



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

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

Master Thesis
from

Markus Paeschke

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

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

- Lucius Annaeus Seneca -

Markus Paeschke
Trachtenbrodtstr. 32
10409 Berlin

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

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

Berlin, June 19, 2017

_____

Markus Paeschke

# Acknowledgments

thank your supervisors

thank your colleagues

thank your family and friends

Berlin, June 19, 2017

# Abstract

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

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

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

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

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

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

# Zusammenfassung

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

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

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

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

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

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

# Contents

# List of Figures

# List of Tables

# List of Listings

# Chapter 1

# Introduction

## 1.1 Motivation

The Internet of Things (IoT) is one of the biggest topic in the recent years. Companies with a focus in that area have an enormous market growth with plenty of new opportunities, use cases, technologies, services and devices. Bain & Company predicts an annual revenue of $450 billion for companies who selling hardware, software and comprehensive solutions in the IoT context by 2020.[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. All of them are well connected, for example a Smart Factory, which is a part of the Smart Industry, can get a delivery from a self driving truck (Smart Transport) which navigates through a Smart City to get to the factory. Such information networks are one of the main goals of IoT. In the Industry 4.0 for example multiple smart factories should be interconnect into a distributed and autonomous value chain. Also the automation in a single factory will be increased which helps to have a more flexible and efficient production process. Currently a factory has a high degree of automation, but due to a lack of intelligence and communication between the machines and the underlying system, they can not react to changing requirements or unexpected situations. One solution to achieve that are Cyber-Physical Systems (CPSs). These are virtual systems which are connected with embedded systems to monitor and control physical processes.[cf. 2, p. 363] A normal Cyber Systems (CSs) is passive, means it could not interact with the physical world, with the appearance of CPSs things can communicate so the system has significantly more intelligence in sensors and actuators.[cf. 3, p. 1363 f.]

Another solution is to change the fundamental architecture of such a system from a monolithic to more distributed multicloud architecture. With Fog Computing the cloud moves away from centralized data centers to the edge of the underlying network.[cf. 4, 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"[5, p.236] enables a lot of new possibilities like pre-computation and storage of gathered data, which reduces traffic and resource overhead in the cloud, it keeps sensitive data on-premise[cf. 5, p.236] and enables real-time applications to take decisions based on analytics running near the device and a lower latency. On the other hand there are also a lot of challenges in these highly heterogeneous

and hybrid environment. As an example in some scenarios multiple low power devices have to interact with each other, lossy signals and short range radio technologies are widely used and nodes can appear and disappear frequently.[cf. 6, p. 325] Especially the last case is elaborated because the underlying system has to handle that. Furthermore the required applications running on these nodes can be change commonly and have to be deployed and removed in a dynamical way. Virtualization with Virtual Machines (VMs) is a common approach in Cloud systems to provide elasticity of large-scale shared resources.[cf. 7, 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"[7, p. 117]. Orchestration tools like Kubernetes[1] and Docker Swarm[2] which deploy, scale and manage containers to clusters of hosts have become established in the last years. Moving this technology over to the IoT area many challenges can be solved. Dynamically deployed applications at the edge of a network can store and preprocesses gather data even if a node have no connection to the cloud because of lossy signals. Traffic can be reduced by only transmitting aggregated data back to the cloud. More often small low power devices with limited computational power are be used as IoT nodes which also profit rather from lightweight container solutions than from resource consuming VMs. This thesis shows the capabilities of container orchestration for the IoT and smart factories by creating a prototype which can be executed on fog nodes. Therefor an engine will be created which can orchestrate containers based on functional and non-functional constraints on a single fog node or between a cluster of fog nodes.

## 1.2 Objective

This thesis describes an approach to design and implement a fog service orchestration engine for smart factories. The aim of this work is to create a prototype for a fog node, which can deploy containers on the same node or on other network nodes. A for Node is typically a low power device at the edge of a network, especially in the area of Industry 4.0 and smart factories. Furthermore the prototype should consider specific functional and non-functional constraints while deploying the containers. A condition can be a hardware requirement, a required software or a dependencies to another node. The technical prerequisite and detailed requirements for the prototype as well as the usability of a 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 they have a lot of knowledge and experience especially in this area of IoT and NFV.

## 1.3 Scope

As mentioned before, the prototype will be an prototype for a fog node, this means the creation of a MANO engine on the server side is out of scope for this thesis. Open Baton serves as the template for that project and especially the modular architecture and the resulting

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

extensibility will be targeted. Beside that, the whole prototype will be created completely from scretch. The container virtualization will be realized with Docker[3], an open source container platform. Docker has on Application Programming Interface (API) where third party apps can communicate with and can control the engine himself. The functional and non-functional constraints could base on YAML Ainť Markup Language (YAML) schemas which are part of the Topology and Orchestration Specification for Cloud Applications (TOSCA) standard for example. These schemas should be extendable and should fit the needs of the constraint functionality.

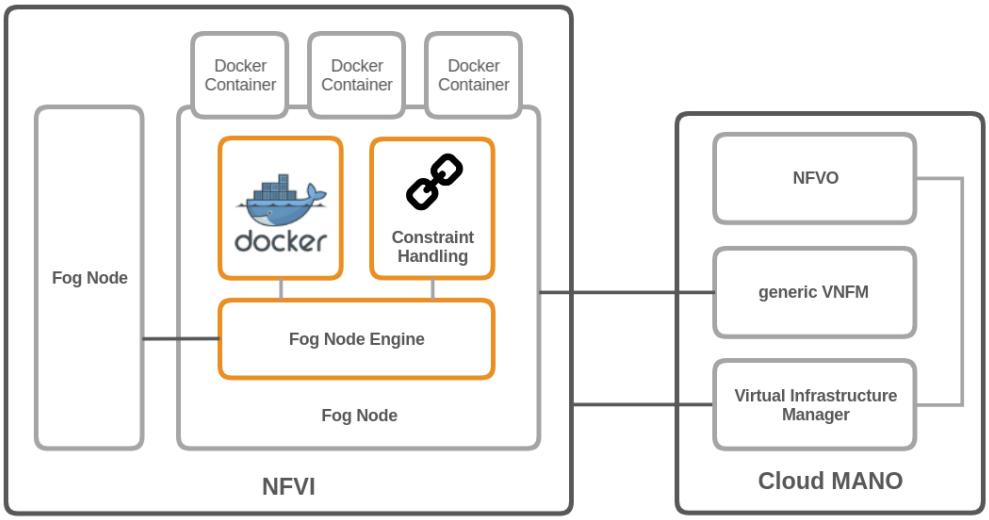


Figure 1: Conceptual architecture.

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

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

## 1.4 Outline

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

# Chapter 2

# State of the art

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

## 2.1 Internet of Things

The IoT has been a subject of great media- and economically growth in the recent years. In the year 2008 the number of devices which are connected to the Internet was higher than the human population.[cf. 8, p. 3] Cisco Internet Business Solutions Group predicted that the number will grow up to 50 billion in 2020, this equates to around 6 devices per person.[cf. 8, p. 4] Most of todayś interactions are Human-to-Human (H2H) or Human-to-Machine (H2M) communication. The IoT on the other hand aim for the Machine-to-Machine (M2M) communication. This allows every physical devices to be interconnected and to communicate with each other. These devices are also called "Smart Devices". Creating a network where all physical objects and people are connected via software is one primary goal of the IoT.[cf. 9, p.206][cf. 10, p.2] When objects are able to capture and monitor their environment, a network can perceive external stimuli and respond to them.[cf. 11, p. 40] Therefore a new dimension of information and communication technology will be created, where users have access to everything at any time, everywhere. In addition to smart devices, subcategories are also emerging from the IoT which, in addition to the physical devices, also describe technologies such as protocols and infrastructures. The "Smart Home" has been a prominent topic in media and business for many years. Smart City or Industrie 4.0 are also becoming established and are increasingly popular. But the Internet started with the appearance of bar codes and Radio Frequency Identification (RFID) chips.[cf. 10, p. 13] The second step, which is more or less the current situation, sensors, physical devices, technical devices, data and software are connected to each other.[cf. 10, p. 13] This was

achieved, in particular, by cloud computing, which provides the highly efficient memory and computing power that is indispensable for such networks.[cf. 9, 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. 10, p. V] Just as the omnipresent information processing in everyday life, also known as "Ubiquitous Computing", which was first mentioned in the "The Computer for the 21st Century"[12] by Marks Weiser, it will take some time until it is ubiquitous.

### 2.1.1 Industry 4.0 and smart factories

The industry as an changing environment is currently in the state of the so called "fourth industrial revolution". The first industrial revolution was driven by steam powered machines. Mass production and division of labor was the primary improvement of the second industrial revolution, whereas the third revolution was characterized by using electronics and the integration of Information Technology (IT) into manufacturing processes.[cf. 13, p. 1] In the recent years the size, cost and power consumption of chipsets are reduced which made it possible to embed sensors into devices and machines much easier and cheaper.[cf. 5, 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. 13, p. 1] "Industrie 4.0 is a collective term for technologies and concepts of value chain organization."[cf. 14, 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.[15, cf.] It also aims for an intelligent connection between different companies and units. Autonomous production and logistics processes creating a real-time lean manufacturing ecosystem that is more efficient and flexible.[15, 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."[15] At the end, the system can use customer wishes in every step in the process to be flexible and responsive.[15, 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. 14, p. 11]

Table 1 shows the six design principles which can be from the Industrie 4.0 components. They can help companies to identify and implement Industry 4.0 scenarios.[cf. 14, p. 11]

1. *Interoperability* CPS of various manufacturers are connected with each other. Standards will be the key success factor in this area.[cf. 14, 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. 14, 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. 14, 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. 14, 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. 14, 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. 14, p. 11]

Another important aspect of Industry 4.0 is the implementation of process automation with the focused on three distinct aspects. Starting with the vertical integration, which contains the connection and communication of subsystems within the factory enables flexible and adaptable manufacturing systems.[cf. 16, p. 7 ff.] The horizontal integration, as the second aspect, enables technical processes to be integrated in cross-company business processes and to be synchronized in real time through multiple participants to optimize value chain outputs.[cf. 16, p. 7 ff.] Finally end-to-end engineering, planning, and process control for each step in the production process.[15, cf.]

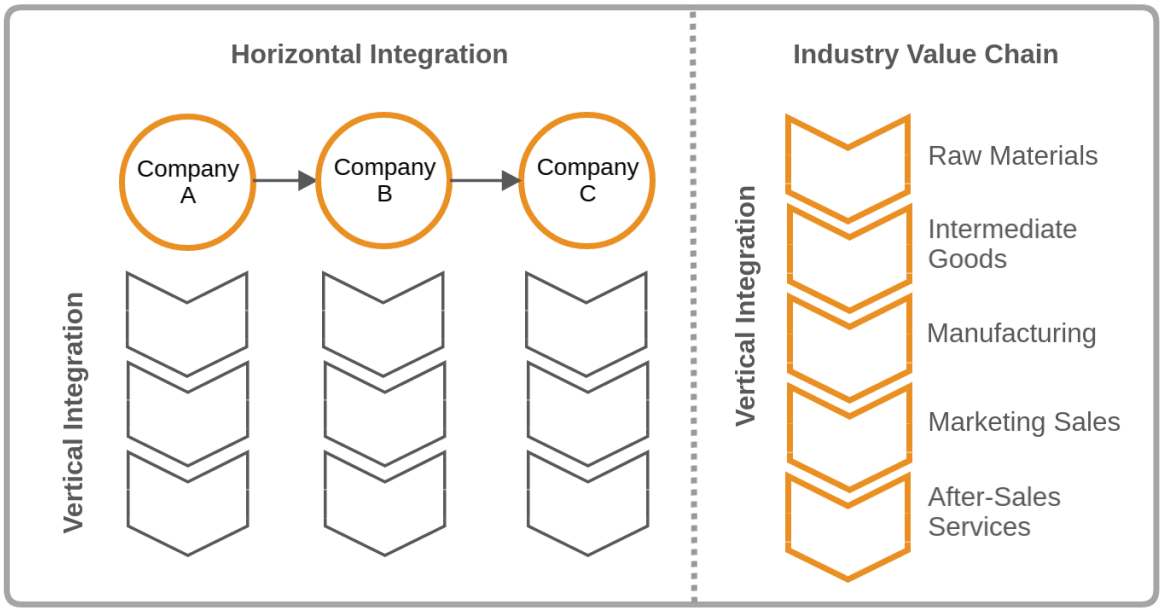


Figure 2: Horizontal vs. Vertical Integration. Adapted from: [17]

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

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

### 2.1.2 Cyber Physical Systems

As we already now, in smart factories every physical device is connected to each other. Everything can be captured and monitored in each step of a production process. With CPSs every physical entity has a digital representation in the virtual system.[cf. 3, p. 1363] Before a CS was passive, which means there was no communication between the physical and the virtual world.[cf. 3, p. 1364] While new technologies in the physical world, like new materials, hardware and energy, are developed, the technologies in the virtual worlds are also being improved, for example through the use of new protocols, networking, storage and computing technologies.[cf. 3, p. 1364] This adds more intelligence in such systems, as well as a much more flexible and modular structure. A CPS can organize production automatically and autonomously, which eliminate the need of having a central process control.[13, cf.] Thereby the system can handle lossy signals and short range radio technologies, which are widely used in such a context.[6, cf.] In summary CPSs can help to enable the vision of smart factories in both the horizontal as well as the vertical integration.

### 2.1.3 Fog Computing

In the beginning of Cloud Computing most of the systems based on a monolithic architecture. Over time the system was broken down to a more distributes multicloud architecture, similar to microservices. With the appearance of Fog Computing the Cloud also moves from centralized data centers to the edge of the underlying network. Main goal is to improve the efficiency, reduce the traffic and the amount of data which is transferred to the cloud and also process, analyze and store data locally, as well as keeping sensitive data inside the network for security reasons.[cf. 5, p. 236][cf. 6, p. 325][cf. 13, p. 4] In contrast to the goals the definition and understanding of Fog Computing differs. One perspective is to that the processing of the data take place on smart devices, e.g. sensors, embedded systems, etc., at the end of the network or in smart router or other gateway devices.[cf. 13, p. 4] Another interpretation is that fog computing appears as an intermediate layer between smart devices and the cloud.[cf. 5, 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. 6, 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)."[18, p. ES-1]. This means a VM, also referred as guest system, can be executed in a real system, which is referred as host system. A VM has its own Operating System (OS) which is completely isolated from the other VMs and the host system.[cf. 19, p. 2] Basically there are two types of virtualization: Process virtualization where the virtualizing software also known as Virtual Machine Monitor (VMM) is executed by the host OS and only an application will be executed inside the guest OS and on the other side there is the the system virtualization where the whole OS as well as the application are running inside the virtualizing software. Figure 3 illustrate both concepts. Examples for process virtualization could be the Java Virtual Machine (JVM)[1], the .Net framework[2] or Docker[3], where VMWare[4], Oracle Virtual Box[5], XEN[6] or Microsoft Hyper-V[7] are only some examples for system virtualization. The benefits of all virtualization techniques are the rapid provisioning of resources which could be Random Access Memory (RAM), disk storage, computation power or network bandwidth. Beside that, no human interaction is necessary during the provisioning process. Elasticity which scales a system in a cost-efficient manner in both directions, up and down. Customer as well as the provider profit from such a system. Security based on the isolation of the VMs is another huge benefit. Different processes can not interfere with each other and the data of a single user can not be accessed by other users of the same hardware. A challenge despite all the mentioned benefits is the performance. Running VMs increases the overhead and reduces the overall performance of a system. Therefore the specific use case have to consider these behavior.

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

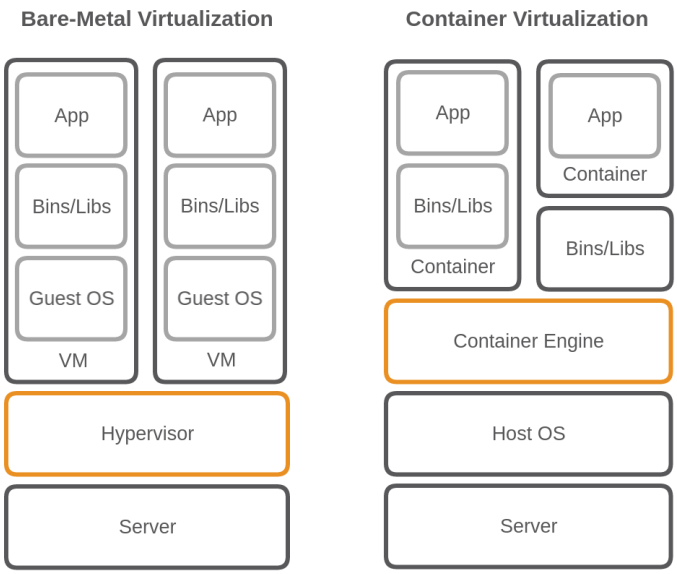


Figure 3: Structure bare-metal virtualization vs. container virtualization. Adapted from: [20, p. 2]

### 2.2.1 Virtual Machines

VMs are the core virtualization mechanism in cloud computing. There are also two different designs for hardware virtualization. The first and more popular type for cloud computing is the *bare-metal virtualization*. It needs only a basic OS to schedule VMs. The hypervisor runs directly on the hardware of the machine without any host OS in between. This is more efficient, but requires special device drivers to be executed. The other type is the *hosted virtualization*. Unlike the first type the VMM run as a host OS process and the VMs as a process supported by the VMM. No special drivers are needed for these type of virtualization, but by comparison the overhead is much bigger. For both types, the performance limitation remains. Each VM need a full guest OS image in addition to binaries and libraries which are necessary for the application to be executed.[cf. 4, p. 381] If only a single application, which only needs a few binaries and libraries, is needed to be virtualized, VMs are too bloated.

### 2.2.2 Container Virtualization

Container virtualization which is also known as Operating System-level virtualization, is the second virtualization mechanism. It based on fast and lightweight process virtualization to encapsulate an entire application with its dependencies into a ready-to-deploy virtual container.[cf. 21, p. 72] Such a container can be executed on the host OS which allows an application to run as a sand-boxed user-space instance.[cf. 22, p. 1] All containers share a single OS kernel, so the isolation supposed to be weaker compared to hypervisor based virtualization.[cf. 19, 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. 19, p. 2]

### 2.2.3 Container Orchestration

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

### 2.2.4 Network Function Virtualization

NFV is an architectural framework to provide a methodology for the design, implementation, and deployment of Network Functions (NFs) through software virtualization.[cf. 23, p. 8][24, cf.] "These NFs are referred as Virtual Network Functions (VNFs)."[23, p. 8] It takes into consideration Software Defined Networking (SDN) and preparing for the use of non-proprietary software to hardware integration instead of multiple vendor specific devices for each function, e.g. routers, firewalls, storages, switches, etc.[24, cf.] Now high-performance firewalls and load balancing software for example can run on commodity PC hardware and traffic can be off-loaded onto inexpensive programmable switches.[25, cf.]

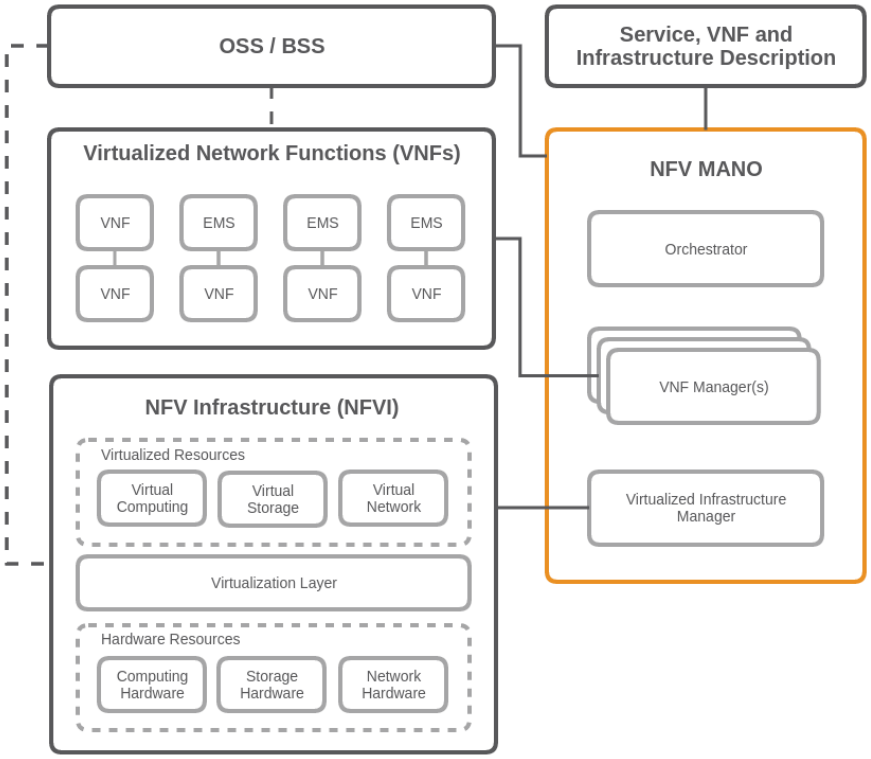


Figure 4: NFV architecture. Adapted from: [26]

Some benefits are speed, agility and cost reduction in deployment as well as execution manner.[25, cf.] Using homogeneous hardware simplifies the process of planning and reduces power, cooling and space needs.[25, cf.] Through virtualization providers can utilize resources more effectively, by allocating only the necessary resources for a specific functionality.[25, cf.] Overall NFV can reduce Operating Expense (OpEx) as well as Capital Expenditure (CapEx) and can decreasing the time necessary to deploy new services to the network.[25, cf.] To achieve NFV the ETSI has defined a framework the Network Function Virtualistion Management And Orchestration (NFV-MANO)[8] and the Organization for the Advancement of Structured Information Standards (OASIS) created TOSCA a NFV specific data model and templates to coordinate and orchestrate the NF into the cloud.

**OSS / BSS** refers to the Operations Support Systems (OSS) and the Business Support Systems (BSS) of a telecommunication operator.[27, cf.] The OSS is responsible for the underlaying 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.[27, cf.]

**VNFs** are the virtualized network elements, for example a virtualized router or virtualized firewall. Even if only a sub-functions or a sub-components of a hardware element is virtualized, it is called VNF.[27, cf.] Multiple sub-functions can act together as one VNF.

**Element Management System (EMS)** is responsible for the functional management of a single or multiple VNFs.[27, cf.] This includes fault, configuration, accounting, perfomance and security management.[27, cf.] Furthermore the EMS itself can be a VNF or it can handle a VNF through proprietary interfaces.[27, cf.]

**NFVI** is the environment where VNFs are executed. This includes physical resources as well as virtual resources and the virtualization layer. The physical resources could be a commodity switch, a server or a storage device. These physical resources can be abstracted into virtual resources through the virtualization layer which is normally a hypervisor. If the virtualization part is missing, the software runs natively on the hardware and the entity is no longer a VNF it is then a Physical Network Function (PNF).[27, 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"[27]. 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.[28, cf.] In contrast to the EMS, the VNFM handles the virtual part of the VNF, for example instantiate an instance, while the EMS handles the functional part of an VNF, such as issue handling for a VNF. The orchestrator as the third component in the NFV-MANO block manage network services of VNFs. It is responsible for the global resources management, such as computing

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

and networking resources among multiple VIMs.[27, cf.] The orchestrator interacts with the VNFM to perform actions, but not with the VNFs directly.[27, cf.] TOSCA is often used with NFV-MANO frameworks like Cloudify[9] or Open Baton.[28, cf.]

**TOSCA** is developed by the OASIS to deliver a declarative description of a NFV application topology for network or cloud environments.[28, cf.] In figure 4 it is represented by the *Service, VNF and Infrastructure Description* block. Beside that, it can also be used to define workflows which should be automated in a virtualized environment.[28, cf.] The TOSCA modeling language can specify ndoes, 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.[28, cf.] To define schemas, relationships and the configuration of such an infrastructure, it uses YAML files for ease the usage.[28, cf.] TOSCA works pretty well with NFV-MANO components to automate the deployment and management of NFs and services.

## 2.3 Existing tools

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

### 2.3.1 Linux Containers

When we talk about container virtualization nowadays, Docker have to become one of the most famous tools out there. It based on Linux Containers (LXC)[10] a technology which uses kernel mechanisms like *namespaces* or *cgroups* to isolate processes on a shared OS.[cf. 4, 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. 4, 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."[21, p. 72] There are several other container virtualization tools out there like OpenVZ[11] or Linux-VServer[12]. In contrast to them, an advantage of LXC is that it runs on an unmodified Linux kernel. This means that LXC can be executed in most of the popular Linux distributions these days.

### 2.3.2 Docker

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

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

from images. A mayor benefit of Docker is the fact, that Docker images can be combined like build blocks. Each image can build on top of another. Figure 5 illustrates the concept for the image of the pretty famous Django[13] web framework. In that case, the Django image, which is the resulting image, based on the Python 3.4 image which again based on the Debian Jessy image. All of them are read-only, but Docker adds a writable layer, also known as *container layer*, on top of the images as soon as the container will be created. File system operations such as creating new files, modifying or deleting existing files are written directly to these layer.[29, cf.] The other images dont́ get involved. These chaining mechanism of images allows Docker to ease the use of dependencies and administrative overhead.

Beside that, Docker is split up in several components, such as the Docker Engine, the Docker Registries and Docker Compose to mention only a few. The Docker Engine is a client-server application which can be distinguished by the Docker client, a Representational State Transfer (REST) API and the Docker server. Latter is a daemon process which "creates and manages Docker objects, such as images, containers, networks, and data volumes"[30]. The client is a Command Line Interface (CLI), which interact via a REST API with the daemon.[30, cf.]

Another component, the Docker Registry, is basically a library of Docker images. They can be public or private available, as well as on the same machine like the Docker daemon or an external server.[30, cf.] The most popular one it the official Docker Hub[14]. There is also an Docker Store[15], where customers can buy and sell trusted and enterprise ready containers and plugins. With the Docker client it is pretty easy to *search* for new containers and to *pull* containers from or *push* containers to a specific repository.

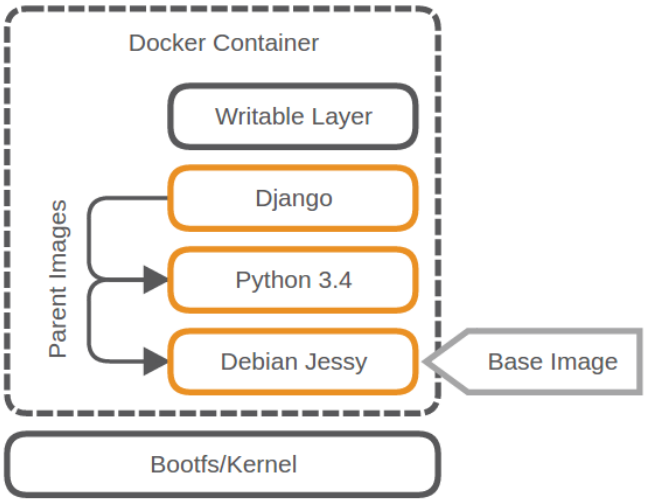


Figure 5: Docker container structure.

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

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

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

A major benefit of Docker is that the execution environment of an application is completely the same on a local machine as on the production environment.[cf. 20, 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. 20, p. 2]

### 2.3.3 Kubernetes

Kubernetes is an open source container cluster manager which was released 2014 by Google. It is "a platform for automating deployment, scaling, and operations"[31, 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.[31, p. 5 f.]

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

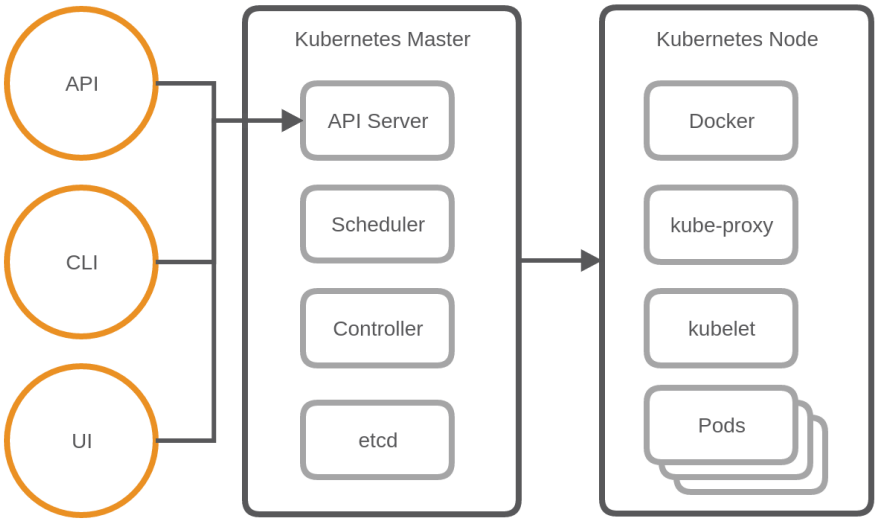


Figure 6: Kubernetes architecture. Adapted from: [32, p. 4]

Each node has a two way communication with the master via a kubelet. In addition each node has the services necessary to run container applications like Docker. Furthermore Kubernetes can combine one or multiple containers into one so called Pod.[cf. 33, p. 7] "Pods are always co-located and co-scheduled, and run in a shared context."[34] One node again

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

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. Pods can also share functionality to other Pods inside a Kubernetes cluster. A logical set of Pods and the access policy of them is called a Kubernetes service. Such a service can abstract multiple Pod replicas and manage them. A frontend which have access to the service do not care about changes in the service. Any change, be it a down scale or an up scale of the system, remains unseen for the frontend. They are exposed through internal or external endpoints to the users or the cluster.[cf. 32, p. 11] Labels can be used to organize and to select subsets of Kubernetes Objects, such as Pods or Services.[35, cf.] They are simply key-value pairs which and should be meaningful and relevant to users, but do not imply semantics to the core system.[35, 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. 33, p. 7] "Each node and can do simple TCP,UDP stream forwarding or round robin TCP,UDP forwarding across a set of backends."[36] The Replication Controller is one of the mayor controllers in a Kubernetes System. It ensures that a specified number of pod replicas are running and available at any time.[37, cf.] If for example one node disappear because of connection issues, the Replication Controller will start a new one. If the disappeared node is available back again it will kill a node. These functionality increases the stability, the availability and the scalability of the system in an autonomous manner. The last important component in Kubernetes are rolling updates. With rolling updates the system can update one pod at a time, rather than taking down the entire service and update the whole system.[38, 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.[39, cf.] No additional software is necessary to have a bunch of machines work together as a so called swarm. Similar to Kubernetes, Docker Swarm needs a master node called manager and several worker nodes. The manager for example keep track of the nodes and their lifecycle and it can start new instances of an image if one or multiple nodes disappear. Furthermore Docker Swarm has a build in proxy and load balancer, which can redirect requests to the node with the necessary container running on it or redirect requests based on the workload of the machines. Compared to Kubernetes, Docker Swarm is more lightweight, but misses some features like the label functionality or the schema definition of a pod. But as mentioned before, both tools are pretty similar and aim for the same goal.

### 2.3.5 Open Baton

Open Baton[20] is an open source ETSI NFV compliant MANO Framework[40, cf.]. "It enables virtual Network Services deployments on top of heterogeneous NFV Infrastructures."[40] It works together with OpenStack and provides a plugin mechanism which allows to add additional VIMs.[40, cf.] In Open Baton it is implemented as the VIM as first Point of Pres-

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

ence (PoP) and uses the OpenStack APIs.[40] All the resources in the NFVI are controlled by the VIM, in this case OpenStack.

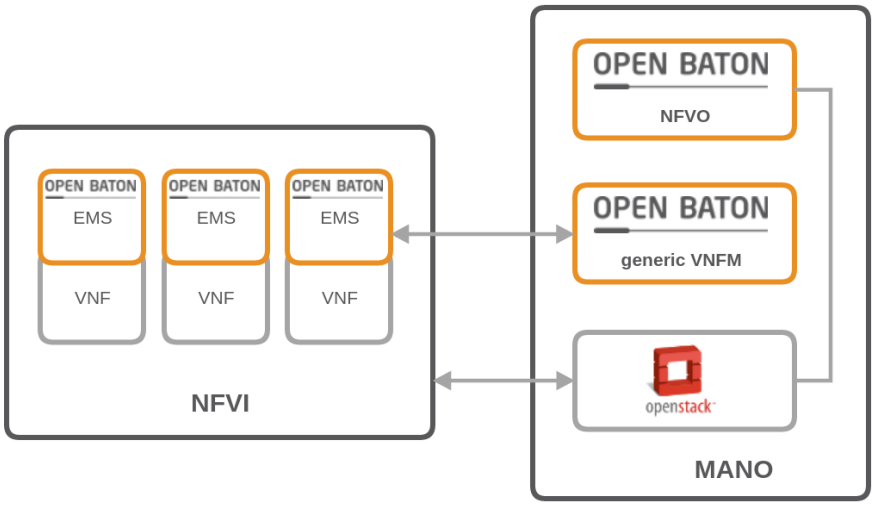


Figure 7: Open Baton abstract architecture. Adapted from: [40]

In the basic configuration Open Baton provides a generic VNFM with generic EMS related to the VNFs, but it can also be replaced with custom components. The VNFM can use a REST API or an Advanced Message Queuing Protocol (AMQP) message queue to communicate with the system. Figure 7 illustrates the abstract architecture of Open Baton together with OpenStack and a generic VNFM. The Network Function Virtualistion Orchestrator (NFVO) is completely designed and implemented as described in the ETSI MANO standard.[40] 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.[40] 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.[40] The VIM Driver mechanism allows to replace OpenStack with external heterogeneous PoPs, but without the need of modifying the orchestration logic.[40] Beside the generic VNFM, also the Juju[21] VNFM can be used to deploy Juju Charms or Open Baton VNF packages. Open Baton also provides a marktplace[22] for free and open source VNFs, which can directly be loaded into the system.

Furthermore Open Baton comes with a modern and easy to use GUI and user management. The typical workflow of running a NFVI is by starting them through the dashboard. The user input, in this case deploying a VNF, will be submitted as a request to the NFVO. There the orchestrator request the VIM, for example OpenStack, to instantiate the network service. The VIM allocated the resources on the datacenter and starts the VMs based on the provided

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

service description, for example through a TOSCA description. After the machines are finally booted, the EMSs will be installed to communicate with the VNFMs. Open Baton now can send lifecycle events to all the VNFMs responsible for the VNFs which are part of the network service. Finally the VNFMs processes the VNFs via the EMS on to the given resources of the NFVI on the datacenter. The services are started and the system is up and running.

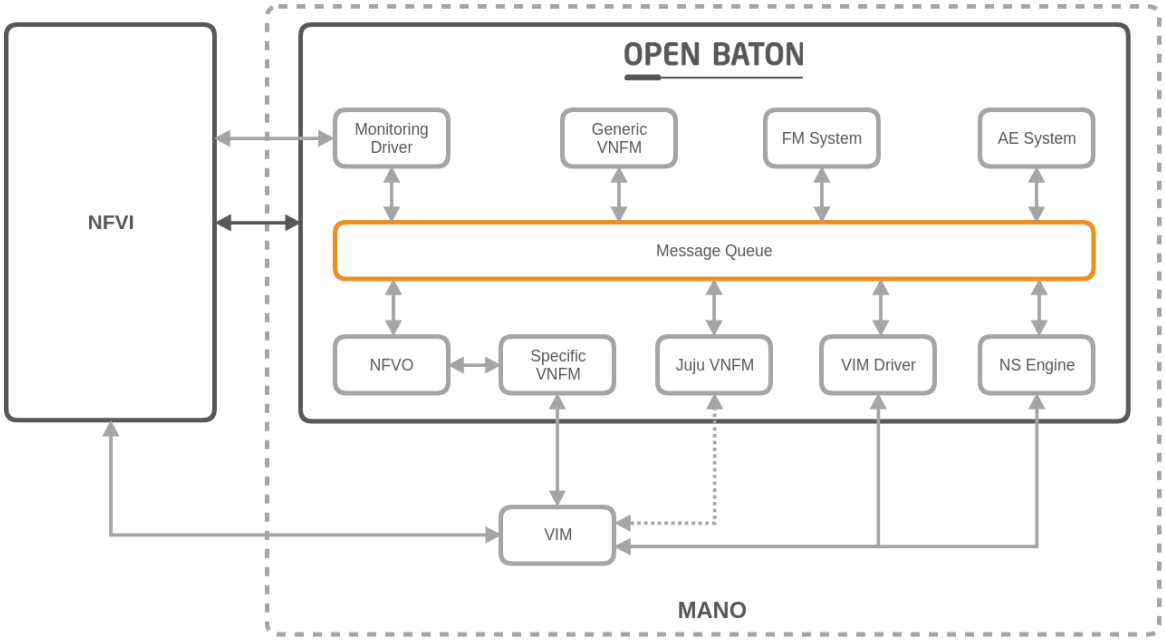


Figure 8: Open Baton detailed architecture. Adapted from: [40]

## 2.4 Messaging

### 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.[41, cf.] Meanwhile MQTT is an OASIS standard[23], which is often used in an IoT and M2M context.[cf. 42, p. 5] It has a publish/subscribe architecture, which makes it easy to implement and allows thousands of remote clients to be connected to a single server at the same time.[cf. 42, p. 5] The recipient of a message which is called consumer in the MQTT context is completely decoupled from the sender, mostly called producer, via a broker. In general the workflow is that a consumer subscribe to a specific topic at the broker and the producer can send messages with a specific topic to the broker. The producer does not know if there is any consumer subscribed to the topic of a message. The broker is resonsible 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 eventoriented, which means that the client does not have to constantly ask the server if there is new data, the

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

broker informs the consumer when there is new data on a topic.[43, cf.] In direct comparison, this concept decrease the traffic, the amount of connections at the server and the delay of the message to be send to the clients.

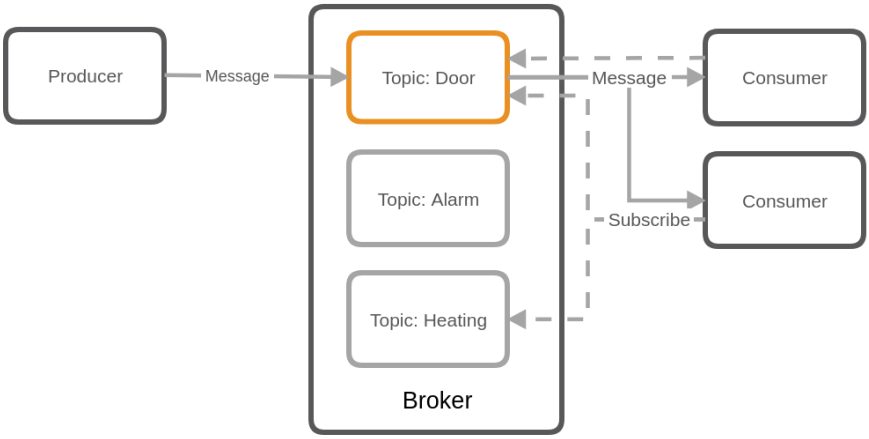


Figure 9: MQTT publish/subscribe architecture. Adapted from: [43]

Beside the publish/subscribe architecture and the lightweight protocol, MQTT has only few but usful features. **Quality of Service (QoS)** define the level of reliability with which messages are delivered.[43, cf.] There are three different levels, where each level differs in reliability and resources usage.[43, cf.] In an IoT context, for instance gathering temperature sensor data from a low power device, most of the time keeping the resources usage low is more important than the reliability of getting every single message.
At *QoS level 0 - at most once*, it is garantied that a message can only arrieves once, but they can also be lost during transfer. A single message will be send once and the publisher does not check the success of receiving them. This pattern is also called *Fire and Forget*, it is fast and resources friendly.[43, cf.]
At *QoS level 1 - at lest once*, it is grantied that a message will be received at the consumer. After the publisher send the message with a specific packet identifier, the consumer will confirm the receipt of the message with a so called pubback packat. The pubback packet also have the same packet identifier included, so that the publisher will know that the message was successfully delivered. If a message get lost during transmission, they will be resend. It is also possible, that the same message appears multiple times at the consumer, depending on the delay in the network.
The most secure but also most resources consuming level is At *QoS level 2 - exactly once*. At this level it is garantied, that a message is exactly received once. It is not possible that a message appears multiple times or never. This level has a two-level confirmation process, where at first the consumer confirms the receive of the message and afterwards the producers confirms the receive of the confirmation. Therefor both sides can be sure to not send or receive duplicates and it is garantied that a message will be received.

Forthermore the broker has two features to handle connection loses: the *last will testament* and the *message persistence*. The former feature will send a last message from the broker to a consumer if the connection to the publisher will get lost. This could be for instances the information that the connection get lost or something similar. The message persistance will be used if one consumer lose the connection to the broker. In this case the message will be

stored and delivered as soon as the consumer reconnects. With the *retrained messages* feature, a consumer which is connected for the first time to the broker will get the last message send for a specific topic. This can be useful if the temperature from a sensor should be displayed, but the value will only be fetched every 30 seconds. This would lead to the behaviour that the initial value on consumer side will in worst case be unknown for 30 seconds. *Persistent sessions* allows a connection be established even if the consumer disappers. In this case all the messages incurred will be stored and delivered if the consumer resume the session. The session can be identified with a unique client identifier.[43, cf.]

Due to the fact that MQTT is a simple protocol with a small footprint and the QoS handshake enables the protocol to be independent from TCP so that it can be used even on devices without a TCP/IP stack like embedded devices such as an Arduino.[43, cf.] There the protocol can be used via a bus or a serial port.[43, cf.] MQTT himself support the protection of the messages via username nad password and the communication can be encrypted with SSL or TLS on the transport layer.[43, cf.] The broker can additionally use client certificates to authenticate them or restrict the access via access conrol lists such as IP filtering.[43, cf.] MQTT is available for most of the common programming languages and platforms.

### 2.4.2 ZeroMQ

ZeroMQ is an messaging and communication framework to send atomic messages between applications and processec across various transports like Transmission Control Protocol (TCP), in-process, inter-process or multicast.[44, cf.] Every message will be send over sockets via several patterns like publish-subscribe, fan-out or a simple request-reply.[44, cf.] ZeroMQ has an asynchronous I/O model which allows to create high-performance and scalable multicore applications.[44, cf.] To distinguish it from messaging frameworks like AMQP[24] ZeroMQ has no dedicated broker inbetween. The benefit is that there is no single point of failure, not bottleneck and no need to maintain another component, but ZeroMQ still has the advantages of such a messaging system.

ZeroMQ supports five different transport types.
**In-Process** is used for local (in-process or inter-thread) communication transport. This transport passes the messages directly via memory between threads.[45, cf.] These transport type is optimal for creating multithreaded applications without providing access to the outside. This is the fastes tranport type available in ZeroMQ.
**Inter-Process Communication (IPC)** provides also a local communication transport, but passes messages via the OS dependend IPC machanism, for example UNIX domain sockets. An application can provide a local API for another local appliction 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.[46, cf.]
**Pragmatic General Multicast (PGM)** is a multicast communication transport using the PGM standard protocol[25] and the datagrams are layered directly on top of IP datagrams.[47, cf.] It is helpful to send a squence of packets to multiple consumers at the same time. Therefore PGM and also EPGM can only be used with the publish/subscribe pattern.
**Encapsulated Pragmatic General Multicast (EPGM)** is similar to PGM but with the difference that the PGM datagrams are encapsulated inside UDP datagrams.[47, cf.]

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

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

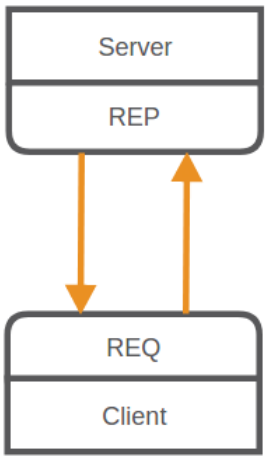


Figure 10: ZeroMQ Request-Reply architecture. Adapted from: [44]

The **Publish-Subscribe** pattern in ZeroMQ is basically comparable with the MQTT publish-susbcribe pattern, but without the broker inbetween. This ends up in the fact that every consumer has to now the publisher and has to connect to them. A direct connection between them will established. Similar to MQTT, the publisher will send out the message to every subscribed consumer. It is also possible to subscribe to more than one topic. In ZeroMQ this pattern is represented by PUB-SUB socket pairs. Figure 11 shows this pattern.

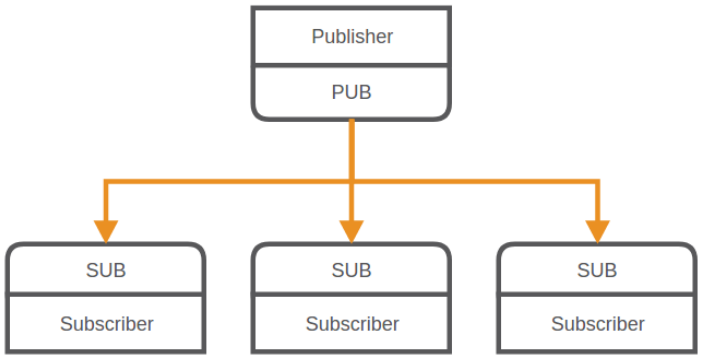


Figure 11: ZeroMQ Publish-Subscribe architecture. Adapted from: [44]

The **Pipeline** pattern is also known as the *Divide and Conquer* pattern. There we have Ventilator which produces multiple tasks that can be done in parallel, a set of worker that can process these tasks and a sink that collects the results from the workers.[44] Benefit of this pattern is, that the workers devide the tasks, this means it will increase the

calculation time based on the connected workers. FUrthermore the works can pull a new task if they are done. This means a worker has a minimal idle time. Queuing is provided by ZeroMQ. It is also possible to add and remove workers dynamically. This makes an application much more scalable. Finally the sink pull the data from the worker in a so called *fair-queuing*. This means the sink will pull one package from each worker one after another, then he starts from the beginning and will pull again only one package, even if one or multiple worker should have multiple packages retrievable. The whole pattern is shown in figure 12

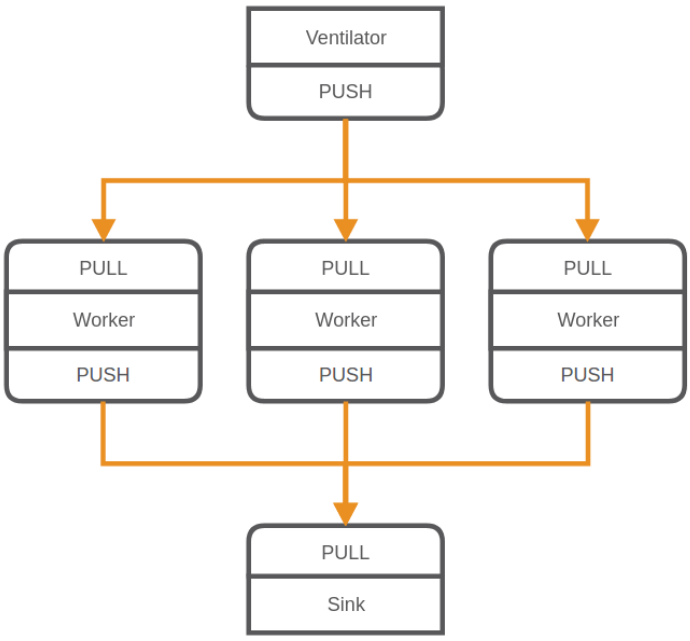


Figure 12: ZeroMQ Pipeline architecture. Adapted from: [44]

**Exclusive pair** connects tow sockets exclusively, for example two threads in a process.[44]
**Combinations of them** at least these some of the patterns in ZeroMQ. All of these patterns can be combined to create much more advanced combinations. Figure 13 illustrates such a combination.

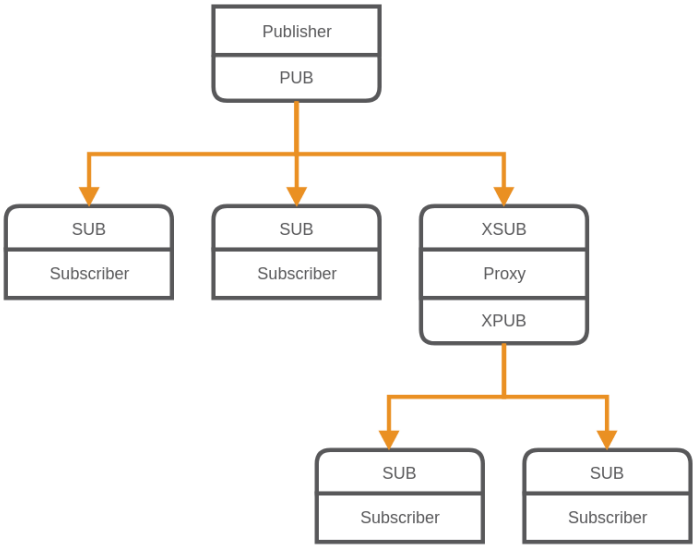


Figure 13: ZeroMQ combination of patterns. Adapted from: [44]

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

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

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

# Chapter 3

# Requirements Analysis

Based on the fundamentals the requirements for the plugin to be developed will be formulated in this chapter. Thereby aspects that will be relevant for the specific implementation will be considered.

## 3.1 Technical requirements

As the fundamental requirement, the prototype to be developed has to create, manage and maintain virtualized containers on a fog node. Tools like Open Baton inherently supports OpenStack as the ETSI MANO VIM layer. Most MANO tools like Open Baton uses OpenStack which deploys virtual machines to virtualize the NFs. This is a rock solid solution for a cloud environment. Unfortunately a bare-metal virtualization is most of the times not feasible on small power devices like they are used in the IoT area. Therefore OpenStack should be replaced by a more efficient and lightweight solution like container virtualization. The desired NFs can be bundeled 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 lifecylce 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 NF orchestration. A constraint can be a functional and non-functional constraint. For example a specific hardware component, like a sensor or a ZigBee dongle, which is necessary to execute the NF or a hardware requirement, like CPU power, RAM or disk space. It could also be a non-functional constraint, for example a specific software which has to be installed or a protocal which can be used. The local orchestrator should manage these constraints for itself and if necessary for all adjacent nodes and should consider them while choosing a suitable nodes for the desired NF. The whole functionality should work similar to the labels in Docker Swarm. There a Docker Swarm node can be have multiple labels, which can be considered when deploying an image. This behavior should be achieved by the fog node engine.

Some side 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 controled

via an API. The centralized cloud environment should be easiely replacable and should not be exclusively bound to Open Baton. Finally the whole system should be well tested and documented.

## 3.2 Functional requirements

After the general conditions for the prototype has been defined, the functional requirements will also be addressed. When selecting the programming environment special attention is paid to the compatibility of the Open Baton framework. This is particularly important as the effort to develop a prototype should be a s small as possible and the prototype should be natively applicable on the fog nodes. Open Baton has an build in plugin mechanism, which allows to replace for example the VIM driver or the VNFM so that any third party tool can be accessed via an API. Because Open Baton and the whole environment around them is open source, it would be also feasible to extend the system if necessary, but with the addition, that the system still follows the ETSI MANO conventions. Beside that the prototype should be flexible enough to also fit in any other system by providing an API to control the system.

Additional it would be useful to create a GUI for interacting with the prototype and show the functionallity of the system. This should only be a simple GUI for testing and demonstration purposes and should not be necessary for executing the final prototype. Later on the prototype should be controlled via the API from any other existing orchestration tool. A command line tool to control the system would be useful as well and eases the handling of the prototype.

## 3.3 Use-Case-Analysis

## 3.4 Delineation from existing solutions

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

**Kubernetes** is especially made for Docker and can orchestrate, scale and manage containers. It is open source, made by Google and one of the most popular orchestration tools out

there. It has on huge and pretty active community and is used by several well known companies[48] like ebay[1] and Wikimedia[2]. Due to the fact, that is exclusivly made for Docker, it means that the system is not made to easily switch the underlying container engine if needed.

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

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

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

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

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

**OpenStack**

**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.[49, cf.] It is also able to interact with multiple VIMs, containers, infrastructures and devices and due to the fact that it can be

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

extended with plugins, it is can be used together with several well known tools like OpenStack, Docker or even Kubernetes.[49, cf.] Through these flexibility Cloudify can also be used in an IoT envoironment if an appropriate VIMs plugin is used. Downside is that Cloudify himself is very limited without a powerfull underlaying orchestration tool.

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

## 3.5 Conclusion

# Chapter 4

# Concept

This chapter introduces the architectural design of plugin respect to the previously defined requirements. Therefore the used development environment will be analyzed. Followed by the initial architecture of the plugin to be developed. Based on that the different layer of the system will be elaborated.

## 4.1 Overview

## 4.2 Development environment

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

**Java** is still one of the most important programming languages out there.[50, cf.] Java is platform indipendent, because Java will not be executed directly in the OS, instead it will be executed in the JVM. The downside is, that especially because of the JVM, Java programms are much more inefficient regarding RAM and CPU usage. Java is object oriented, which makes it easy to extend. There are a lot of well designed and developed establised libraries and tools out there, which makes it easy to create rapid prototypes.

**Node.js** is a relativly young programming environment, which was created 2009 by Ryan Dahl and based on the Google JavaScript engine V8. The V8 engine is written in C++,

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

**Python** is one of the most used programming languages with a huge and active community.[50, cf.] It is a dynamic typed programming language with an automatic memory management which uses both, reference counting and garbage collection at the same time.[51, cf.] Python can use several programming paradigmns like the object oriented programming or functional programming. Core philosophy is make it simple, make it beautiful, make it explicit and make it readable. Currently two versions are maintained in parallel, 2.7 and the newer 3.6. Due to the fact that a huge codebase is still working with Python 2.x, this version is still under development.[52, cf.] Python has a tremendous amount of libraries which can be installed via the Python package manager called pip[1]. Similar to Node.js Python allows to write C extension. Finally there are several Python compilers which compiles Python code to other high-level languages like Java, C or JavaScript.

**Go** is the new emerging language created by Google and was announced end of 2009. It follows the traditions of C, but also takes added concepts to create a more modern programming language. According to their own statements go is expressive, concise, clean, and efficient.[53, cf.] Furthermore it has a garbage collection and run-time reflection, it is fast, statically typed and a compiled language.[53, cf.] Go is used in production by several tools like Docker, Kubernetes and OpenShift. Downside is that there are not so much libraries out right now and there are no long time experience in the critical mass.

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

## 4.3 Architecture of the system

The overall architecture of the system can be seperated into two levels. Figure 14 will point up the architecture. The first level is the *centralized fog level*. This could be for example a cloud server or management node in a fog cluster. Ideally this level should be implemented with an MANO compliant framework. Hereinafter Open Baton will referred as the tool of choice for that level. Main function will be the creation of the NFVI as well as the deployment

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

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

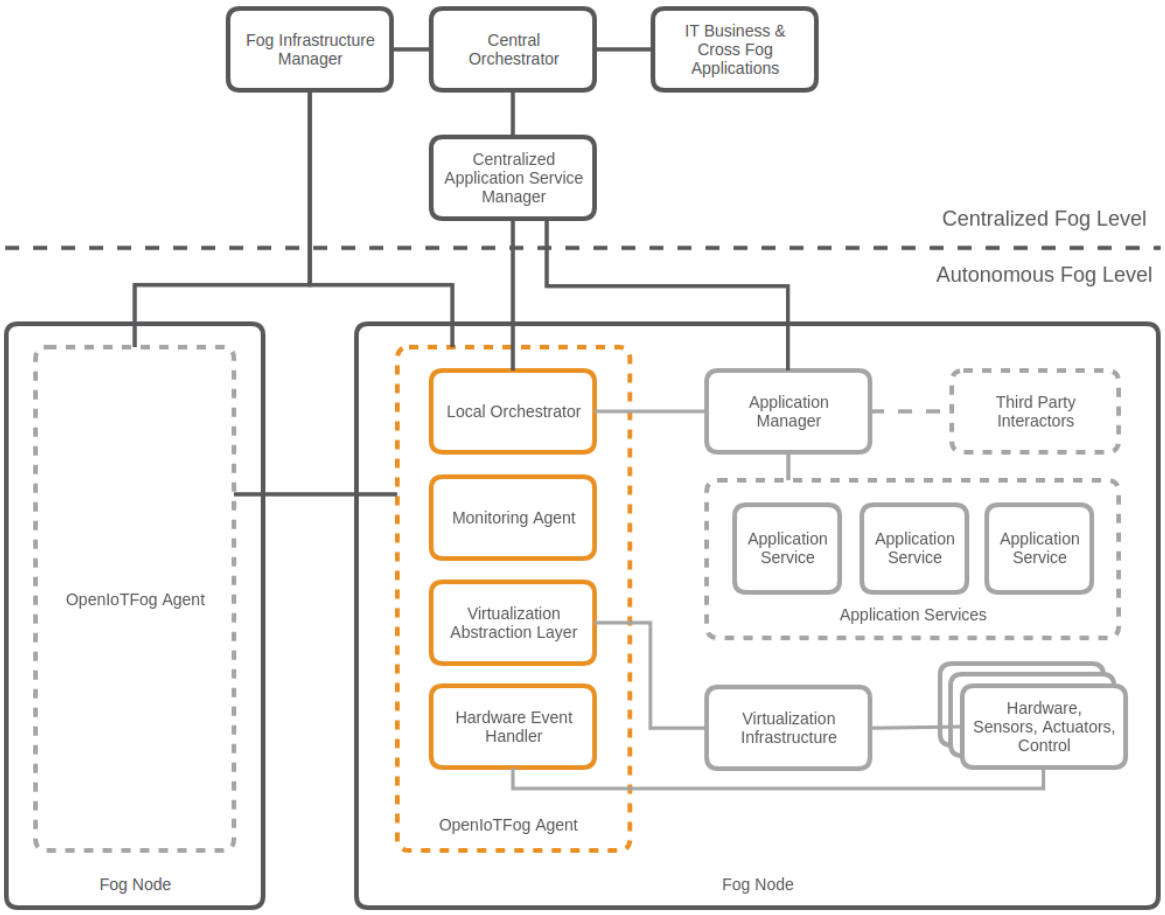


Figure 14: Abstract architecture design

The second level is the *autonomous fog level*. The prototype to be developed will be located in this level. It includes all existing fog nodes, as well as the prototype referenced as the *OpenIoTFog Agent* in figure 14. Each node must have a running instance of the OpenIoTFog Agent to be part of the system. Beside the prototype 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 the prototype. These infrastructure can create the containers and VNFs. Further it is also possible to have additional third party tools installed which can interact with the them as well.

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

One of the most important components is the *Local Orchestrator*. It is responsoible for the deployment of the containers as well as the inter-node communication. The latter is necessary to let the nodes act in an autonomously was, so that they can react to changing requirements, even when the centralized fog level disappears. Therefore each node must have knowledge and should be able to interact with each other. Beside that the local orchestrator is tightly coupled to the *Virtualization Abstraction Layer (VAL) Manager*. This is an abstraction layer for each virtualization component in the system, for example Docker or a bare-metal virtualization like XEN. These components should be implemented as plugins, so that it is easy to add or remove virtualization components. Therefore Yapsy[3] is used as the plugin system of choice.

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

### 4.3.1 Virtualization layer

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

### 4.3.2 Communication layer

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

The first subcomponent is the **node discovery**. The idea behind the node discovery is that each node automatically can register and unregister himself to the cluster. This means in the moment of the startup of a node, they will send out a message, with an information request about all subscribed nodes. To realize that MQTT will be the tool of choice. The MQTT broker will receive and forward the message to all subscribed nodes and each node will response with an information message to the broker again, which will send the message out again. Therefore each node will be up-to-date at each time. As long as there is no appearance or disappearance of any node, the network will not be stressed. To provide a better flexibility the MQTT broker can be executed in the centralized fog level or even on a node in the autonomous fog level. This allows the system to operate even if the centralized fog level disappears due to network issues or any other communication problems. It is also possible to let all

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

nodes communicate which each other once they shared their information. Smaller connection issues can be covered with this mechanism. As the MQTT broker Mosquitto[4] will be the tool of choice. It is an eclipse[5] project which means it is open source and under continous development. The project website descibes themself 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"[54].

The next subcomponent is the **inter- and intra-node communication**. Both kinds of communication will be realised with ZeroMQ. Some of the most important patterns and transport types in ZeroMQ was discribed 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 allways requires an IP to eastablish a connection to another node, therefore we had the node discovery which was described before. This allows us to connect to any other node in the cluster. In addition to the intra-node communication, also an inter-node communication will be implemented. It will be used to add new capabilities to a node. Therefore one or multiple third party applications should be connectable to a single ZeroMQ endpoint via the publish/subscribe pattern over IPC. Different to a normal publish/subscribe pattern, the endpoint will acts as the subscriber so that multiple publishers can push messages to them. Beside that each exposed socket should be configurable via the configuration file.

The last subcomponent is the **REST API**. It will be mainly used for the communication with the centralized fog level, because most of the orchestration tools out there are using REST for transfering data. In comparision to an implementation with ZeroMQ, this is much bigger overhead in terms of traffic and latency, but to have a better compatibility with other systems, this API will be implemented. As the tool of choice Flask[6] will be used. It is a lightweight and robust Python webserver, which is open source, well documented and under constant development. The REST API himself will follow the Hypermedia As The Engine Of Application State (HATEOAS) constraint with the addition that each endpoint will have a version number in the Uniform Resource Locator (URL) to ensure backwards compatability 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 communiction layer from the other components, the whole system can be maintained much easier and each subcomponent can be easy replaced by any other technology if necessary. This also makes the code more readable and leads up to a cleaner code strucutre.

### 4.3.3 Data layer

As mostly the data layer is used to persist necessary data. This includes the deployed services, adjacent nodes and the capabilities of the node. As database engine the lightweight document oriented database TinyDB[7] will be used. It is easy to use and has no execution overhead and is also good to use for small datasets. The whole data layer should be as abstracted as all the

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

other components before. Therefore repositories for each content type will facade the tinydb methods and allows the underlaying library, in this case TinyDB, to be replaced easily and without modifing several classes. The configuration of the databases should be stored in the global config file as well.

### 4.3.4 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 capabilites from third party apps via a ZeroMQ endpoint. As mentioned in the 4.3.2 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.

### 4.3.5 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 prototype himself. Primary it is used to handle the business logic of the deployment of new services. If a new service will be send to the communication layer it will be forwarded to the orchestration layer and will be parsed, validated and handled by them. This layer is also responsible for checking the necessary capabilities for each service component and will search for suitable nodes in the cluster if one or multiple capabilites are not fulfilled. Beside that, this component will also handle the lifecycle of a service during the deployment phase. more

### 4.3.6 User interface

### 4.3.7 Security

### 4.3.8 Continuous Integration

Continuous integration is nowadays a frequently used technique to automate repeating deployment steps into a self executing pipeline. It starts by running unit tests, code style checks and ends up with deploying the compiled programm to a server or a marktplace. Due to the fact that Github[8] will be used to host and maintain the git repository for the prototype, the pretty famous and semless integrated Travis CI[9] 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. Everytime a new commit will be pushed to the git remote repository, a new build will be started at Travis CI. At first a virtual machine will be started by Travis CI. Afterwards all necessary components like libraries and tools will be installed in the virtual machine. Finally all predefined tasks from the YAML file will be executed. In terms of the prototype, 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.

## 4.4 Conclusion

[8] `https://github.com`
[9] `https://travis-ci.org`

# Chapter 5

# Implementation

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

## 5.1 Environment

## 5.2 Project structure

- **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 informations 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 autogenerate 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, the current branch will be switched to *gh-pages* which is be used to display the github page at `https://neoklosch.github.io/Motey/` and a new commit will be pushed. Finally the branch will be switched back again.
- **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.
- **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.

### 5.2.1 motey engine

## 5.3 Used external libraries

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

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

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

Listing 1: Command line interface documentation for the daemon process

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

**dependency-injector** is a microframework for Dependency Injection (DI) in Python. The DI pattern allows to move the responsibility for creating a dependency from the concrete objects to a factory or a framework which creates the dependency graph. This grants the single responsibility concept for classes and makes the whole code base much easier to unit test, because a dummy object can be passed to the constructor of the class. It is also possible to mocked the object with the help of a mocking library. To realize DI in the prototype a so called *app_module.py* was created which uses the *dependency-injector* framework to create the dependency graph. Several IoC containers are created in that file and will be used by the framework to generate the glue code. Most of the injected components are instanciated 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 2 demonstrates the implementation of such an IoC container.

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

Listing 2: Extract of a sample IoC container from the app_module.py

## Docker Software Development Kit (SDK) Write it

**Flask** is a framework to create web applications. Flask does not provide any templating or database engine, nor does it enforce a specific file structure. Instead it will support extensions to add functionalities like that so that the developer can choose the tools of choice.[55, cf.] Nevertheless Flask is production ready and is used in several big projects like Pinterest[56] or Twilio[57]. The simplest sample implementation of a web application with one URL endpoint is shown in listing 3.

```
1 from flask import Flask
2 app = Flask(__name__)
3
4 @app.route('/')
5 def hello_world():
6     return 'Hello, World!'
```

Listing 3: Sample Flask web application

write more about Flask description what i do in code with it

**Logbook** is a small loging 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 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. The libarary allows to connect to a MQTT broker like the Mosquitto broker. It also comes with a variety of helper methods to eases the usage. A wrapper class to centralize the usage of the library was created and the configuration as well as some smaller improvements was made in this wrapper class. The whole configuration of the client can be configures via the global config file again. Furthermore the routes are managed in the wrapper and a after connect handler was implemented. It will be used to perform actions after a successfully created connection to the broker was made and all subscriptions to topics are done. This helps to realize the node discovery mechanism described in section 4.3.2.

**pyzmq**

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

#### Sphinx

**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 optimised concerning performance. But it is easy to use, has no execution overhead and it performs very well on smaller datasets. The main purpose of the library is to be used for small apps where database server like MySQL[2] or MongoDB[3] will be a huge overhead. Furthermore TinyDB has several extension to add more functionalities like indexing or caching. It also allows to easily extend the library with custom middlewares and extensions. description what i do in code with it

#### Yapsy

## 5.4 Important Implementation Aspects

## 5.5 Implementation of the data layer

[2] https://www.mysql.com
[3] https://www.mongodb.com

## 5.6 Implementation of the orchestration layer

## 5.7 Implementation of the virtualization layer

## 5.8 Implementation of the communication layer

## 5.9 Implementation of the capability management

## 5.10 Implementation of the user interface

## 5.11 Deployment and Continous Integration

## 5.12 Conclusion

# Chapter 6

# Evaluation

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

## 6.1 Test Environment

## 6.2 Experimental Validation

## 6.3 Performance Evaluation

## 6.4 Observational Validation

## 6.5 Deployments

## 6.6 Code Verification

## 6.7 Comparative Analysis

## 6.8 Conclusion

# Chapter 7

# Conclusion

## 7.1 Summary

## 7.2 Dissemination

## 7.3 Impact

## 7.4 Outlook

# Acronyms

| | |
|---|---|
| **AE** | Autoscaling Engine |
| **AMQP** | Advanced Message Queuing Protocol |
| **API** | Application Programming Interface |
| **BSS** | Business Support Systems |
| **CapEx** | Capital Expenditure |
| **CI** | Continuous Integration |
| **CLI** | Command Line Interface |
| **CPU** | Central Processing Unit |
| **CPS** | Cyber-Physical System |
| **CS** | Cyber System |
| **DI** | Dependency Injection |
| **EPGM** | Encapsulated Pragmatic General Multicast |
| **EMS** | Element Management System |
| **ETSI** | European Telecommunications Standards Institute |
| **FM** | Fault Management |
| **FOKUS** | Fraunhofer-Institut für Offene Kommunikationssysteme |
| **GUI** | Graphical User Interface |
| **H2H** | Human-to-Human |
| **H2M** | Human-to-Machine |
| **HATEOAS** | Hypermedia As The Engine Of Application State |
| **HTTP** | Hypertext Transfer Protocol |
| **IDE** | Integrated Development Environment |
| **IERC** | European Research Cluster on the Internet of Things |
| **IoC** | Inversion of Control |
| **IoS** | Internet of Services |
| **IoT** | Internet of Things |
| **IP** | Internet Protocol |
| **IPC** | Inter-Process Communication |
| **IT** | Information Technology |
| **JSON** | JavaScript Object Notation |
| **JVM** | Java Virtual Machine |
| **LXC** | Linux Containers |
| **M2M** | Machine-to-Machine |
| **MANO** | Management And Orchestration |
| **MQTT** | Message Queue Telemetry Transport |
| **NF** | Network Function |
| **NFV** | Network Function Virtualistion |
| **NFVI** | Network Function Virtualization Infrastructure |
| **NFV-MANO** | Network Function Virtualistion Management And Orchestration |
| **NFVO** | Network Function Virtualistion Orchestrator |
| **NIST** | National Institute of Standards and Technology |
| **OASIS** | Organization for the Advancement of Structured Information Standards |
| **OpEx** | Operating Expense |
| **OS** | Operating System |
| **OSS** | Operations Support Systems |
| **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 |
| **TLS** | Transport Layer Security |
| **TOSCA** | Topology and Orchestration Specification for Cloud Applications |
| **UI** | User Interface |
| **URL** | Uniform Resource Locator |
| **VAL** | Virtualization Abstraction Layer |
| **VIM** | Virtual Infrastructure Manager |
| **VM** | Virtual Machine |
| **VMM** | Virtual Machine Monitor |
| **VNF** | Virtual Network Function |
| **VNFM** | Virtual Network Function Manager |
| **YAML** | YAML Aint́ Markup Language |

# Glossary

**Algorithmus** a

**Chiffrierung** a

**Dechiffrierung** a

# 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] 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).

[3] 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).

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

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

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

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

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

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

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

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

[12] 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).

[13] 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).

[14] 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).

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

[16] *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).

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

[18] 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).

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

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

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

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

[23] *Network Function Virtualistion (NFV); Architectural Framework*. Tech. rep. Accessed: 2017-02-27. 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).

[24] 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).

[25] 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).

[26] *NFV*. Accessed: 2017-03-19. URL: https://sdn-wiki.fokus.fraunhofer.de/doku.php?id=nfv (cit. on p. 11).

[27] 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).

[28] *Why is TOSCA Relevant to NFV? Explanation*. Accessed: 2017-03-19. URL: https://www.sdxcentral.com/nfv/definitions/tosca-nfv-explanation (cit. on pp. 12, 13).

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

[30] *Docker Overview - Docker*. https://docs.docker.com/engine/understanding-docker/. Accessed: 2017-02-24 (cit. on p. 14).

[31] B. Grant. *Kubernetes: a platform for automating deployment, scaling, and operations*. Accessed: 2017-02-27. Nov. 2015. URL: https://www.slideshare.net/BrianGrant11/wso2con-us-2015-kubernetes-a-platform-for-automating-deployment-scaling-and-operations (cit. on p. 15).

[32] J. MSV. *Kubernetes architecture*. Accessed: 2017-03-03. Oct. 2016. URL: https://www.slideshare.net/janakiramm/kubernetes-architecture (cit. on pp. 15, 16).

[33] E. Mulyana. *Kubernetes Basics*. Accessed: 2017-03-03. May 2016. URL: https://www.slideshare.net/e2m/kubernetes-basics (cit. on pp. 15, 16).

[34] *Pods - Kubernetes*. Accessed: 2017-03-03. Dec. 2016. URL: https://kubernetes.io/docs/user-guide/pods (cit. on p. 15).

[35] *Labels and Selectors - Kubernetes*. Accessed: 2017-03-03. Dec. 2016. URL: https://kubernetes.io/docs/user-guide/labels (cit. on p. 16).

[36] *kube-proxy - Kubernetes*. Accessed: 2017-03-03. Dec. 2016. URL: https://kubernetes.io/docs/admin/kube-proxy (cit. on p. 16).

[37] *Replication Controller - Kubernetes*. Accessed: 2017-03-03. Dec. 2016. URL: https://kubernetes.io/docs/user-guide/replication-controller (cit. on p. 16).

[38] *Rolling Updates - Kubernetes*. Accessed: 2017-03-03. Dec. 2016. URL: https://kubernetes.io/docs/user-guide/rolling-updates (cit. on p. 16).

[39] *Docker Swarm Documentation*. https://docs.docker.com/engine/swarm. Accessed: 2017-03-18 (cit. on p. 16).

[40] *OpenBaton Documentation*. http://openbaton.github.io/documentation. Accessed: 2017-03-18 (cit. on pp. 16–18).

[41] *FAQ - Frequently Asked Questions | MQTT*. Accessed: 2017-06-03. URL: http://mqtt.org/faq (cit. on p. 18).

[42] 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. 18).

[43] 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. 18–20).

[44] *ØMQ - The Guide*. Accessed: 2017-06-06. URL: http://zguide.zeromq.org/page:all (cit. on pp. 20–23).
[45] *zmq_inproc(7) - 0MQ Api*. Accessed: 2017-06-06. URL: http://api.zeromq.org/3-2:zmq-inproc (cit. on p. 20).
[46] *zmq_tcp(7) - 0MQ Api*. Accessed: 2017-06-06. URL: http://api.zeromq.org/3-2:zmq-tcp (cit. on p. 20).
[47] *zmq_pgm(7) - 0MQ Api*. Accessed: 2017-06-06. URL: http://api.zeromq.org/3-2:zmq-pgm (cit. on p. 20).
[48] T. K. Authors. *A Cheat Sheet for Understanding "NFV Architecture"*. Accessed: 2017-04-10. URL: https://kubernetes.io/case-studies (cit. on p. 26).
[49] 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 pp. 26, 27).
[50] *Most Used Programming Languages 2017: The Trendiest & Most Sought After Coding Languages*. Accessed: 2017-06-16. URL: https://stackify.com/trendiest-programming-languages-hottest-sought-programming-languages-2017 (cit. on pp. 28, 29).
[51] *Python Garbage Collection - Digi Developer*. Accessed: 2017-06-17. URL: https://www.digi.com/wiki/developer/index.php/Python_Garbage_Collection (cit. on p. 29).
[52] 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. 29).
[53] *Documentation - The Go Programming Language*. Accessed: 2017-06-16. URL: https://golang.org/doc (cit. on p. 29).
[54] *Eclipse Mosquitto*. Accessed: 2017-06-15. URL: https://projects.eclipse.org/projects/technology.mosquitto (cit. on p. 32).
[55] *Foreword - Flask Documentation*. Accessed: 2017-06-18. URL: http://flask.pocoo.org/docs/0.12/foreword (cit. on p. 38).
[56] *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. 38).
[57] *Introducing Flask-RESTful*. Accessed: 2017-06-18. URL: https://www.twilio.com/engineering/2012/10/18/open-sourcing-flask-restful (cit. on p. 38).