. on p. 8).

[19] 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).

[20] 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. 9–11).

[21] V. K. Manik and D. Arora. „Performance comparison of commercial VMM: ESXI, XEN, HYPER-V KVM“. In: *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*. Mar. 2016, pp. 1771–1775 (cit. on pp. 9, 10).

[22] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. „Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors“. In: *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys ’07. Lisbon, Portugal: ACM, 2007, pp. 275–287. ISBN: 978-1-59593-636-3. DOI: 10.1145/1272996.1273025. URL: http://doi.acm.org/10.1145/1272996.1273025 (cit. on p. 9).

[23] M. Raho, A. Spyridakis, M. Paolino, and D. Raho. „KVM, Xen and Docker: A performance analysis for ARM based NFV and cloud computing“. In: *2015 IEEE 3rd Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*. Nov. 2015, pp. 1–8. DOI: 10.1109/AIEEE.2015.7367280 (cit. on p. 10).

[24] S. Gallagher. *Mastering Docker*. Packt Publishing Ltd., Dec. 2015. ISBN: 978-1-78528-703-9 (cit. on pp. 10, 16).

[25] 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. 11, 14).

[26] 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. 11).

[27] *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).

[28] 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).

[29] 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).

[30] *NFV*. Accessed: 2017-03-19. URL: https://sdn-wiki.fokus.fraunhofer.de/doku.php?id=nfv (cit. on p. 12).

[31] 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).

[32] *Why is TOSCA Relevant to NFV? Explanation*. Accessed: 2017-03-19. URL: https://www.sdxcentral.com/nfv/definitions/tosca-nfv-explanation (cit. on p. 13).

[33] D. Bernstein. „Containers and Cloud: From LXC to Docker to Kubernetes“. In: *IEEE Cloud Computing* 1.3 (Sept. 2014), pp. 81–84. ISSN: 2325-6095. DOI: 10.1109/MCC.2014.51 (cit. on p. 14).

[34] *Understand images, containers, and storage drivers - Docker*. https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/. Accessed: 2017-02-24 (cit. on p. 14).

[35] *Docker Overview - Docker*. https://docs.docker.com/engine/understanding-docker/. Accessed: 2017-02-24 (cit. on p. 15).

[36] 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. 16).

[37] J. MSV. *Kubernetes architecture*. Accessed: 2017-03-03. Oct. 2016. URL: https://www.slideshare.net/janakiramm/kubernetes-architecture (cit. on pp. 16, 17).

[38] E. Mulyana. *Kubernetes Basics*. Accessed: 2017-03-03. May 2016. URL: https://www.slideshare.net/e2m/kubernetes-basics (cit. on pp. 16, 17).

[39] *Pods - Kubernetes*. Accessed: 2017-03-03. Dec. 2016. URL: https://kubernetes.io/docs/user-guide/pods (cit. on p. 16).

[40] *Labels and Selectors - Kubernetes*. Accessed: 2017-03-03. Dec. 2016. URL: https://kubernetes.io/docs/user-guide/labels (cit. on p. 17).

[41] *kube-proxy - Kubernetes*. Accessed: 2017-03-03. Dec. 2016. URL: https://kubernetes.io/docs/admin/kube-proxy (cit. on p. 17).

[42] *Replication Controller - Kubernetes*. Accessed: 2017-03-03. Dec. 2016. URL: https://kubernetes.io/docs/user-guide/replication-controller (cit. on p. 17).

[43] *Rolling Updates - Kubernetes*. Accessed: 2017-03-03. Dec. 2016. URL: https://kubernetes.io/docs/user-guide/rolling-updates (cit. on p. 17).

[44] *Docker Swarm Documentation*. https://docs.docker.com/engine/swarm. Accessed: 2017-03-18 (cit. on p. 17).

[45] *OpenBaton Documentation*. http://openbaton.github.io/documentation. Accessed: 2017-03-18 (cit. on pp. 17–19).

[46] J. E. Luzuriaga, M. Zennaro, J. C. Cano, C. Calafate, and P. Manzoni. „A disruption tolerant architecture based on MQTT for IoT applications“. In: *2017 14th IEEE Annual Consumer Communications Networking Conference (CCNC)*. Jan. 2017, pp. 71–76. DOI: 10.1109/CCNC.2017.7983084 (cit. on p. 19).

[47] *FAQ - Frequently Asked Questions | MQTT*. Accessed: 2017-06-03. URL: http://mqtt.org/faq (cit. on p. 20).

[48] 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. 20).

[49] 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. 20, 21).

[50] *ØMQ - The Guide*. Accessed: 2017-06-06. URL: http://zguide.zeromq.org/page:all (cit. on pp. 22–24).

[51] *zmq_inproc(7) - 0MQ Api*. Accessed: 2017-06-06. URL: http://api.zeromq.org/3-2:zmq-inproc (cit. on p. 22).

[52] *zmq_tcp(7) - 0MQ Api*. Accessed: 2017-06-06. URL: http://api.zeromq.org/3-2:zmq-tcp (cit. on p. 22).

[53] *zmq_pgm(7) - 0MQ Api*. Accessed: 2017-06-06. URL: http://api.zeromq.org/3-2:zmq-pgm (cit. on p. 22).

[54] B. Butzin, F. Golatowski, and D. Timmermann. „Microservices approach for the internet of things“. In: *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. Sept. 2016, pp. 1–6. DOI: 10.1109/ETFA.2016.7733707 (cit. on pp. 24, 25).

[55] J. Stubbs, W. Moreira, and R. Dooley. „Distributed Systems of Microservices Using Docker and Serfnode“. In: *2015 7th International Workshop on Science Gateways*. June 2015, pp. 34–39. DOI: 10.1109/IWSG.2015.16 (cit. on pp. 26, 27).

[56] J. Rufino, M. Alam, J. Ferreira, A. Rehman, and K. F. Tsang. „Orchestration of containerized microservices for IIoT using Docker“. In: *2017 IEEE International Conference on Industrial Technology (ICIT)*. Mar. 2017, pp. 1532–1536. DOI: 10.1109/ICIT.2017.7915594 (cit. on p. 27).

[57] P. Bellavista and A. Zanni. „Feasibility of Fog Computing Deployment Based on Docker Containerization over RaspberryPi“. In: *Proceedings of the 18th International Conference on Distributed Computing and Networking*. ICDCN '17. Hyderabad, India: ACM, 2017, 16:1–16:10. ISBN: 978-1-4503-4839-3. DOI: 10.1145/3007748.3007777. URL: http://doi.acm.org/10.1145/3007748.3007777 (cit. on pp. 27–29).

[58] R. Morabito, R. Petrolo, V. Loscrí, and N. Mitton. „Enabling a lightweight Edge Gateway-as-a-Service for the Internet of Things“. In: *2016 7th International Conference on the Network of the Future (NOF)*. Nov. 2016, pp. 1–5. DOI: 10.1109/NOF.2016.7810110 (cit. on p. 29).

[59] *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. 31).

[60] *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. 31).

[61] T. K. Authors. *A Cheat Sheet for Understanding "NFV Architecture"*. Accessed: 2017-04-10. URL: https://kubernetes.io/case-studies (cit. on p. 35).

[62] *Kubernetes on RaspberryPI*. Accessed: 2017-07-18. URL: https://github.com/Project31/kubernetes-installer-rpi (cit. on p. 35).

[63] *Welcome to the Kubernetes on ARM project!* Accessed: 2017-07-18. URL: https://github.com/luxas/kubernetes-on-arm (cit. on p. 35).

[64] *Setup Kubernetes on a Raspberry Pi Cluster easily the official way!* Accessed: 2017-07-18. URL: https://blog.hypriot.com/post/setup-kubernetes-raspberry-pi-cluster/ (cit. on p. 35).

[65] *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. 36).

[66] *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. 36).

[67] 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. 36).

[68] *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 p. 38).

[69] *Python Garbage Collection - Digi Developer*. Accessed: 2017-06-17. URL: https://www.digi.com/wiki/developer/index.php/Python_Garbage_Collection (cit. on p. 38).

[70] 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. 39).

[71] *Eclipse Mosquitto*. Accessed: 2017-06-15. URL: https://projects.eclipse.org/projects/technology.mosquitto (cit. on p. 42).

[72] *Foreword - Flask Documentation*. Accessed: 2017-06-18. URL: http://flask.pocoo.org/docs/0.12/foreword (cit. on p. 52).

[73] *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. 52).

[74] *Introducing Flask-RESTful*. Accessed: 2017-06-18. URL: https://www.twilio.com/engineering/2012/10/18/open-sourcing-flask-restful (cit. on p. 52).

[75] *Travis CI Motey build 148*. Accessed: 2017-07-03. URL: https://travis-ci.org/Neoklosch/Motey/builds/242813268 (cit. on p. 67).

[76] 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. 68).

[77] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. *An Updated Performance Comparison of Virtual Machines and Linux Containers*. Tech. rep. Accessed: 2017-07-06. IBM Research Division, July 2014. URL: http://domino.research.ibm.com/library/cyberdig.nsf/papers/0929052195DD819C85257D2300681E7B/$File/rc25482.pdf (cit. on p. 72).

[78] B. Russell. *KVM and docker LXC Benchmarking with OpenStack*. Accessed: 2017-07-10. Apr. 2014. URL: https://de.slideshare.net/BodenRussell/kvm-and-docker-lxc-benchmarking-with-openstack (cit. on p. 72).

[79] F. Ramalho and A. Neto. „Virtualization at the network edge: A performance comparison“. In: *2016 IEEE 17th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*. June 2016, pp. 1–6. DOI: 10.1109/WoWMoM.2016.7523584 (cit. on p. 73).

[80] *ØMQ - The Guide - Unicast Transports*. Accessed: 2017-07-10. URL: http://zguide.zeromq.org/page:all#Unicast-Transports (cit. on p. 75).

[81] *mosquitto.conf*. Accessed: 2017-07-11. URL: https://mosquitto.org/man/mosquitto-conf-5.html (cit. on p. 77).